CIFE CENTER FOR INTEGRATED FACILITY ENGINEERING

# Integrated Case-Based Reasoning for Structural Design

by

Jenmu Wang
H. Craig Howard

**TECHNICAL REPORT**
**Number 58**

October, 1991

**Stanford University**

If you would like to contact the authors please write to:

*c/o CIFE, Civil Engineering,*
*Stanford University,*
*Terman Engineering Center*
*Mail Code: 4020*
*Stanford, CA 95305-4020*

# Abstract

Recent knowledge-based expert systems for structural engineering design have focused on case-independent knowledge (abstract reasoning rules for designing), and while great strides have been made in that area, there is still a significant need to develop systems to take advantage of the wealth of knowledge contained in every substantial structural design. On the other hand, previous database-oriented design efforts have focused primarily on knowledge-poor databases of solutions, in which the traditional engineering handbook of solutions has simply been replaced by digital data. The challenge is to find a way to capture and apply the kind of case-dependent knowledge that structural engineers have traditionally used.

This thesis examines the new approach of implementing knowledge-based structural design systems using both case-dependent and case-independent knowledge. The resulting system, DDIS[1], combines case-based reasoning with case-independent reasoning in a blackboard framework. In the blackboard model, the knowledge needed to solve a problem is partitioned into independent knowledge sources that are grouped into several knowledge modules in the knowledge base. The knowledge sources communicate design results via a global knowledge structure (the blackboard) and respond opportunistically to the changes on the blackboard. DDIS has two major knowledge modules: case-dependent and case-independent. The case-independent module represents abstract knowledge about the problem domain and problem solving strategies. The case-dependent module uses case-based reasoning techniques to transfer knowledge from previous designs to current design tasks. Using the blackboard control mechanism and the two knowledge modules, DDIS can apply both case-dependent and case-independent knowledge to perform collaborative and opportunistic design.

The most important elements of case-dependent knowledge identified in this study are design solutions, justifications, constraints, failures, plans and goals. Design solutions include final solutions, intermediate solutions and partial solutions. Design justifications are the calculations of previous design variables and the dependences of their values. Design constraints are used to evaluate designs and are very important information for understanding the history of a design case. Previous design failures can be used to avoid unsuccessful design alternatives in the future. Design plans are the strategies used to solve a design problem. Specific knowledge about how to achieve a particular design step is

---

1 DDIS stands for Design-Dependent and Design-Independent System. "Design-dependent" and "design-independent" are the terms used for "case-dependent" and "case-independent" in the early stage of this research.

contained in a design goal. Design goals are included in plans to form complete case-dependent control knowledge of a case.

DDIS has a very flexible architecture and representation that can use any subset of the above mentioned case-dependent knowledge. Past design solutions can be applied to very similar new designs while previous design plans can be applied to guide designs with less surface similarity. However, this study did not address the similarity problem. DDIS relies on users to retrieve relevant designs from the case memory and to decide how similar they are to the new design.

Two structural design applications have been built using DDIS to demonstrate its integrated design approach. The demonstration applications design structural steel beam-columns and anchor base plates for electrical transmission poles.

# Acknowledgements

This report is based on the Ph.D. thesis submitted by Jenmu Wang to the Department of Civil Engineering, Stanford University. The doctoral committee comprised of Professor H. Craig Howard, Dr. Barbara Hayes-Roth, Professor Helmut Krawinkler and Professor Kincho H. Law. Financial support from the National Science Foundation (Grant No. MSM-8958316) and the Center for Integrated Facility Engineering (CIFE) at Stanford University is gratefully acknowledged.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1
# Introduction

Design databases represent design solutions without capturing the knowledge behind them. Rule-based expert systems capture *case-independent knowledge*—abstract reasoning rules independent of specific designs. To more fully capture the kind of knowledge employed by experienced designers, knowledge-based design systems need to incorporate *case-dependent knowledge*—a memory of good (and bad) designs and design strategies together with the rationale that supports them.

Previous knowledge-based systems for structural design are based primarily on case-independent knowledge. This research combines case-dependent and case-independent knowledge in an integrated, knowledge-based structural design system. The case-independent components use rule-based and frame-based methods to represent abstract knowledge about the problem domain and problem solving strategies. The case-dependent components use case-based reasoning techniques to transfer knowledge from previous designs to current design tasks.

The overall objectives of this research are to formalize case-dependent knowledge in the structural engineering domain, to demonstrate the feasibility of using case-dependent knowledge in knowledge-based structural design systems, and to develop a prototype integrated design framework utilizing case-dependent and case-independent knowledge cooperatively.

## 1.1. Motivation

This study arises from an intersection of interests in design databases and knowledge-based expert systems. In looking at the problems of representing engineering design data in formal databases and CADD (computerized drafting and design) systems, it quickly becomes apparent that something is missing. The computerized design database fails to capture the reasoning behind the design (as did the paper plans of the manual era); i.e., the database represents the *what* of the design, but not the *why*. Taken as a whole, the design database represents the problem solution as data without knowledge. If that exact problem is to be solved again, then the design task is trivial—just a database look-up. However, the challenge of design is to solve new and different problems. Thus the use of the design database in future design tasks requires that the designer recall at least part of the original reasoning chain to determine where the design or parts of the design might be applicable.

In contrast to the design database approach, knowledge-based expert systems attempt to codify the abstract reasoning processes of the expert into "if-then" rules; i.e., the expert system captures the *why*, but not the *what*. The expert system knowledge can be characterized as **case-independent knowledge**. Examples of case-independent knowledge from the structural engineering domain are:

**IF** the component is a column
**THEN** area and radius of gyration are the critical dimensions

**IF** the component is a beam
**THEN** moment of inertia and depth are the critical dimensions

Experienced designers do not design strictly by abstract reasoning processes, nor do they exhaustively search a space of previous design solutions, testing whether each exactly matches the current design criteria. Because they have typically performed many similar design tasks to reach a level that we would term "experienced", these expert designers have been exposed to a wide variety of design problems and the reasoning processes associated with those design problems. Therefore, the experienced design professional has a memory of good (and bad) designs and knows the rationale behind each of them. Potentially, the designer may have generalized some of this experience into abstract reasoning rules, but most of the experience is still in the form of **case-dependent knowledge**—knowledge about specific previous designs and their supporting reasoning.

Previous knowledge-based systems for structural design such as HI-RISE [Maher 85] and DESTINY [Sriram 86] are based primarily on case-independent knowledge (see Section 2.2 for more about these systems). The use of case-dependent knowledge in structural engineering can have significant benefits in the performance of the design process, in the capture and formalization of the engineer's design knowledge, and in our understanding of the art and science of structural design. This study presents a prototype solution to integrate case-dependent and case-independent knowledge in a knowledge-based design system and explores the issues that arise in the development of the system.

## 1.2. Definition of Case-Dependent Knowledge

The design of engineering structures is a complex process requiring knowledge of structural material properties, mechanics of materials, structural analysis and design specifications, in combination with experience built up over years of practice. The experiential knowledge ranges from detailed positive and negative experiences associated with specific design cases to abstract heuristics and general rules of thumb generalized from many projects. The former is what we call *case-dependent knowledge*, and the latter is *case-independent knowledge*.

Case-dependent knowledge is the knowledge about specific previous designs and their supporting reasoning, including previous design solutions, plans, assumptions, history, decisions and the rationale behind design decisions. On the other hand, case-independent knowledge is generalized from many design experiences. It is independent of specific design cases and can be applied generally to the problem domain.

There are many computer programs that provide algorithmic solutions to structural design. A number of prototype expert systems [Maher 85, Garrett 86, Sriram 86, etc.] have been developed to integrated abstracted reasoning heuristics (i.e., case-independent knowledge) with algorithmic design tools. Current development of case-based reasoning in artificial intelligence has made it possible to capture the case-dependent knowledge into knowledge based system.

To apply case-based reasoning techniques using case-dependent knowledge, we must first identify those aspects of the case-dependent knowledge that can have a positive impact on the solution process. In the discussion below, we examine some contrasting views of potentially useful case-dependent knowledge.

- **Design solutions and design plans**—Frequently, previous design solutions provide good starting points for new design problems. In modifying a previous solution, knowing the reasoning behind that solution permits the designer to quickly ascertain whether the proposed changes violate any of the fundamental constraints on that design or what interactions those changes may have with other aspects of the design. Similarly, it is as important to capture the solution strategy (i.e., the plan) and the associated rationale as it is to capture the actual solution. While a specific design solution can be applied to a small range of similar problems, a previous solution strategy can be applied to generate a completely different design.

- **System designs and component designs**—The granularity of the case-dependent knowledge is important. From a single complete structural design case, a designer can reason about one component (e.g., a standard beam), about a module composed of several components systems (e.g., a floor slab with its supporting beams), or the entire structural system composed of many components (e.g., a building frame). Each complete design contains a wealth of knowledge at all three levels.

- **Good designs and bad designs**—Since almost all design tasks require multiple iterations, designers may actually have more knowledge about bad solutions and solution strategies then they do about good ones. Therefore, it is important to capture this knowledge to eliminate as many alternatives as possible. In particular, knowledge about design failures can be used to bring downstream constraints into consideration at the very early stages of the design

process. For instance, a concrete beam design may satisfy all applicable building codes but be impractical to build on the construction site.

These pairs of ideas are intended to represent contrasting ends of non-exclusive ranges. The full spectrum of useful case-dependent knowledge is constituted from the intersection of these ranges and others as well. The objective is to capture knowledge about *good* component solutions, *bad* system design plans, etc. A case-dependent, knowledge-based system should be able to apply all the knowledge available from previous design tasks to the improvement of the current design.

## 1.3. Objectives

The purpose of this research is to explore the integration of case-dependent and case-independent knowledge in a knowledge-based structural design system. The major objectives are to:

- **Formalize case-dependent knowledge in structural engineering design**—Although case-dependent knowledge is commonly used by human structural engineering designers, it has not been formalized for applications in computerized design systems. The first objective of this research is to identify and formalize the case-dependent knowledge that is required for case-based inference in structural engineering design; i.e., *what* knowledge and data is important and useful in future design tasks and *how* should that knowledge and data be captured and represented.

- **Develop a prototype knowledge-based design environment integrating both case-dependent and case-independent knowledge**—A prototype computer system has been implemented to serve as an experimental framework so that the new approach of implementing a knowledge-based design system using both case-dependent and case-independent knowledge can be examined.

- **Demonstrate the feasibility of using case-dependent knowledge in knowledge-based structural design system**—Previous knowledge-based systems for structural design are based primarily on case-independent knowledge. This research focuses on how case-dependent knowledge can be used effectively within the design process: identifying similar design solutions, applying previously successful design plans, etc. The prototype system is demonstrated with two structural engineering applications: steel beam-column design and anchor base plate design for electrical transmission poles.

It is important to point out that the long-term goal is <u>not</u> to build a "black-box" designer. There are many aspects of problem-solving that humans perform very well, and

our goal is to improve the interaction of computer tools and human designers by constructing intelligent design assistants that support rather control the design process. The aim with this research is to investigate a new approach of implementing those intelligent design assistants. Although the knowledge acquisition and explanation interface were not the emphasis of this project, they are taken into account in the overall system architecture and considered as an influential factor throughout the project so that later expansion is possible.

## 1.4. Organization

The rest of this thesis is organized as follows:

- **Chapter 2** reviews background information relevant to this research. Two reasoning paradigms, rule-based reasoning and case-based reasoning, are examined. This chapter also reviews related efforts in case-based reasoning and research in knowledge-based systems.

- **Chapter 3** constructs an integrated design model based on the integration of rule-based and case-based reasoning paradigms. The overview of this integrated, knowledge-based design model and the functionality of its elements are described in this chapter.

- **Chapter 4** introduces DDIS, the prototype system implementing the integrated design model. The overall functionality and architecture of DDIS are presented in the overview section. Then, the details of the elements comprising the system are reported. The last section of this chapter summarizes the implementation issues.

- **Chapter 5** describes the two demonstration applications with some illustrative design examples. The intent of this chapter is to help the reader understand the design approaches that are available in the integrated, case-dependent and case-independent environment of DDIS.

- **Chapter 6** summarizes this research and discusses DDIS in terms of its successes and shortcomings.

# Chapter 2
# Background

This chapter presents background information relevant to this research. The first section describes two different inference methodologies used by knowledge-based expert systems (KBES): rule-based reasoning and case-based reasoning. The second section reviews previous KBES research in structural engineering design (and related disciplines). The final section discusses a number of case-based design systems.

## 2.1. Reasoning Paradigms

The inference methodology of expert systems can be divided into two broad categories: rule-based reasoning and case-based reasoning. This division is based on problem-solving approaches rather than knowledge representation methods. Generally speaking, a rule-based reasoning system relies on abstracted heuristic knowledge to solve problems, but a case-based reasoning system relies on similar previous cases to solve problems.

Since 1976 when MYCIN [Shortliffe 76] was developed as part of the Stanford Heuristic Programming Project, rule-based reasoning has been the most popular expert system reasoning paradigm. However, case-based reasoning is based on the fact that human beings consider not only rules but also previous cases in problem solving. In the following subsections, these two reasoning paradigms are examined in more detail.

### 2.1.1. Rule-Based Reasoning

Rule-based reasoning can be defined as the application of heuristic and causal knowledge encoded as formal rules. A rule-based reasoning system reasons by heuristics drawn from experience, judgment or insight of domain experts. Mostly, these domain-specific heuristics are represented as rules (also called *productions* or *production rules*) in a knowledge base. However, the knowledge representation of rule-based reasoning systems is not restricted to production rules. For example, frame-based methods can also be included to store object-oriented knowledge.

There is a large body of literature discussing knowledge representation for knowledge-based systems (e.g., see [Brachman 85] and Chapter 7 of [Waterman 86]), but that is not the intention here. The rule-based knowledge representation method will be described briefly in order to illustrate the classification of a rule-based reasoning system.

Production rules are conditional statements that have the form:

IF *conditions* THEN *actions*.

The IF part is called the left-hand side (LHS) of the rule, which can be any fact that can be matched to fire the rule. The THEN part, which is called the right-hand side (RHS) of the rule, can be problem-solving actions or conclusions.

The problem space of a rule-based system can be viewed as consisting of many independent states (e.g., a initial state, intermediate states and a goal state), and a rule is a legal transition from one state to another. Once the IF part (LHS) of a rule has been matched, the THEN part (RHS) of the rule can be executed. The execution of a rule adds new information to the problem space or alters the old context. Therefore, a problem is at a different state after a rule is fired.

A rule-based system performs its problem-solving task either by applying rules to move from an initial state toward a goal state (*forward chaining*) or by applying rules to move from a goal state (or hypothesis) toward supporting data (*backward chaining*). The expert system's *inference engine* determines the order in which the rules are to be applied in a particular problem. Given the initial state of a problem (i.e., a set of facts), the inference engine uses knowledge in the knowledge base (i.e., production rules) to form a line of reasoning which denotes a path toward the goal state.

### 2.1.2. Case-Based Reasoning

Case-based reasoning uses earlier experience of similar situations to help solve new problems. Sometimes called reasoning by analogy, case-based reasoning is an important problem-solving strategy that humans use frequently. When we encounter a problem, we recall a similar one that we solved before, compare the two problems, solve the new one analogously, and store it as part of our memory. While humans apply case-based reasoning automatically to a wide range of tasks, programming computers to recognize problem similarities and to transfer complete or partial solutions with analogic reasoning has proven to be very difficult.

Case-based reasoning is distinguished from rule-based reasoning system in two ways. First, it uses the knowledge associated with previous individual cases rather than abstracted heuristic rules to assist the current reasoning process. Second, its ability to reuse its experience is actually an automated learning process. Therefore, case-based reasoning is a powerful reasoning paradigm, especially for incomplete and intractable domains such as design and planning.

In general, case-based reasoning involves three stages:

- The *access* (or retrieval) phase involves the location of previous cases (stored in the memory) that are similar to a new case and the comparison of pertinent features that are critical to a particular class of problems. Then, the best feasible case(s) is selected as the source of analogy.

- At the *mapping* stage, correspondences between the source case and the new case are established to analyze differences and find similarities between them.

- Once the mapping is complete, *analogical inference* is applied to transfer knowledge from the source case to the new case based on the similarities identified. The identified differences also help to modify the previous experience to fit the new situation.

To reason by analogy, we must first have some cases to reason from and a memory structure in which to store them. The case memory is the resource of analogical problem solving. The functionality of its organization is to index the cases by their use and make it easy to access them. On the other hand, to retrieve a case that best matches the situation of a new problem, a case-based reasoning system needs to have some metrics that judge partial matches and serve to choose between potential cases. Therefore, the major issue in the access phase is finding a case indexing and retrieval mechanism that can work together.

Currently, there are some fairly good memory models that can represent knowledge and cases in a computer program. For examples, *semantic network* memory model [Quillian 68] typically represents static information and *episodic memory* model [Tulving 83] uses temporally related information (e.g., events, scenes, occurrences, etc.).

With regard to analogical inference, there are three approaches in general. Carbonell's two theories, *transformational analogy* [Carbonell 82] and *derivational analogy* [Carbonell 85], are useful in transferring solution and reasoning strategy of previous successes, while *failure driven learning* [Schank 81, Hammond 86] makes use of previous failures to avoid repeating the same mistakes. Transformational analogy transfers the previous solution of a similar problem to the new problem and modifies it based on differences between the problems. The solution modification is performed by predefined transformation operators and guided by heuristic rules. Derivational analogy uses the past reasoning process to reconstruct a solution for the new problem. The decisions that underlie the past solution are justified by the new problem situation. Violations are overcome by searching for alternatives.

Two case-based reasoning tools are currently available: CBR Express™ (Inference Corporation) and ReMind™ (Cognitive Systems, Inc.). They are recently commercialized case-based reasoning technology. However, the functions of the two systems are limited—emphasizing on the search and retrieval mechanism as well as user interfaces for

entering cases and browsing memory. They are not well suited for solving the kind of structural engineering design problems that this study is interested in.

## 2.2. Knowledge-Based Systems in Structural Engineering Design

Knowledge-based system prototypes have been developed for a wide range of civil engineering problems. A survey of those applications can be found in [Maher 87]. In this section, we will focus on structural design systems and notable design systems from related engineering fields.

Three of the most significant knowledge-based systems for structural design are HI-RISE [Maher 85] (for preliminary design of high-rise structures), SPEX [Garrett 86] (structural component design) and DESTINY [Sriram 86] (integrated building structure design). The first two systems will be described in more detail shortly. Other structural design systems include SSPG [Adeli 86a], BDES [Biswas 87], RTEXPERT [Adeli 86b], and truss design [Chan 87] (exploratory design using constraints). SSPG is a system for design of stiffened steel plate girders. To start the design, 300 production rules are used to select the ratio of the depth of the web and the length of the span (h/L). BDES and RTEXPERT are microcomputer-based expert systems. BDES designs small to medium span highway bridges. Heuristic rules are used to support the front-end decision-making process (e.g., choice of templates from a finite number of patterns of configuration, selection components of the template, and preliminary sizing of the selected components). RTEXPERT is developed using INSIGHT 2+, a rule-based expert system shell implemented on IBM PC, for roof truss design.

There are several experimental systems developed for other engineering design problems that are of interest in this research. AIR-CYL [Brown 84] designs specialized air cylinder. PRIDE [Mittal 86] designs paper handling systems inside copiers using a generalized design plan represented as an object-oriented model with design rules associated. DOMINIC [Howe 86] is a domain-independent system for mechanical engineering design derived from Dixon's earlier V-belt and shaft design KBES [Dixon 84]. DOMINIC takes a domain-independent approach to design using an evaluate-and-redesign architecture that applies a best-first search and domain-specific heuristic knowledge.

All the systems mentioned above are based primarily on case-independent knowledge. In order to fully illustrate this point, two systems are described in more detail below, namely, HI-RISE and SPEX. The discussion emphasizes the kinds of knowledge that they apply to the design process.

- HI-RISE [Maher 85] is a knowledge-based expert system for the preliminary design of high-rise buildings. The design process model used by HI-RISE is divided into three stages: synthesis, analysis, and evaluation. During the

synthesis process, HI-RISE employs heuristic elimination rules to guide the search in constructing all feasible structural configurations. The following are the typical elimination rules (in English translation) that are used to select the lateral load resisting system:

IF number of stories > 50 AND 3D system is core
THEN alternative is not feasible

IF 3D system is tube AND 2D system is solid shear wall
THEN alternative is not feasible

IF 2D system is rigid frame AND material is concrete
AND number of stories > 20
THEN alternative is not feasible

During the analysis task, HI-RISE must initialize some parameters (e.g., geometric and material properties of components) to evaluate the feasibility constraints of each alternative. The parameter values are selected using heuristic rules such as those that follow:

- Typically a W14 section is used for steel column design.
- The depth of a reinforced concrete slab is one twenty-eighth of its span.
- A double angle section is usually used for braced frame diagonal design.

Both the elimination rules and parameter selection rules represent the abstracted heuristic knowledge in high-rise building design. They are independent of specific design cases and can be applied generally to the problem domain. So, the knowledge that drives the inference of HI-RISE can be characterized as rule-based, case-independent knowledge.

- SPEX (Standards Processing Expert) [Garrett 86] is a knowledge-based, specification-independent structural component design system. SPEX selects a few key behavior limitations that govern the current design problem and locates the corresponding design requirements in the standard. Then, a set of constraints is constructed from the logical content of each requirement and other known structural, material and geometric relationship. Once the constraint set has been generated, the design can be solved as a optimization problem bounded by these constraints. The major role of design heuristics in SPEX is to generate or complete the design focus hypothesis identifying the key behavior limitations. The design heuristics are represented as abstracted reasoning rules called hypothesis generation rules in SPEX. An example is showed below in English translation:

IF the component type is column
AND the unbraced length about the x-axis is long
AND the unbraced length about the y-axis is long
AND the hypothesis is NIL
THEN the complete design focus hypothesis is
long column buckling about the x-axis due to axial compression
AND long column buckling about the y-axis due to axial compression

In addition to selecting governing behavior limitations, heuristic rules are also used for backtracking control, constraint set modification and hypothesis modification. As with HI-RISE, SPEX can be categorized as a rule-based reasoning expert system using case-independent knowledge.

## 2.3. Related Case-Based Reasoning Efforts

Analogy has been an active research topic in cognitive psychology for many years [Winston 80, Gentner 83, Kedar-Cabelli 85, Prieditis 88, etc.]. Recently, increasing attention is given to case-based reasoning in the AI community. A variety of recent efforts can be found in Kolodner 88, AAAI 88, DARPA 89, AAAI 90, Slade 91]. However, using previous experiences to solve a new design problem analogously is a relatively new research topic in applying knowledge-based system in the design process.

Some of the applicable analogy techniques have been implemented recently in a number of design systems, which include STRUPLE, CADSYN, CYCLOPS, ARGO and BOGART.[1]

- STRUPLE [Zhao 88] is a prototype system that uses previous design solutions to identify the relevant design elements for a structural design synthesizer (e.g., HI-RISE). The system applies the transformational analogy technique to transfer a set of appropriate design elements to a new design as the design vocabulary so that the search space of the new design is better confined. In STRUPLE, case-based reasoning is used only at the beginning of the design process to identify the most likely solutions.

- CADSYN [Maher 91] is an extension of EDESYN [Maher 88] (a domain independent inference mechanism for design synthesis derived from HI-RISE)

[1] This list is not intended to be comprehensive, but rather a sampling of the recent work that we find relevant to our projects.

to accommodate case-based reasoning. At every subsystem design level, CADSYN searches its case memory for relevant cases using pattern matching. If a relevant case is found, it uses adaptation rules to transfer solutions from a relevant case to a new design before EDESYN's design constraints and decomposition rules are used. CADSYN uses case-based reasoning only for adapting previous solutions as initial trials. The system, however, can reuse solutions at different levels of abstraction during various stages of the design process.

- CYCLOPS [Navinchandra 88] is a problem-solver that uses case-based reasoning as one of its strategies for solving landscape design and planning problems. The system focuses on the use of past failures and their repairs. Causal explanations of the goals and subgoals of previous cases are stored in a semantic network memory. Therefore, potential problems can be recognized, and cases can be identified and retrieved by demand posting technique. To repair a problem, CYCLOPS analogically transfers the previous solution to the new situation.

- ARGO [Huhns 87] is an analogical reasoning system that has been applied to the design of VLSI digital circuits. It constructs micro rules with pre and post conditions to represent the design plan of a design session, and the micro rules can be reused to reduce the amount of search in similar designs. A design plan can be replayed by executing the corresponding macro rules. A problem solving session is compiled into several micro rules with different abstractions, which can be reused independently to increase design efficiency in later design sessions. The design strategy of ARGO is to perform top-down design by applying the most specific micro rule first (i.e., analogical reasoning). If no micro rule is available, general design rules in the knowledge base are used to refine the current design. ARGO has a fixed design strategy and uses only previous design plans. However, its design plans can be reused entirely or in less detailed abstract forms.

- BOGART [Mostow 87] reuses design plans for designing VLSI circuits. It is built on top of the interactive circuit design system VEXED to capture human design decisions and to reduce the amount of required user interaction later. The user selects which module to refine and which decomposition rule to apply at each design cycle in BOGART. The design plan is represented as tree-like structure with decomposition rules and modules as nodes. Plans have to be retrieved by the user and can be reused for design iteration (revising an early implemented decision and replaying subsequent decisions) and design by analogy (adapting a model using derivational analogy). BOGART is really a plan replay mechanism, which can automate many of the repetitious aspects of design. It is mostly used for design iterations.

Two small prototypes have been experimented in conjunction with this project at Stanford University: RESTCOM (REdesign of STructural COMponent) and FIRST.

- RESTCOM [Rafiq 89 and Howard 89] is a Prolog-based research prototype for experimenting with similarity and matching in a small case base of reinforced concrete beams. Given a problem to solve, RESTCOM compares that problem to each previous problem in its case base, using six different methods to score the matches. The case base is restricted to single-span, singly-reinforced rectangular concrete beams.

- FIRST [Daube 89 and Howard 89], implemented in LISP on top of the BB1 system [Hayes-Roth 85], redesigns structural beams by transformational analogy using a case memory of solution plans. FIRST starts by analyzing an existing design, using general knowledge about elementary physics. If design constraints are unsatisfied, FIRST searches for a similar problem situation in its case memory and retrieves the solution plan associated to that situation. FIRST then performs an analog transfer of the plan into the new problem situation and applies the plan to its current design. By working on solution plans, rather than solutions, FIRST's case memory captures those decisions for which the underlying rationale, often based on empirical evidence, experience and context, cannot be derived from first principles.

The ideas from case-based reasoning in the artificial intelligence community combined with the two initial experiments have led us to the model presented in this thesis for combining case-dependent reasoning and case-independent reasoning in an integrated, cooperative design system.

# Chapter 3
# An Integrated Model of Design

This chapter examines the nature of integrating case-dependent and case-independent reasoning paradigms and what it takes to build such an integrated design system. The desire for an integrated reasoning paradigm inspired this research. A useful design model gradually emerged. We started with a design process model and expanded it with additions for design recording, reuse, and, finally, complete integration.

## 3.1. Integrated Reasoning Paradigm

In general, rule-based reasoning depends on heuristic and causal rules and, case-based reasoning relies on memory. We believe that the integration of case-based and rule-based reasoning can give knowledge-based design systems great power and flexibility. Figure 3-1 shows the different design approaches available in such an integrated reasoning paradigm. The first four approaches listed below are the problem solving methods classified by Carbonell in [Carbonell 85].

1. **Applying heuristic rules to search the design space for solutions and plans.** Since design can be viewed as a search for a solution or sequence of design actions in a large space of possibilities [Howe 86], it is possible to find a design solution or plan by searching heuristically through a finite space. This approach is performed by the rule-based reasoning part of the integrated system and is the basic design control strategy.

2. **Instantiating a specific design plan.** An experienced designer can perform a routine design by following a certain procedure without going through all the design decisions that might provide alternative paths to a solution. If this kind of knowledge can be generalized from cases in the form of design plans, the system may retrieve and reuse them as the procedures for solving design problems.

3. **Applying a generalized design.** A parameterized design can be instantiated directly when its generalized specification is met. For some kinds of design problems, if the configuration of the design can be predetermined, the fixed components of the design can be decided by parameters. Therefore, the system may design from such generic structures if they can be produced by memory generalization or directly provided by domain experts.

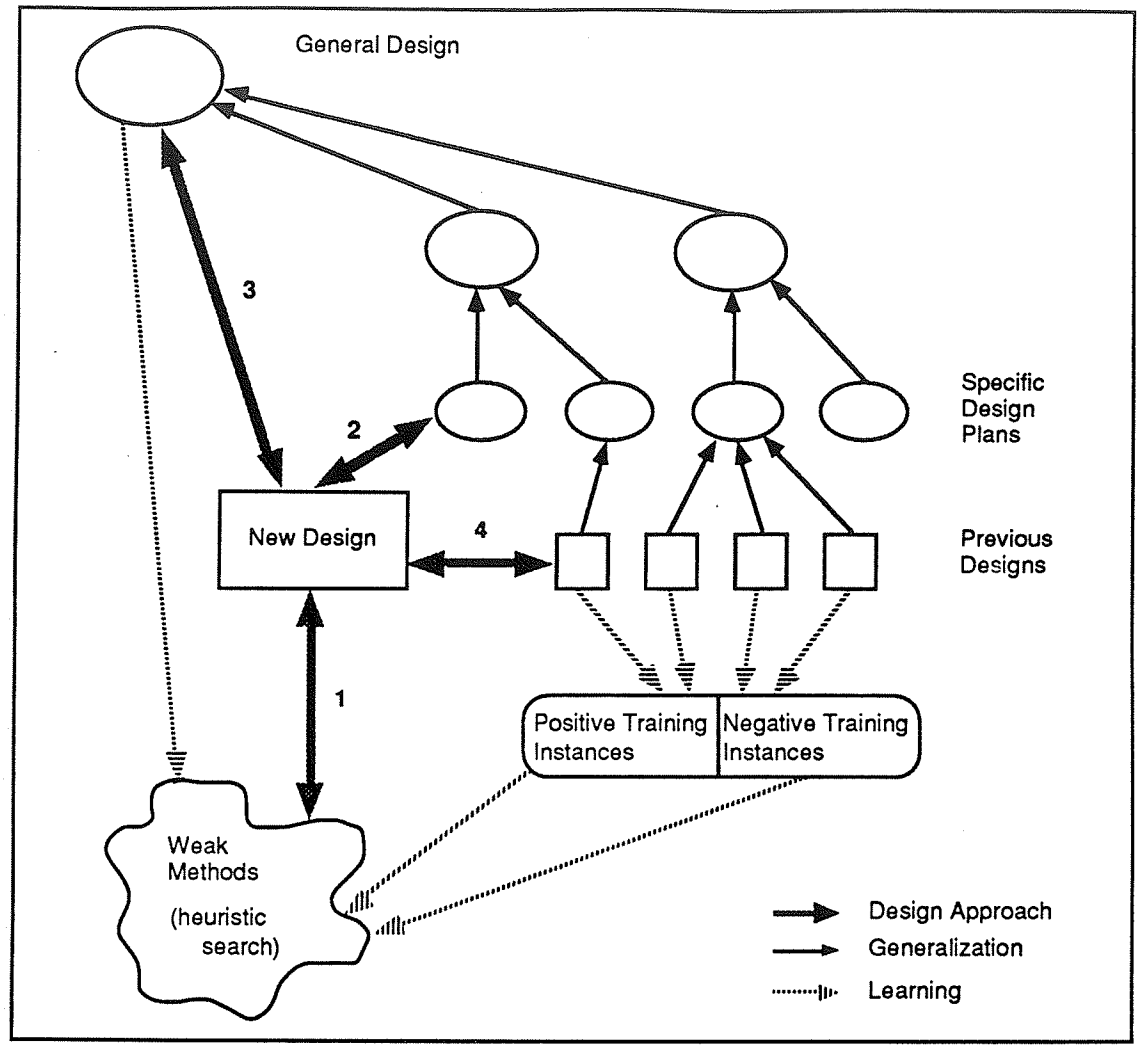



**Figure 3-1:** Reasoning Paradigms in an Integrated Design System (after [Carbonell 85])

4. **Applying an analogical transformation to adapt the solution or plan of a similar previous design.** The system should be able to utilize a variety of case-dependent knowledge in the memory to assist the design of structures using case-based reasoning. For instance, it is easier to design a structural component with some similar past design experiences than solely with codes and general procedures presented in handbooks or texts. One can get important information (e.g., initial assumptions, crucial design variables and constraints, or even design decisions) from similar past designs to guide the current design.

5. **Combining multiple approaches.** For example, a design problem can be divided into subproblems by applying heuristic rules. Then, each subproblem may be solved by instantiating a specific design plan, a generalized design, etc. This can be achieved by a global design monitoring and control mechanism of

the integrated design system, which dynamically selects an applicable design approach.

## 3.2. Modeling the Design Process

A knowledge-based design system needs an underlying design model to represent the design process. Designing a structure is a complex task. Usually, the problem space is too big to search. Decomposition is a common strategy used by engineers to divide the search space. Domain specific knowledge and general problem-solving skills are required to control the decomposition and to search effectively for the reduced design space. However, the complexity of individual subproblem and the interaction between subdesigns can still make the design an iterative process. The generated design has to be tested against all the design constraints and relevant parts of the design must be modified to overcome any constraint violation. This process may have to be repeated several times to produce an acceptable design.

Therefore, designs are divided into subtasks in the design system, and the knowledge for solving individual subtasks is stored in a knowledge base. The problem solver uses that knowledge to develop design solutions through a generate-test-modify paradigm. Top-down decomposition plans are also stored in the knowledge base. A plan represents a design strategy that consists of a sequence of design steps. Each step in a plan is called a goal; goals guide the problem solver toward the desired solution. Different levels of abstraction of plans can help decompose and organize design knowledge. A plan can specify a sequence of goals that produce a subdesign. A plan can be a top level design strategy that points to other plans.

The problem solver checks design constraints throughout the design as appropriate. Whenever a constraint violation is found, the problem solver has to overcome the design failure by redesign (i.e., modifying the partial design). Redesign in the system is based on dependency-directed backtracking with knowledge-based advice. The problem solver's truth maintenance system (TMS) provides dependency links for a design failure. The dependency network is built during design. The design variables involved in the current design failure can be traced through the links. The problem solver analyzes the dependency information and the violated constraints to suggest what part of the partial design to modify and how to fix it. The TMS updates the current design according to the modification. Design values that are no longer valid (as well as values derived from those values) are removed, and the problem solver proceeds from there to its generate-test-modify cycle.

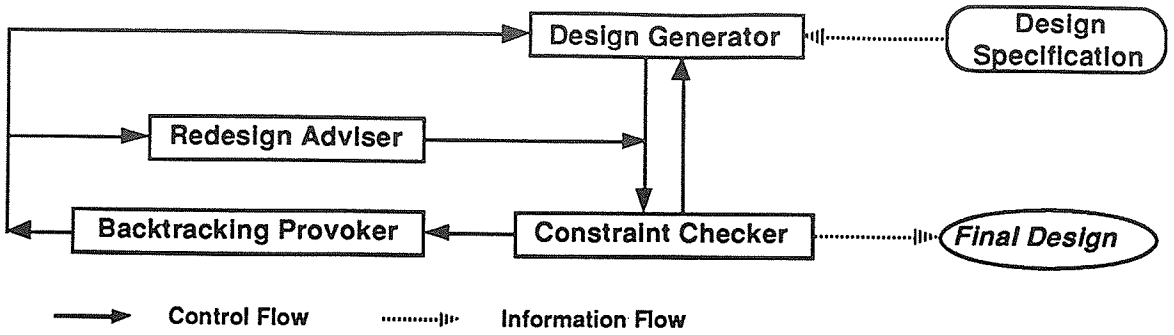



**Figure 3-2:** Control Flow in the Case-Independent Reasoner

Based on the design model, our case-independent reasoner has four modules (see Figure 3-2): a design generator, a constraint checker, a backtracking provoker and a redesign adviser. They are comprised of design rules corresponding to the general design heuristic knowledge. The design generator assigns values for design variables based on design heuristics, analysis, calculation, etc; the constraint checker checks for constraint violations; the backtracking provoker controls the extent of design modifications and the redesign adviser gives redesign recommendations. The case-independent reasoner also contains some deep knowledge about domain-specific aspects such as design codes, constraints and analysis procedures.

## 3.3. Saving a Design

One of the motivations for building an integrated case-dependent and case-independent system arises from the desire to improve the system's abilities on the basis of its own experience. Therefore, the next step is to build a recording mechanism (the case recorder) to store case-dependent knowledge to the system's memory (the case memory) after a design session is done. Figure 3-3 shows the system with the additional case recorder and memory.

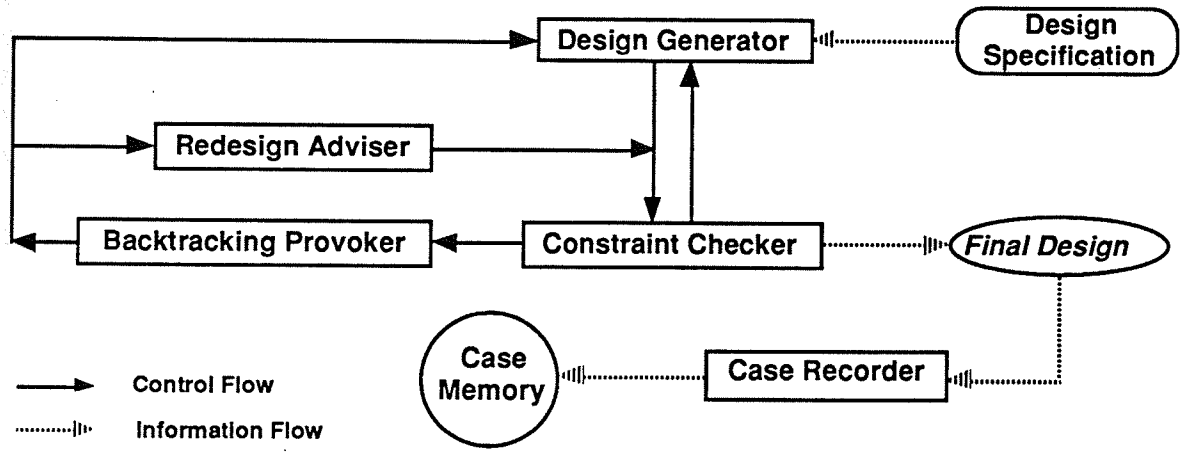



**Figure 3-3:** The Case-Independent Reasoner, Case Recorder and Case Memory

A lot of information is generated during design. Because of the limitation of memory space and efficiency of search, only a subset of the design information should be recorded. The system should save only the most potentially reusable case-dependent knowledge and index it with its significant features in the memory. The most important elements of case-dependent knowledge identified in this research are:

- **Design Solutions**—The final design solution of a design problem is certainly the most apparent information that can be reused in the future. However, intermediate solutions and partial solutions can also be useful. Solutions need to be indexed by the goals that they achieved and by the physical components that they belong to.

- **Design Justifications**—The calculations of previous design variables and the dependences of their values can be used to predict design outcome without going through all the design steps that lead to previous result. Design justifications are associated with each design case.

- **Design Plans**—Design plans are the strategies used to solve a design problem. Plans are the control knowledge of design cases, which can be used to schedule design steps and to select a design method for each design step. Plans for the global control of design processes and the control of backtracking processes need to be separated in the memory. Plans can be indexed in a number of ways (i.e., by the goals that they accomplished, the design values that they produced and the constraints that they satisfied and unsatisfied).

- **Design Goals**—Specific knowledge about how to achieve a particular design step is contained in a design goal. Design goals are included in plans to form complete case-dependent control knowledge of a case.

- **Design Failures**—Previously unsuccessful design alternatives can be used to avoid similar design failures in the future. Design failures should be linked to their recovery plans.

- **Design Constraints**—Constraints are used to evaluate designs. Their statuses throughout the design indicate the progress of a problem-solving session. They are very important information for understanding the history of a design case.

In order to support the capture of different types of case-dependent knowledge, the design process model discussed in the previous section not only needs to define the computational process of design, but also should provide the basis for representing different types of useful case-dependent knowledge. Then the case recorder can process the design case after a design session is finished to capture the associated case-dependent knowledge and store it in the case memory knowledge base. The case memory knowledge

base should be organized to provide easy access to all kinds of case-dependent knowledge. For example, the case memory knowledge base can be divided into a memory of plans, a memory of solutions and a memory of failures, according to the contents that the case recorder stores.

Furthermore, if generalization of the case memory is desired, we could include a generalizer to generalize the case memory. The design cases or their indices can be generalized to have uninstantiated variables, and the case memory can be further divided into a case-specific and a generalized memory.

## 3.4. Using CBR in Design

With the case recorder and memory, we can now store the system's own design experience. However, the design system needs a case-based reasoner to reuse the saved case-dependent knowledge. The function of the case-based reasoner is to assess previous design cases and transfer knowledge from them to the current design task. Therefore, a memory prober is added to the case-based reasoner to retrieve potentially applicable cases from the case memory. These cases supply the knowledge used in the other case-dependent modules. Then, an analogy transformer is used to transfer the design solution or solution strategy of a retrieved design to the new design and modify it to satisfy the requirements of the new design.

We discussed the different aspects of case-dependent knowledge in Section 1.2 and 3.3. Ideally, the case-dependent reasoner is capable of performing case-base reasoning using all available case-dependent knowledge in different phases of the design process. For example, a failure anticipator can be included to check for the potential for failures and avoid them in the new design. Past design failures and the reasons that they occurred should be stored in the case memory so that the failures can be predicted if the same causal conditions appear again in a new design. If the system is capable of memory generalization, we can also add an instantiater to instantiate a generalized design plan or solution retrieved by the memory prober. The generalized design solutions can be idealized configurations of components or detailed, parameterized representations of an individual component.

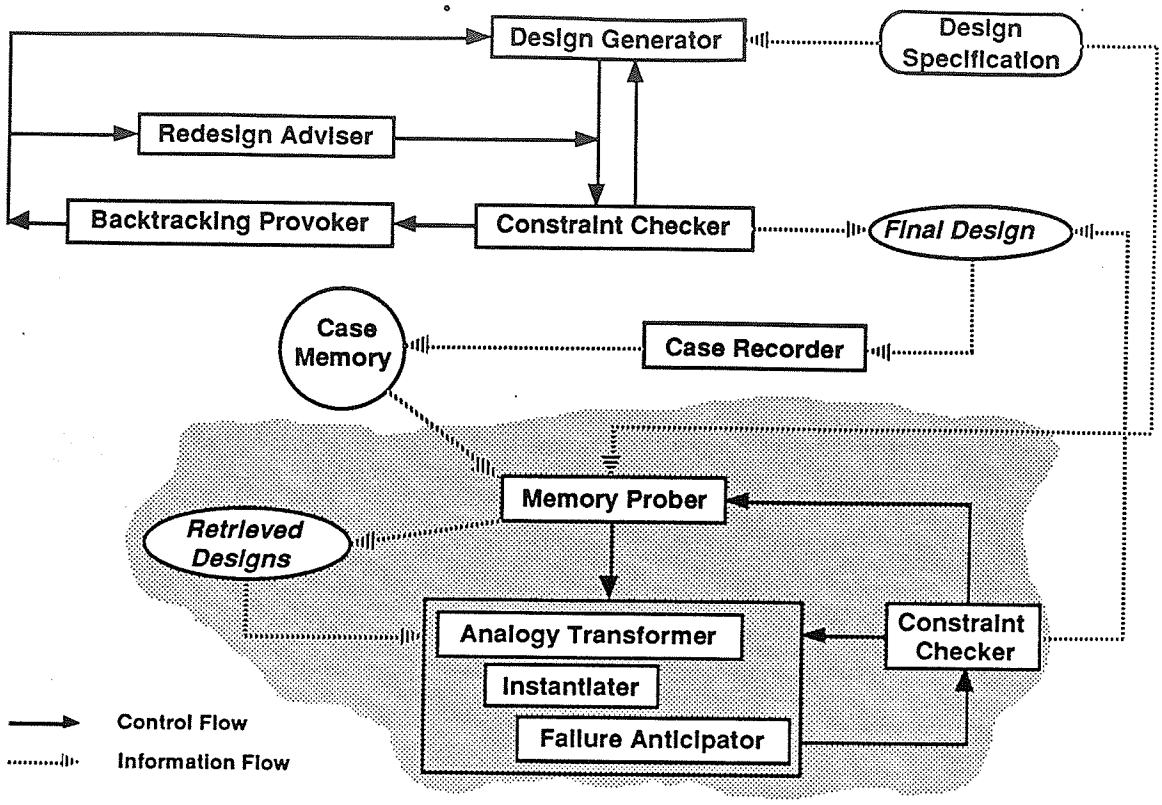



**Figure 3-4:** The System with Case-Based Reasoner

The expanded system is show in Figure 3-4. Up to this point, our design system can perform either a heuristic-based design or a case-based design. However, the two reasoning paradigms are independent. We can only select one reasoning method at the beginning of each design session and stick to that ideal to the end.

## 3.5. Integrating the Two Design Approaches

Case-based design alone is not usually the best way to tackle a design problem. The recognition of similar cases and the availability of representative cases can influence the effectiveness of case-dependent reasoning. It is our belief that the combination of case-dependent and heuristic knowledge gives the most power and flexibility to the human designer. Therefore, the last step of building our system is to integrate case-based reasoning into the basic case-independent reasoning loop (see Figure 3-5).

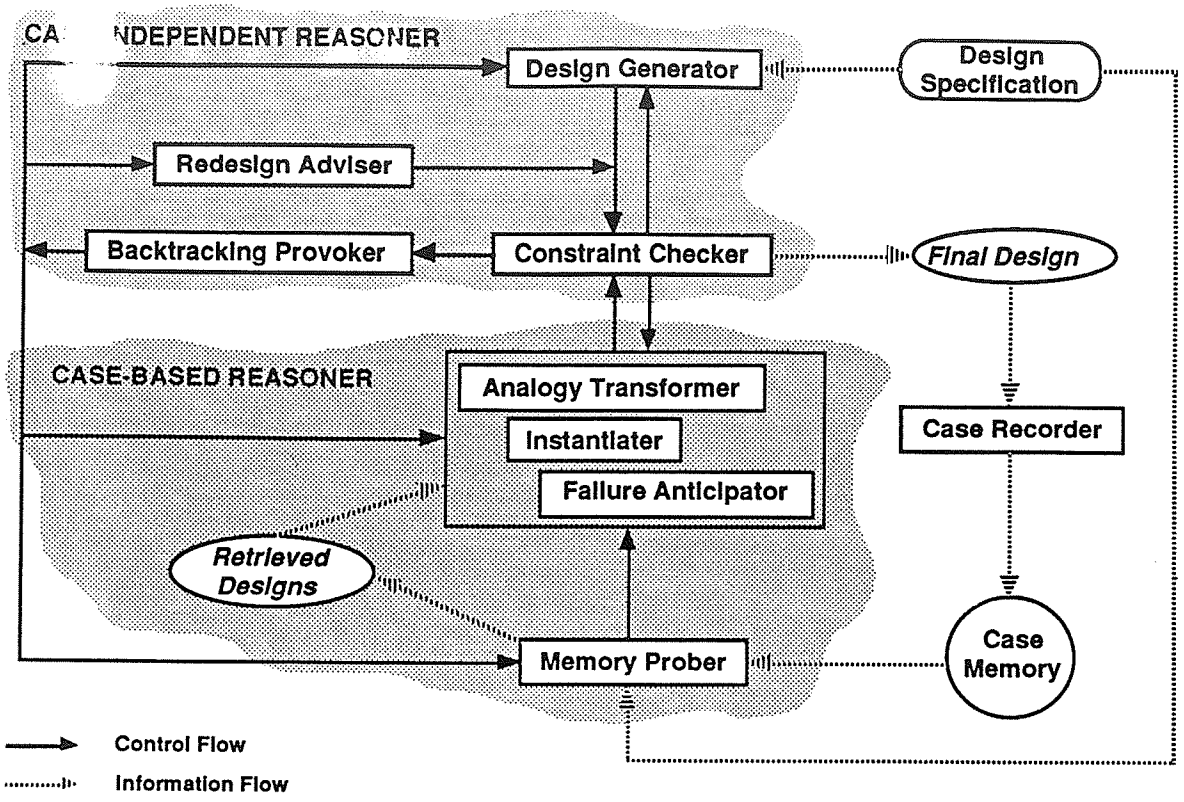



**Figure 3-5:** The Final Integrated Knowledge-Based Design System

In the new arrangement, a design session can start from the design generator or memory prober as usual. However, the new arrows between the case-independent reasoner and the case-based reasoner allow both of them to get their hands on a problem. For instance, the design generator can be used first to generate a partial design, then the design is checked by the constraint checker. If a constraint violation is found, the backtracking provoker launches the design modification process. Instead of going to the redesign adviser, the system can use the memory prober to find a past design that had encountered the same design failure. The analogy transformer transfers the previous fix to the new design and the constraint checker confirms that the current design satisfies the previously violated constraint. Now, the system can go back to the design generator to develop more design solutions for the other parts of the design, or it can use the analogy transformer to transfer more solution from the retrieved design.

This kind of integration is truly flexible and powerful, furnishing the system with all the five design approaches discussed in Section 3.1. Combined with a good control mechanism, the integrated reasoning paradigm can be used opportunistically to take full advantage of available design knowledge including both case-independent and case-dependent.

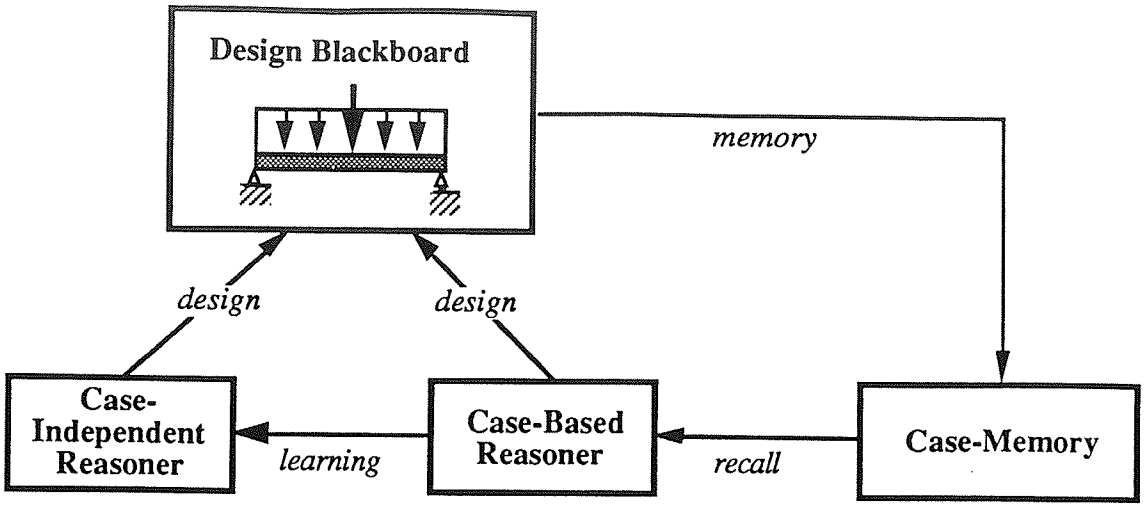



**Figure 3-6:** Overview of Integrated Knowledge-Based Design System

Now the question is; How can we achieve this kind of integration? The answer is to use a blackboard model, which supports collaborative problem solving. The conceptual model of the integrated case-dependent and case-independent design system is shown in Figure 3-6. Design processes are divided into subtasks. Knowledge that specifies how to carry out a subtask, give a redesign suggestion, or decompose a design goal can be either case-based or heuristic-based. With a case memory, the integrated system can perform each design subtask based on either case-dependent or case-independent knowledge. The blackboard architecture allows case-based reasoning to compete with case-independent reasoning throughout the design process, which makes the system an integrated opportunistic design model.

Furthermore, the model provides for the extraction of case-independent knowledge from case-dependent knowledge (a process otherwise known as learning). The long-term objective is to produce integrated design systems that interact with designers during design tasks, functioning both as intelligent design assistants and as knowledge acquisition systems that record the designers' steps and rationale. In this way, the design systems become true apprentices to the experienced designers, progressively learning to solve more and more difficult design problems.

# Chapter 4
# The DDIS Environment

The integrated design model described in the last chapter is implemented in the computer program DDIS[1]. DDIS is an integrated knowledge-based design system incorporating both rule-based and case-based problem-solving techniques to combine case-independent and case-dependent knowledge. The overall organization of DDIS is based on a blackboard model very similar to the BB1 architecture [Hayes-Roth 84 and 85] and is implemented in KEE. In the blackboard model, the knowledge needed to solve a problem is partitioned into independent knowledge sources that are grouped into several knowledge modules in the knowledge base. The knowledge sources modify only a global knowledge structure (the blackboard) and respond opportunistically to the changes on the blackboard. Using a blackboard architecture, DDIS can apply both case-dependent and case-independent knowledge to perform collaborative and opportunistic design. This chapter first presents the overall functionality and architecture of DDIS Then, the details of its blackboard are described, the strategies for execution control are elaborated, and the elements composing its case memory and knowledge base are reported. A summary of the implementation is given at the end.

## 4.1. Overview

The behavior of DDIS's blackboard architecture draws on the following analogy. A design problem is stated on the blackboard in a room, and a team of designers with different talents surrounds it (see Figure 4-1). None of the designers can solve the design alone, but each is capable of solving a piece of the problem. Each designer knows exactly what he can do and when he can do it. The designers look at the blackboard to see if their solutions can fit into the current problems. Those who can contribute to the current design situation raise their hands. It is possible that several options may exist for one designer under certain conditions. In this case, the designers with more than one option need to identify all the possible design actions that they can provide. The team manager chooses one designer (and one action if the designer has more than one alternative) based on some criteria, which are presented as design strategies on the blackboard. The chosen designer goes up to the blackboard and makes changes to the evolving solution. Note that, at one time, no more than one designer can work on the blackboard and only one design option

---

1 DDIS stands for Design-Dependent and Design-Independent System. "Design-dependent" and "design-independent" are the terms used for "case-dependent" and "case-independent" in the early stage of this research.

can be used. After the designer is done, a blackboard engineer makes necessary updates to the other parts of the design that are related to the changes and to the design strategies on the blackboard to maintain design consistency and focus.

All the designers can now respond to the new updates and raise their hands if they know what to do next. The manager once again selects one person to work on the design, and the blackboard engineer updates the blackboard situation accordingly. Consequently, the changing blackboard state causes other designers to work on the design and other pieces of the design solution to form incrementally.

It is important to point out that the design team consists of two major design groups: a case-independent group and a case-based group (see Figure 4-1). The designers in the case-independent group specialize in analysis and heuristic design methods (e.g., equation calculation, finite element analysis, optimization, rule-based design, etc.), and the designers in the case-based group specialize in case-dependent methods (e.g., solution adaptation, design plan reuse, case-based failure anticipation, etc.). Of course, the case-based group has a collection of previous design cases and associated case-dependent knowledge. The designers do not directly talk with each other. They propose their design actions to the team manger when proper situations appear. The manager decides whose proposal is the best. At a given situation, there may be proposals from both the case-independent group and the case-based group. Therefore, the case-independent group competes with the case-based group for the design task. At the same time, the designers within each group have to compete with each other as well.

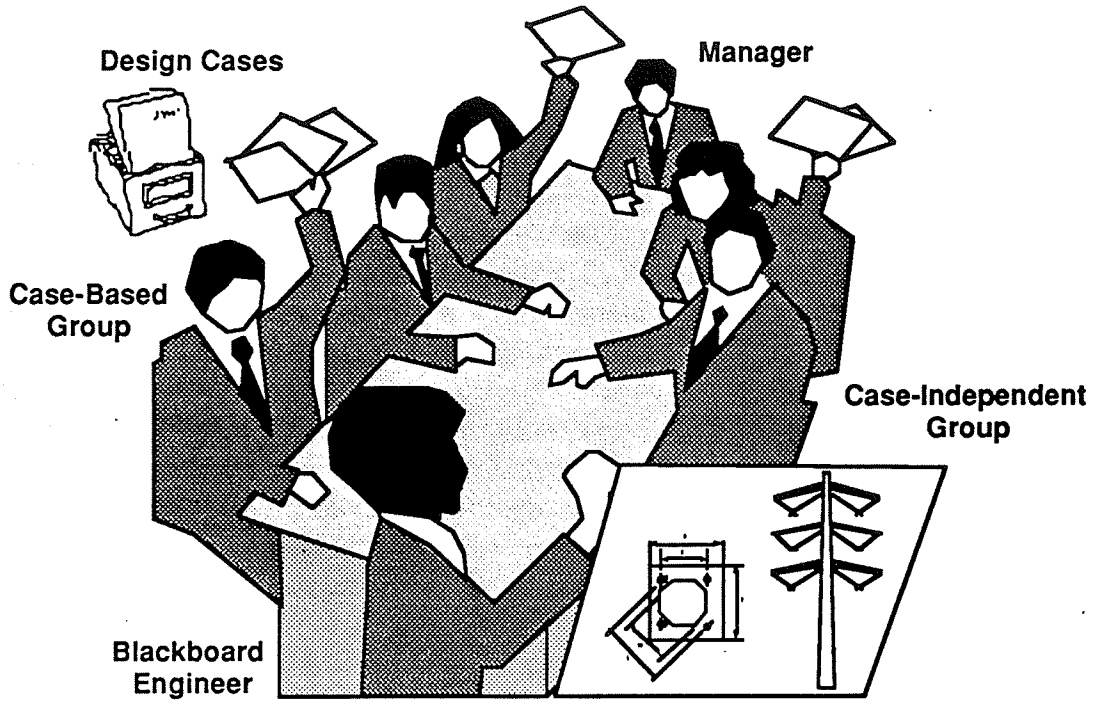



**Figure 4-1**: Blackboard Metaphor

The team manager dynamically selects a sequence of actions from different designers to produce a design according to the design strategies that are in use. For example, a design strategy can be "design component B1 before component A2" or "favor case-based design group over case-independent design group." The manager's choice of a designer is based on the latest solution state and design strategies and on the existence of designers capable of improving the current design. At each design step, either a case-independent or a case-based designer can be called in. The design team performs collaborative and opportunistic design.

### 4.1.1. System Architecture

The preceding analogy illustrates the integrated nature of DDIS's problem-solving behavior. Now, the blackboard metaphor is mapped into the system architecture of DDIS (see Figure 4-2). The design team is the knowledge base and the two design groups are called the case-independent reasoner and case-based reasoner knowledge modules in DDIS. The collection of case-dependent knowledge that the case-based designers use to do case-based reasoning is called the case memory knowledge base in DDIS. The subsidiary components in each of the knowledge modules were discussed briefly in Chapter 3 when we built up the integrated design model and will be further described later. Note that DDIS does not have the capability of learning new rules and generalizing cases—the shaded area in Figure 4-2 is not implemented in the current version of DDIS.

Each "designer" in the design team is called a knowledge source (KS), and each "design option" that is applicable under the given condition is called a knowledge source activation record (KSAR). A KS represents a kind of problem-solving action in DDIS. The KS knows when it becomes applicable and what actions it proposes to take. The condition under which a KS can be executed is specified by a blackboard state with a set of variables to represent different triggering contexts, and the action of the KS is also specified by the same set of variables to represent alternative actions under different contexts. Therefore, multiple KSARs can be generated, one for each context, when more than one triggering context exists.

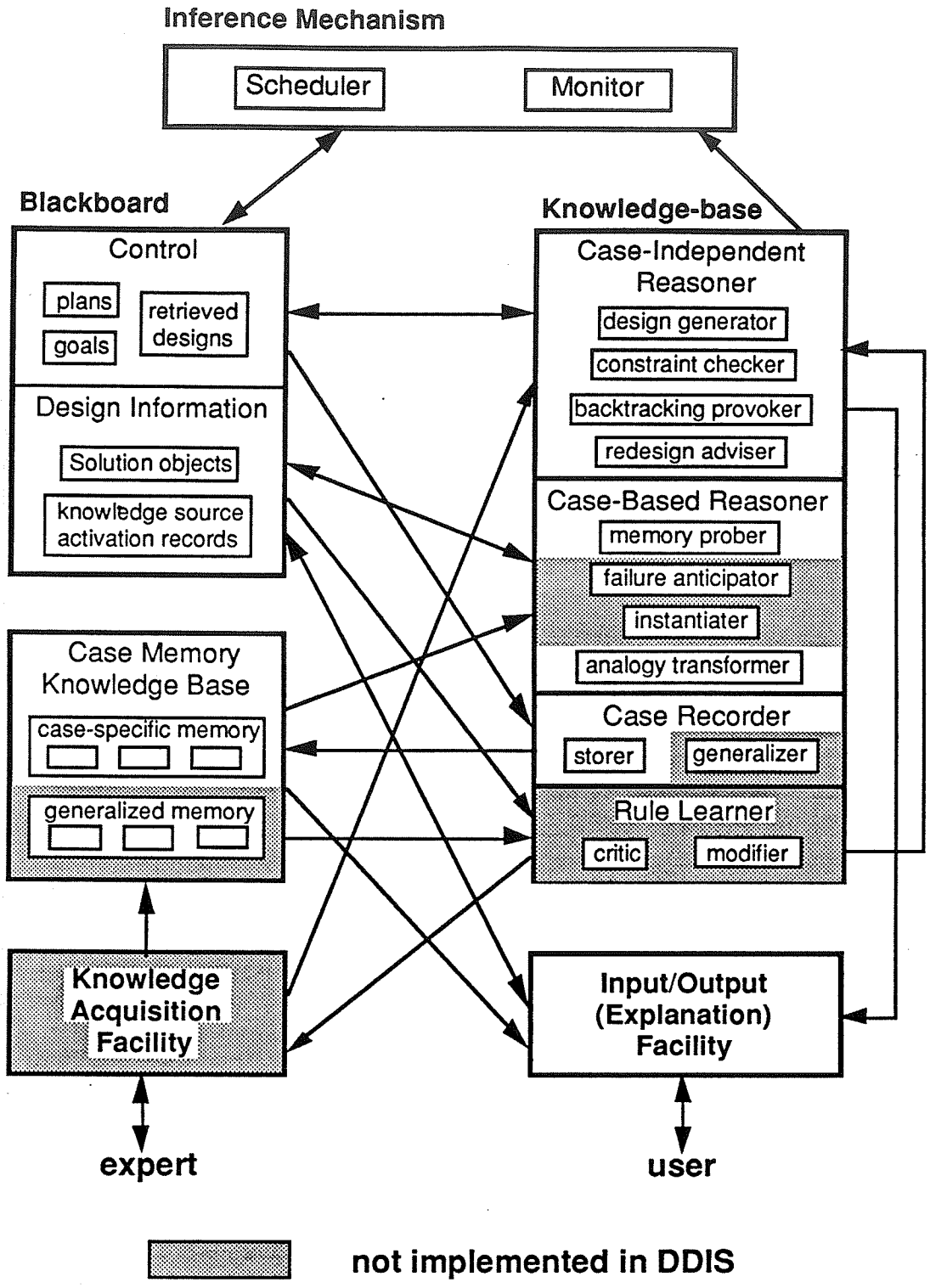



**Figure 4-2:** Data and Knowledge Flow in DDIS

The "team manager" is the scheduler in DDIS. It looks over the design and selects one KSAR for execution among all the alternatives at each design step. The scheduler's decision is based on how well the action of a certain KSAR matches the intent of the design strategies that are in use. No preset order of execution is defined in DDIS. The design strategies are represented as plans and goals in DDIS. A goal is a design consideration that DDIS uses to rate KSARs. A plan is a sequence of design goals that represents a design strategy. Each goal in a plan is a strongly desired design objective that leads to what the plan intends to achieve. The rating of a KSAR against all plans on the blackboard is a numeric value ranging from zero to one hundred, which represents "no match" to "perfect match." At every cycle, the scheduler selects the highest rated KSAR to execute (or the user can override DDIS's scheduler by selecting a different KSAR). The design solution is built incrementally and opportunistically. The blackboard engineer is the inference monitor that keeps track of the status of plans and goals on the blackboard and performs the truth maintenance task described in Section 3.2.

With this basic understanding of DDIS, now let us look at how the above elements are implemented in DDIS.

### 4.1.2. Knowledge Representation

Knowledge representation in DDIS is based on an object-oriented (or frame-based) approach. Due to the hierarchical nature of the physical elements, the frame-based data structure with value inheritance is convenient to represent design information at both component and system levels. The abstract data type and class-instance inheritance supported by an object-oriented paradigm captures the interrelatedness of design information and stores different levels of generalization of design knowledge.

Almost all knowledge in DDIS is stored as objects at the lowest level (e.g., physical objects, design actions, plans, constraints, and design cases). Figure 4-3 shows some objects contained in the knowledge base of DDIS. Each object type defines special values and procedures for a certain class of design objects or knowledge. The design objects, variables, constraints, methods, plans, and heuristics that make up the knowledge base are instances of those predefined object types.

In addition to the object-oriented model of design objects and knowledge, DDIS uses Garrett's object-oriented design standard model [Garrett 89]. All the logical expressions and variables in the design standard are represented as objects that are called data items. Each self-contained data item has a value, type information and methods for determining its value. The data items and their interrelationships can be organized into a hierarchy of object classes. The hierarchy of data items used in DDIS is shown in Figure 4-4. The instances created from the leaf classes are used as the foundation for representing knowledge in DDIS. Data items provide mappings between basic data items in the standard

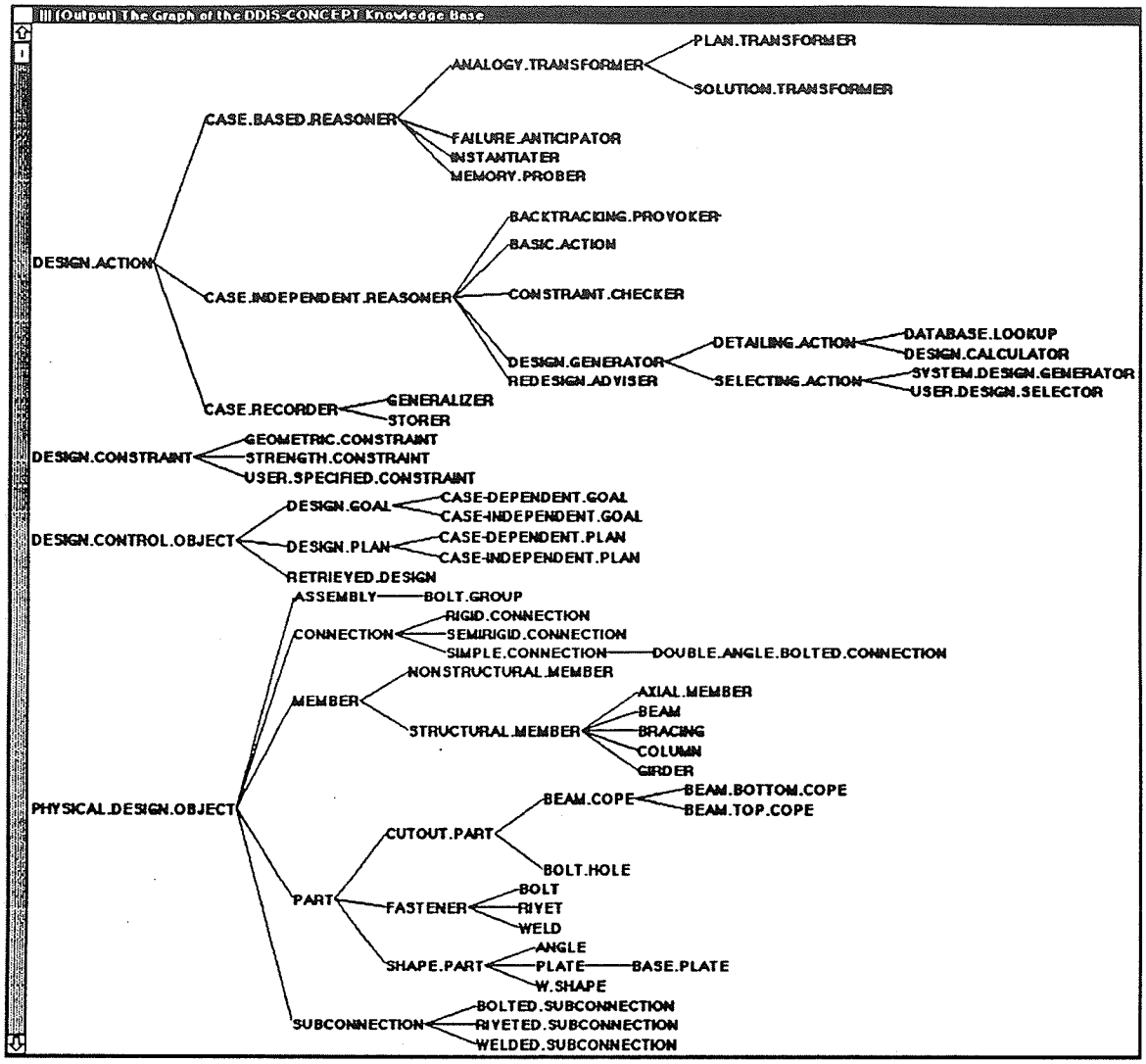



**Figure 4-3:** A Sample of the DDIS Object Hierarchy

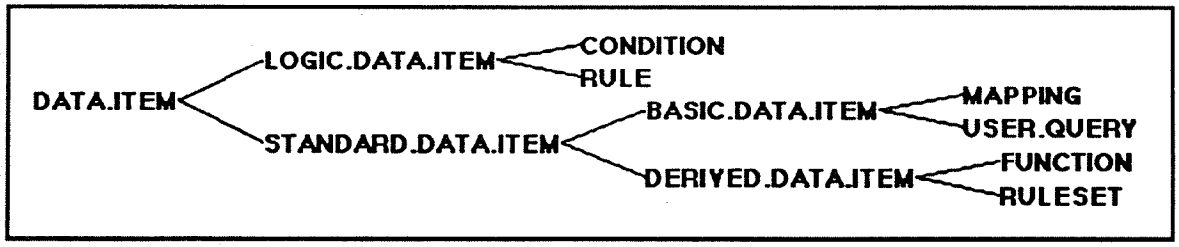



**Figure 4-4:** Hierarchy of Data Items in DDIS

model and design object attributes in the design model. An important additional function of data items in DDIS is to establish justifications of variable dependance during design, which are used by the truth maintenance system of DDIS to justify redesigns and maintain design consistency. When the value of a data item is accessed, its self-contained method of evaluation is used to determine its value. At the same time, the attached procedure creates a justification for that value. The following is a brief description of the leaf data items:

- **Mapping** is a subclass of basic data item. A mapping data item has a description of where to find its value. Mapping data items are used to link variables in the design standards to the design objects. Therefore, an attribute of a design object is referred to as the value source. An example of the mapping data item is given below. The value of a mapping form should be a wff form (well-formed formula in KEE's TellAndAsk language) that indicates how to retrieve the value for the data item. Note that the variable ?VALUE must be used to indicate the final value returned form the wff form.

```
A  (cross section area of a beam-column)
     MAPPING.FORM:    (THE AREA OF (THE DESIGNATION OF
                          BEAM-COLUMN.1) IS ?VALUE)
     VALUE:       17
     VALUE?:      a method for locating the value of
                  mapping objects based on the provided
                  mapping forms
```

- **User.Query** is another subclass of basic data item. A user query object is a data item that needs to be supplied by the user. It contains the descriptions of how to ask the user for its value and where to store the value. The following is an example of the User.Query object.

```
P  (factored axial loading)
     QUERY.LABEL:     "P (factored axial loading in kips)"
     STORAGE.FORM:    (THE MAGNITUDE OF
                         FACTORED.AXIAL.COMPRESSION IS
                            ?VALUE)
     VALUE:       UNKNOWN
     VALUE?:      a method for retrieving the value of
                  query objects
```

- **Function** is a subclass of derived data item. A function data item has an expression of how to compute its value. The expression is a non-conditional function and may involve other data item objects in the problem. Lisp functions can also be used in the expression. For example, GET.VALUE is a KEE function that retrieves the specified slot value from the given object. It can be used in the expression to access slot values (see the first example). When the value of a function data item is requested, the attached VALUE? method evaluates the expression and returns a value for the inquiry.

```
PE  (Euler buckling load)[1]
      SYMBOLIC.EXPR:  (/ (* (EXPT PI 2) E A)
                         (EXPT (/ (* K (GET.VALUE
                                        'BEAM.COLUMN.1 LENGTH))
                                  RX) 2))
      VALUE:      UNKNOWN
      VALUE?:     a method for computing the value of function

P.BOLT.MAX.RULE.1.CONCLUSION
      SYMBOLIC.EXPR:  (+ (/ P 4) (/ M BOLT.DIST (COSD 45) 2))
      VALUE:      UNKNOWN
      VALUE?:     a method for computing the value of functions
```

- **Condition** is a subclass of logic data item. A condition data item has a symbolic expression that describes a logical relationship between data items. Its value is either true or false. The following example is the condition of rule P.BOLT.MAX.RULE.1 (see the example of rule data item).

```
P.BOLT.MAX.RULE.1.CONDITION
      SYMBOLIC.EXPR:  (= BOLT.PER.QUADRANT 1)
      VALUE:      T
      VALUE?:     a method for determining the value of
                  conditions
```

- **Rule** is a subclass of logic data item. A rule data item has two parts: a condition and a conclusion. The condition part is a collection of condition data items and the conclusion part is a function data item that represents the value of the rule if the conditions are met. The example below is the first rule contained in ruleset P.BOLT.MAX (see the example of ruleset data item).

```
P.BOLT.MAX.RULE.1
      CONDITION:    #[UNIT: P.BOLT.MAX.RULE.1.CONDITION][2]
      CONCLUSION:   #[UNIT: P.BOLT.MAX.RULE.1.CONCLUSION][3]
      SYMBOLIC.EXPR:  (IF (= BOLTS.PER.QUADRANT 1)
                          THEN (+ (/ P 4)
                                  (/ M BOLT.DIST (COSD 45) 2)))
      VALUE:      UNKNOWN
      VALUE?:     a method for determining the value of rules
```

---

1 This function object represents the equation $P_c = \frac{\pi^2 EA}{(KL/r_x)^2}$ in Figure 4-12. The definitions of the ingredient variables are given in Section 5.1.

2 a condition object (see the example of condition data item) created from the IF part of the symbolic expression

3 a function object (see the second example of function data item) created from the THEN part of the symbolic expression

- **Ruleset** is the second subclass of derived data item. A ruleset data item collects a set of rule data items to represent its conditional value. Each rule represents a possible value of the data item under certain conditions. Only one value is feasible under a given condition. An examples of the ruleset object are given below.

```
P.BOLT.MAX  (max.  anchor  bolt  load)
      RULES:      #[UNIT: P.BOLT.MAX.RULE.1][1]
                  #[UNIT: P.BOLT.MAX.RULE.2]
      VALUE:      UNKNOWN
      VALUE?:     a method for determining the value of
                  rulesets
```

## 4.2. Blackboard

The blackboard is a centralized data structure responsible for the storage of dynamic design information generated during the design process. All information is stored on the blackboard as DDIS objects. The blackboard of DDIS is divided into a control blackboard and a design information blackboard.

- **The Control Blackboard** collects three types of control objects: plans, goals and retrieved designs, reflecting the different levels of abstraction of DDIS's control knowledge. The information on this blackboard contains the control decisions that the system uses to schedule the activities.

- **The Design Information Blackboard** consists of solution objects and action objects as well as information generated by various design knowledge sources, including evolving solutions, design history, etc.

Descriptions of the various types of objects follow.

### 4.2.1. Control Objects

The blackboard control objects are used to define control knowledge in DDIS. When they are posted on the control blackboard, DDIS uses the information contained in them to rate and schedule KSARs. The DESIGN.CONTROL.OBJECT class in Figure 4-3 shows the control object hierarchy of DDIS. The following are explanations of their properties.

---

1 a rule object defined in the example of rule data item

#### 4.2.1.1. PLANS

A plan is a problem solving strategy in DDIS. It specifies a sequence of design goals to be followed and the intention of the plan as well as several other attributes. Plan objects can be classified into case-independent plans and case-dependent plans (design plans associated with a particular case in the case memory knowledge base). Case-dependent plans are treated exactly like case-independent plans except that they have an additional attribute to indicate their origin. Below are required attributes of a plan.

- **GOAL.LIST**—A list of goal objects to be activated sequentially. Each element of the list can be a single goal object or a list of goals to be activated at the same time. For example, BASE.PLATE.DESIGN.PLAN.1 given in Figure 4-5 is a heuristic plan for steel anchor base plate design. To represent the four steps of the plan, its goal list contains four goals: (DESIGN.ANCHOR.BOLTS (CHECK.BOLT.CONSTRAINTS DESIGN.BASE.PLATE) CHECK. PLATE.CONSTRAINTS). It is a valid goal list as long as all the elements in the list are the names of goal objects defined in DDIS. The first objective of this plan is to design the anchor bolts. After the bolts are designed, DDIS can check bolt constraints or design the base plate. The two goals are made active at the same time and can be pursued in parallel. Therefore, DDIS doesn't have to finish checking all the bolt constraints before it can perform base plate design. For instance, DDIS can select several actions for constraint checking first and follow with a few actions for base plate design. Then, it can go back to check more bolt constraints. However, the final goal of the plan (i.e., CHECK.PLATE.CONSTRAINTS) should only be pursued after CHECK. BOLT.CONSTRAINTS and DESIGN.BASE.PLATE are accomplished.

- **INTENTION**—A LISP form that indicates when the plan needs to be deactivated. When it evaluates to true, either the intention of the plan is satisfied or it is no longer applicable. For instance, the intention of BASE.PLATE.DESIGN.PLAN.1 indicates what design attributes must have values before the plan can be terminated (see Figure 4-5). VALUE-DESIGNED-P, a predicate function, is true if all the specified object and attribute pairs have design values on the blackboard.

- **STATUS**—Either OPERATIVE or INOPERATIVE. If the status of a plan is inoperative, it is excluded from the problem solving process.

- **WEIGHT**—A number between 1 and 10 (inclusive), representing the importance of the plan. It is used in KSAR rating to account for different priorities of design plans. The default value is 5. It can be changed by the user when creating the plan object or by control KSs during design. A plan with a higher weight value is considered more important than a plan with a lower weight and is more influential in scheduling.

```
GOAL.LIST:
    (DESIGN.ANCHOR.BOLTS (CHECK.BOLT.CONSTRAINTS
            DESIGN.BASE.PLATE) CHECK.PLATE.CONSTRAINTS)

INTENTION:
    (VALUE-DESIGNED-P
            '(BASE.PLATE.1 MATERIAL)
            '(BASE.PLATE.1 THICKNESS)
            '(BASE.PLATE.1 DIAMETER)
            '(BOLT.GROUP.1 NUMBER.OF.BOLTS.PER.QUADRANT)
            '(BOLT.1 DIAMETER) '(BOLT.GROUP.1 BOLT.SPACING)
            '(BOLT.GROUP.1 BOLT.DISTANCE))

STATUS:     OPERATIVE

WEIGHT:     5

ACTIVE.GOAL:    NIL

REMAINING.GOAL.LIST:  NIL
```

**Figure 4-5:** BASE.PLATE.DESIGN.PLAN.1

- **ORIGINATED.CASE**—A design case object in the case memory knowledge base from which this plan was originally abstracted. This attribute is unique to case-based plans. It is used by the rating mechanism to trace the origin of the plan in order to modify the weight of the plan according to the similarity of the case. Since the given example in Figure 4-5 is a case-independent plan, it does not have this attribute.

- **ACTIVE.GOAL**—The goal (or goals) currently active in this plan. It is an internal attribute used by the blackboard maintenance mechanism. The default value is NIL, and it will be set to an appropriate value during blackboard updating.

- **REMAINING.GOAL.LIST**—A list of goals that remain to be activated in this plan. It is an internal attribute with a default value of NIL, and it will be set to an appropriate value by the blackboard maintenance mechanism.

#### 4.2.1.2. GOALS

A goal is the primary rating object for the blackboard system. It can be a part of a design plan or a stand-alone design consideration. DDIS uses all activated goals on the blackboard to rate KSARs.

Goal objects can be classified into case-independent goals and case-dependent goals (design goals associated to a particular case-dependent plan). Case-dependent goals have exactly the same structure as case-independent goals and are treated the same as case-independent goals on the control blackboard. Below are required attributes of a goal.

- **FUNCTION**—A LISP form that returns a value between 0 and 100. This is the rating function that evaluates how well a KSAR serves the goal. The higher the returned number is the better the KSAR is for achieving the goal. The variable $KSAR must be used in the function to refer to the KSAR under evaluation when information of the KSAR is needed. For example, `(* (GET.VALUE $KSAR 'PRIORITY) 0.9)` indicates that the rating of a KSAR is equal to the "priority" specified in the priority attribute of the KSAR times 0.9. A more comprehensive example is given in Figure 4-6. DESIGN.BASE.PLATE is one of the goals in BASE.PLATE. DESIGN.PLAN.1. Its function attribute is a conditional statement that specifies the different ratings of the KSARs that know how to pursue the goal.

- **INTENTION**—A LISP form that indicates when the goal needs to be deactivated. When it evaluates to true, either the intention of the goal is satisfied or it is no longer applicable. For instance, Figure 4-6 shows that the intention of DESIGN.BASE.PLATE is to generate design values for the material, thickness and diameter of the base plate.

- **STATUS**—Either OPERATIVE or INOPERATIVE. If the status of a goal is inoperative, it is excluded from the problem-solving process.

- **WEIGHT**—A number between 1 and 10 (inclusive), representing the importance of the goal. It is used in KSAR rating to account for different priorities of design goals. The default weight of a goal is 5. It can be changed by the user when creating the goal object or by control KSs during design. A goal with a higher weight value is considered more important than a goal with a lower weight and is more influential in scheduling.

- **INCLUDE.IN**—The plan object to which the goal belongs. If the value is NIL, the goal is not attached to any plan. Usually, a stand-alone goal is an important design decision that needs immediate attention. For example, stand-alone redesign goals fix constraint violations.

```
FUNCTION:
    (COND
        ((OR (EQUAL (UNIT.NAME (GET.VALUE $KSAR 'KS))
                    'REUSE.PREVIOUS.DESIGN)
             (EQUAL (UNIT.NAME (GET.VALUE $KSAR 'KS))
                    'REUSE.WHOLE.SOLUTION))
         (GET.VALUE $KSAR 'PRIORITY))
        ((MEMBER 'BASE.PLATE.1
                 (STRIP-LIST
                          (GET.VALUES $KSAR 'ACTION.CODE)))
         (* (GET.VALUE $KSAR 'PRIORITY) 0.9))
        ((EQUAL (UNIT.NAME (GET.VALUE $KSAR 'KS))
                'END.DESIGN) 20)
        (T 0))

INTENTION:
    (VALUE-DESIGNED-P
            '(BASE.PLATE.1 MATERIAL)
            '(BASE.PLATE.1 THICKNESS)
            '(BASE.PLATE.1 DIAMETER))

STATUS:     OPERATIVE

WEIGHT:     5

INCLUDED.IN:    BASE-PLATE.DESIGN.PLAN.1
```

**Figure 4-6:** The DESIGN.BASE.PLATE Goal

### 4.2.1.3. RETRIEVED DESIGNS

A retrieved design is the source of case-dependent reasoning in DDIS. It provides previous design solutions and plans that may be reused and specifies their similarity ratings, which can influence the blackboard's rating and scheduling decisions. Knowledge sources in the memory prober module are responsible for creating retrieved design objects and posting them on the blackboard. The following are required attributes of a retrieved design.

- **CASE.NAME**—The name of the design case object in the case memory knowledge base. The name provides the link to all previous design information associated with the case (a detailed description of the case memory is presented in Section 4.4).

- **SOLUTION.SIMILARITY**—A number between 1 and 100 (inclusive), representing the usefulness of the solution of the retrieved design in the current design. It is used to adjust ratings of KSARs that transfer solutions from the case. The larger the number, the better the case is for solution reuse.

- PLAN.SIMILARITY—A number between 1 and 100 (inclusive), representing the usefulness of the design plans of the retrieved design in the current design. It is used to adjust ratings of KSARs that transfer plans from the case. The larger the number, the better the case is for design plan reuse.

The criteria for judging the usefulness of previous solutions and case-dependent plans are different, and the methods for finding cases to use with solution transformer and plan transformer can be different. For example, a design case can be retrieved to reuse its associated design plans because of its highly successful control tactics. However, its solution may not be very useful according to the solution similarity ranking method. DDIS can retrieve the case for its plan transformer without confusing the solution transformer. Therefore, the retrieved design object has two different attributes, SOLUTION.SIMILARITY and PLAN.SIMILARITY, to take into account the need of two kinds of similarity ranking.

### 4.2.2. Solution Objects

Solution objects are used to hold design data generated during the design process. They include the following:

- **Physical Objects**, used for representing the design problem, are instantiated from design object classes in the knowledge base. Design values are stored in their attributes. The following is an example of a physical object.

```
BASE.PLATE.1
      DIAMETER:      21.75
      MATERIAL:      ASTM.A572.GR60
      THICKNESS:     0.75
```

- **Variable Objects**, used for design calculation and constraint checking, are instances of the data items discussed in Section 4.1.2. Their behaviors are inherited from their parent data item. The following example is a variable object instantiated from the function data item.

```
I.BOLT  (moment of inertia of bolt)
      SYMBOLIC.EXPR:  (* NUMBER.OF.BOLTS.PER.QUADRANT
                         (/ PI 64) (EXPT BOLT.DIA 4))
      VALUE:      UNKNOWN
      VALUE?:     a method for computing the value of
                  function objects
```

- **Constraint Objects**, including strength constraints, geometric constraints and user specified constraints, are design requirements to be satisfied. They are instances of the condition data item with special attributes and methods for constraint confirmation. The following is an example of constraint objects.

```
AISC2.4-2
      SYMBOLIC.EXPR:  (< (+ (/ P PCR) (/ (* CM M)
                           (* (- 1 (/ P PE)) MM))) 1)
      STATUS: SATISFIED
```

### 4.2.3. Action Objects

In DDIS, action knowledge is stored in knowledge source objects in the knowledge base. The action objects on the design information blackboard are KSARs (knowledge source activation records) instantiated from triggered knowledge sources.

#### 4.2.3.1. KNOWLEDGE SOURCES

A knowledge source (KS) is an action object that contains information about when it is applicable, how its variables are bound, and what action to perform. Knowledge sources can be divided into domain and control knowledge sources. Domain knowledge sources contain knowledge about performing a particular design task in the problem domain. They modify only the design information blackboard. Control knowledge sources, which modify the control blackboard, contain knowledge about planning the blackboard activities. Knowledge sources may also be classified according to the knowledge they apply—both domain and control knowledge sources can be further categorized as case-independent and case-dependent. Case-independent knowledge sources are based on generalized domain knowledge, and case-dependent knowledge sources contain cased-based knowledge to transfer information from previous design cases to new designs.

Despite the different KS classifications (defined for the sake of clarity, inheritance and implementation), all the knowledge sources are treated the same in DDIS. All kinds of knowledge sources compete at each design cycle based on how well they fit into the current design plans on the blackboard. The rest of this section describes the attributes of a knowledge source.

- **TRIGGER.CONDITION**—A LISP predicate that indicates the applicability of the KS. When it evaluates to true, the KS is triggered and eligible for instantiation. No context variables are allowed. For example, the trigger condition of REUSE.WHOLE.SOLUTION (see Figure 4-7) is an AND statement with three conditions: the design inputs are completed (problem-solving cycle is greater than 2), no design attribute has values and at least one retrieved design is on the blackboard.

```
TRIGGER.CONDITIONS:
    (AND (> (GET.VALUE 'CONTROL 'CURRENT.CYCLE) 2)
         (NULL (GET-DESIGN-SOLUTION-WFF-LIST))
         (GET.VALUE 'CONTROL 'RETRIEVED.DESIGNS))

CONTEXT.VARIABLES:
    ((($CASE $SOLUTION)
      (MAPCAR #'(LAMBDA (R-CASE)
                  (LIST R-CASE
                        (GET.VALUE (GET.VALUE R-CASE
                                              'CASE.NAME)
                                   'SOLUTION)))
              (GET.VALUES 'CONTROL 'RETRIEVED.DESIGNS))))

NAME.GENERATOR:   (REUSE SOLUTION FROM $CASE)

PRECONDITIONS:
    (< 40 (GET.VALUE $CASE 'SOLUTION.SIMILARITY))

ACTION.CODE:
    (LAMBDA (THISUNIT)
       (ASSERT (ADD-AND (GET.VALUE (GET.VALUE $CASE
                 'CASE.NAME) 'SOLUTION)) NIL
                                         $CURRENT.WORLD)
       (FORMAT T ~%The solution is: ~%)
       (LOOP FOR SOL IN '$SOLUTION DO (FORMAT T ~a~% SOL)))

DESCRIPTION:This knowledge source has no description yet.

PRIORITY:
    (COND ((> (GET.VALUE $CASE 'SOLUTION.SIMILARITY) 95)
           70)
          ((> (GET.VALUE $CASE 'SOLUTION.SIMILARITY) 80)
           50)
          (T 20))

ORIGINATED.CASE: $CASE

STATUS:   OPERATIVE
```

**Figure 4-7:** Knowledge Source REUSE.WHOLE.SOLUTION

- **CONTEXT.VARIABLES**—A list of variable-value pairs used to generate multiple KSARs from a single knowledge source. The variables are bound to the values when the KS is triggered. The variables are usually defined with a "$" sign and can be used in all other attributes of the KS except the TRIGGER.CONDITION. The value part of the list is a quoted list or a function that returns a list of values. If no value is provided for this attribute, this knowledge source generates only one KSAR when it is triggered. In the REUSE.WHOLE.SOLUTION example, two context variable $CASE and $SOLUTION are used to generate one KSAR for each retrieved design and to bind $SOLUTION to its previous solution. Let us assume that there are two

retrieved designs on the blackboard, CASE.1 and CASE.2. At trigger time, the function in the context variable attribute is evaluated and the list becomes `(($CASE $SOLUTION) '(case.1 case.1.solution) (case.2 case.2.solution)))`. Two KSARs are then generated by DDIS with bindings of `$CASE=case.1 $SOLUTION=case.1.solution` and `$CASE=case.2 $SOLUTION=case.2.solution` respectively. Also, there is an implicit semantic in the context variable list. Providing multiple lists in the attribute causes nested "for-loops" to be performed while generating bindings. For example, let's add a third context variable, $ADAPTATION.PARAMETER, to the list. Then the context variable list `(($CASE $SOLUTION) (MAPCAR #'(LAMBDA (...))) ($ADAPTATION.PARAMETER '(0.8 0.6)))` with two adaptation parameters generates four KSARs with bindings of `$CASE=case.1 $SOLUTION=case.1.solution $ADP.PARAMETER=0.8; $CASE=case.1 $SOLUTION=case.1.solution $ADP.PARAMETER=0.6; $CASE=case.2 $SOLUTION=case.2.solution $ADP.PARAMETER=0.8` and `$CASE=case.2 $SOLUTION=case.2.solution $ADP.PARAMETER=0.6` respectively. Using multiple context variable lists is a way to generate large number of design options in DDIS.

- **NAME.GENERATOR**—A list that translates to the name of each KSAR generated from this KS. For example, `(reuse solution from $case)` generates KSAR `reuse.solution.from.CASE.1, reuse.solution.from.CASE.2`, or whatever is bound to the context variable `$CASE` at trigger time. Note that critical context variables must be used in the form in order to generate distinct name for each KSAR.

- **PRECONDITION**—A LISP predicate that is passed to the KSARs generated from this knowledge source. It then becomes the precondition of those KSARs. Usually, variables defined in CONTEXT.VARIABLES are used in PRECONDITION to represent context specific requirements of the action. For example, the precondition of REUSE.WHOLE.SOLUTION is that the solution similarity rating of the retrieved design must be greater than 40. Every KSAR generated from this KS may have different similarity ratings and the ratings are not available at trigger time.

- **ACTION.CODE**—A LISP lambda expression that performs blackboard modifications. This action is carried out when the KSAR instantiated from this KS is executed.

- **DESCRIPTION**—A textual description of the knowledge source.

- **PRIORITY**—The priority of the knowledge source. Its value should be a constant or a LISP function that returns a number when evaluated. It represents

the importance of the KS under different conditions and is one of the rating elements of DDIS. The normal range of its value is between 0 and 100 (inclusive) with higher numbers signifying greater weights. In the given example, the priority of REUSE.WHOLE.SOLUTION depends on the solution similarity rating of the retrieved design. The proper use of this attribute and its role in rating is discussed in Section 4.3.2 and 4.3.5.

- **ORIGINATED.CASE**—A retrieved design object on the control blackboard from which this KS intends to transfer knowledge. This attribute is unique to case-dependent knowledge sources. It is used by the rating mechanism to trace the source of the knowledge that the KS uses in order to modify the rating according to the similarity of the retrieved design.

- **STATUS**—Either OPERATIVE or INOPERATIVE. If the status of the KS is inoperative, it is not considered in the problem-solving process.

#### 4.2.3.2. KSARS

A knowledge source activation record (KSAR) is created from a triggered knowledge source with all the context variables bound. KSARs are the action objects that actually perform the backboard modification task. In order to simplify KS writing and to describe several contexts of applicability of a KS and their different actions, DDIS allows the creation of multiple KSARs from a single knowledge source with multiple contexts when triggered. A KSAR is created at runtime for each possible combination of a triggered knowledge source and a set of problem variables (a context). KSARs instantiated from the same KS with different bindings have different names. The attributes of a KSAR and their values are largely inherited from its parent KS. The attributes of a KSAR that are carried over from the knowledge source object include TRIGGER.CONDITIONS PRECONDITION, ACTION.CODE, PRIORITY and ORIGINATED.CASE. However, every KSAR has three additional attributes:

- **VARIABLE.BINDINGS**—A list of dotted pairs that presents the context variables and their bindings. The variables are bound when the original KS is triggered and the KSAR is created. Figure 4-8 shows the REUSE.SOLUTION.FROM.CASE.2 KSAR created from knowledge source REUSE.WHOLE.SOLUTION. Note that all the context variables in the original KS are replaced with actual values now.

```
TRIGGER.CONDITIONS:
    (AND (> (GET.VALUE 'CONTROL 'CURRENT.CYCLE) 2)
         (NULL (GET-DESIGN-SOLUTION-WFF-LIST))
         (GET.VALUE 'CONTROL 'RETRIEVED.DESIGNS))

PRECONDITIONS:
    (< 40 (GET.VALUE CASE.2 'SOLUTION.SIMILARITY))

ACTION.CODE:
    (LAMBDA (THISUNIT)
       (ASSERT (ADD-AND (GET.VALUE (GET.VALUE CASE.2
                 'CASE.NAME) 'SOLUTION)) NIL
                                          $CURRENT.WORLD)
       (FORMAT T ~%The solution is: ~%)
       (LOOP FOR SOL IN
         '((THE BOLT.SPACING OF BOLT.GROUP.1 IS 6.75)
           (THE NUMBER.OF.BOLTS.PER.QUADRANT OF
                BOLT.GROUP.1 IS 1)
           (THE BOLT.DISTANCE OF BOLT.GROUP.1 IS 19.5)
           (THE DIAMETER OF BOLT.1 IS 0.625)
           (THE DIAMETER OF BASE.PLATE.1 IS 21.75)
           (THE MATERIAL OF BASE.PLATE.1 IS ASTM.A572.GR60)
           (THE THICKNESS OF BASE.PLATE.1 IS 0.75))
        DO (FORMAT T ~a~% SOL)))

PRIORITY: 20

ORIGINATED.CASE: CASE.2

VARIABLE.BINDINGS:
    (($CASE . CASE.2)
     ($SOLUTION . ((THE BOLT.SPACING OF BOLT.GROUP.1 IS
                        6.75)
                   (THE NUMBER.OF.BOLTS.PER.QUADRANT OF
                        BOLT.GROUP.1 IS 1)
                   (THE BOLT.DISTANCE OF BOLT.GROUP.1 IS
                        19.5)
                   (THE DIAMETER OF BOLT.1 IS 0.625)
                   (THE DIAMETER OF BASE.PLATE.1 IS 21.75)
                   (THE MATERIAL OF BASE.PLATE.1 IS
                        ASTM.A572.GR60)
                   (THE THICKNESS OF BASE.PLATE.1 IS
                        0.75))))

RATING:     ((FAVOR.CONTROL.ACTIONS 10 8 0)
             (START.DESIGN 5 5 20))

KS:      REUSE.WHOLE.SOLUTION
```

**Figure 4-8:** KSAR REUSE.SOLUTION.FROM.CASE.2

- **RATING**—The rating information of the KSAR under the current blackboard situation. When the KSAR is rated, the control mechanism of DDIS puts the individual ratings against each of the active goals in this attribute. In the

example, the two active goals on the blackboard are FAVOR.CONTROL.ACTIONS and START.DESIGN. The KSAR's ratings against these two goals are 0 and 20 respectively. The two numbers preceding each rating are the weight of the goal and the weight of its parent plan.

- KS—The knowledge source object from which the KSAR is instantiated. It is used as a link to the parent KS to provide additional information for rating, backtracking and design plan abstraction that is not locally available.

## 4.3. Control Strategy

The blackboard control strategy of DDIS can be broken down into five areas: problem-solving cycle, KSAR rating, plan and goal maintenance, goal expansion, and general scheduling criteria. A comprehensive understanding of these five sections is essential for understanding the integrated case-dependent and case-independent design of DDIS.

### 4.3.1. Execution Cycle

The basic approach to solve a design in DDIS is based on the blackboard model of problem solving. The design solution is generated on the blackboard incrementally by applying knowledge sources one at a time. The system is able to reason opportunistically because the sequence of executable knowledge source is determined dynamically based on the applicability of knowledge sources and the latest blackboard state (information on both control and design information blackboard).

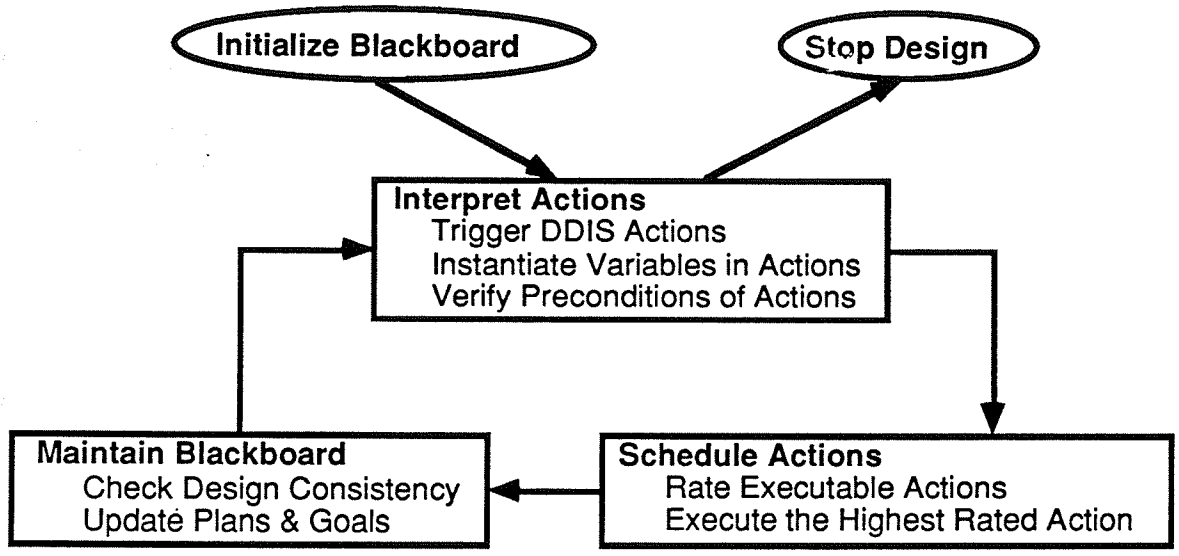



**Figure 4-9:** Execution Cycle of DDIS

The problem-solving cycle of DDIS (shown in Figure 4-9) involves three stages: blackboard updating, knowledge source interpreting, and scheduling:

- **Blackboard Updating**—At the beginning of every cycle, control objects on the control blackboard are updated according to the new blackboard modification resulting from the completion of the previous cycle (this updating process is discussed in detail later).

- **Knowledge Source Interpretation**—The applicability of every knowledge source (KS) under the current blackboard situation is determined by a three step process.

  - **Triggering**—A knowledge source is triggered when its primary requirements (the trigger condition) are satisfied. A triggered knowledge source is placed on the triggered agenda of DDIS.

    For example, the trigger condition of the CHECK.CONSTRAINTS KS (see Section 4.5.1.2) is the completion of problem input. When the problem input is completed, the KS is triggered and placed on the triggered agenda.

  - **Instantiating**—A knowledge source activation record (KSAR) is created for each possible combination of a triggered knowledge source and a set of problem variables (a context). In DDIS, KSs are written with context variables and are triggered by a blackboard state and context combination. Therefore, several KSARs can be created from a single KS when multiple triggering contexts exist on the blackboard.

    Instead of writing a specific constraint-checking KS for every single constraint, a context variable $CONST is used in CHECK.CONSTRAINTS to represent a unique constraint in the design problem. When CHECK.CONSTRAINTS is on the triggered agenda, DDIS instantiates it into as many KSARs as there are active constraints on the blackboard.

  - **Verifying**—After the triggering and instantiating process, every KSAR's context specific requirements (the precondition) are checked. A KSAR is executable and placed on the executable agenda when its precondition is met.

    For example, the precondition of CHECK.CONSTRAINTS KS is that the status of $CONST is unknown and the values of all the variables involved in $CONST are known. Therefore, DDIS checks the status and ingredient variables of each of constraint to determine if the corresponding constraint checking KSARs can be executed.

- **Scheduling**—Every KSAR on the executable agenda is rated according to the control objects on the blackboard (the plans and goals), and the highest rated

KSAR is recommended to the user for execution. The result of executing the KSAR is posted back to the blackboard, and the process repeats itself until the problem is solved or no executable KSARs are found.

Note that the problem solving cycle does not differentiate between case-dependent and case-independent knowledge sources. All the KSs are treated alike in the integrated system except that case-dependent knowledge sources are triggered by the presence of retrieved similar designs on the blackboard and have special context variables to associate them with the retrieved designs. The ratings of case-dependent KSARs, however, have to be adjusted according to the similarity of the case from which they transfer case-dependent knowledge. The strategies for rating are discussed next.

### 4.3.2. KSAR Rating

DDIS uses the control knowledge in plan and goal objects on the control blackboard to schedule the problem-solving actions. When case-dependent actions are involved, knowledge about design similarity in retrieved designs also plays a part in the schedule rating.

Each KSAR in the executable agenda is rated against every goal on the control blackboard by calling the function stored as the value of the FUNCTION attribute of the goal. These are the KSAR's individual ratings with respect to different active goals (see Section 4.3.5 for more information about individual ratings). To rate a KSAR with respect to the whole control blackboard situation, each individual rating is multiplied by the weight of each goal and each parent plan. Then the triple products for each KSAR is summed and divided by the maximum possible rating sum.

This weighting procedure takes into account of the relative importance of design plans and goals. The value of the weight attribute of a plan (or a goal) indicates its priority (see Section 4.2.1.1 and 4.2.1.2). A goal and a plan with higher weight values are considered more important than those with lower weights. Therefore, they should be more influential in scheduling. After all the executable KSARs are rated, DDIS recommends the highest rated action for execution. If there are no posted plans and goals on the control blackboard, the ratings for all KSARs are zero and the most recently invoked KSAR is recommended.

Control actions compete with domain actions, and case-dependent actions compete with case-independent actions for scheduling. The rating method described above applies to every KSAR except that the individual rating of a case-dependent KSAR has to be adjusted for the similarity of its source case.

#### 4.3.2.1. RATING FUNCTION FOR CASE-INDEPENDENT KSARS

The function DDIS uses to calculate the priority of a case-independent KSAR is:

$$\frac{\sum_{\text{GOAL}}^{\text{active goals}} \left( \begin{array}{c} \text{rating of the KSAR against GOAL} \\ \times \text{ weight of GOAL} \times \text{weight of GOAL's parent plan} \end{array} \right)}{\sum_{\text{GOAL}}^{\text{active goals}} \left( \text{weight of GOAL} \times \text{weight of GOAL's parent plan} \right)}$$

Stand-alone goals have no parent plans. They are usually important aspects that need to be dealt with promptly (like backtracking goals). Therefore, DDIS uses 10 as the weight of the non exist parent plan when calculating ratings of stand-alone goals.

To illustrate the use of the above equation, consider the blackboard state shown later in Figure 4-13. The active goals on the control blackboard are:

1. SELECT.MATERIAL (weight = 5) from
   CASE.2.DESIGN.PLAN (weight = 20/3),
2. DESIGN.BEAM-COLUMN (weight = 5) from
   BEAM-COLUMN.SYSTEM.DESIGN.PLAN (weight = 5) and
3. FAVOR.CONTROL.ACTIONS (weight = 8), which is
   the stand-alone system default goal discussed in Section 4.3.5.

Then, the overall rating of KSAR USE.ASTM.A36.STEEL.FOR.BEAM.COLUMN against all the active goals on the control blackboard is calculated as follow:

$$\frac{R_{\text{SELECT.MATERIAL}} + R_{\text{DESIGN.BEAM-COLUMN}} + R_{\text{FAVOR.CONTROL.ACTIONS}}}{(5 \times \frac{20}{3}) + (5 \times 5) + (8 \times 10)}$$

$$= \frac{(72 \times 5 \times \frac{20}{3}) + (18 \times 5 \times 5) + (0 \times 8 \times 10)}{(5 \times \frac{20}{3}) + (5 \times 5) + (8 \times 10)} = \frac{2400 + 450 + 0}{138.333} = 20.6$$

in which

$R_{\text{SELECT.MATERIAL}}$ = (the rating of the KSAR against SELECT.MATERIAL
x the weight of SELECT.MATERIAL
x the weight of CASE.2.DESIGN.PLAN)
= $(72 \times 5 \times \frac{20}{3})$

$$R_{\text{DESIGN.BEAM-COLUMN}} = (\text{the rating of the KSAR against DESIGN.BEAM-COLUMN} \times \text{the weight of DESIGN.BEAM-COLUMN} \times \text{the weight of BEAM-COLUMN.SYSTEM.DESIGN.PLAN}) = (18 \times 5 \times 5)$$

$$R_{\text{FAVOR.CONTROL.ACTIONS}} = (\text{the rating of the KSAR against FAVOR.CONTROL.ACTIONS} \times \text{the weight of FAVOR.CONTROL.ACTIONS} \times 10) = (0 \times 8 \times 10)$$

The action USE.ASTM.A36.STEEL.FOR.BEAM.COLUMN matches the intent of the SELECT.MATERIAL goal, partly serves the DESIGN.BEAM-COLUMN goal and is irrelevant for goal FAVOR.CONTROL.ACTIONS. Therefore, the individual rating of the KSAR against these three goals are 72, 18 and 0, respectively.

### 4.3.2.2. RATING FUNCTION FOR CASE-DEPENDENT KSARS

The priority of a case-dependent KSAR is determined by the following function:

$$\frac{\sum_{\text{GOAL}}^{\text{active goals}} \left( \text{Min}\left[ 100 \text{ , rating of the KSAR against GOAL} \times \frac{\text{similarity rating of KSAR's originated design}}{\text{base similarity of DDIS}} \times \text{weight of GOAL} \times \text{weight of GOAL's parent plan} \right] \right)}{\sum_{\text{GOAL}}^{\text{active goals}} (\text{weight of GOAL} \times \text{weight of GOAL's parent plan})}$$

The base similarity represents the threshold value for case similarity adjustment. It is a global constant in DDIS and currently set to 60. If the similarity rating of the KSAR's originated case is greater than the base similarity, the individual rating of the KSAR is increased. The higher the case similarity rating is the better the KSAR is for transferring case-dependent knowledge from the case. On the other hand, the individual rating of the KSAR is decreased if the case similarity rating is less than the base similarity. Any individual rating of a KSAR against a goal should not be greater than 100, so the maximum individual rating after case similarity adjustment is 100.

The case similarity rating used in the above equation can be plan similarity or solution similarity. When the action being rated is in the solution transformer knowledge module, the solution similarity of the retrieved design where the action intends to transfer the solution is used. If the action is in the plan transformer knowledge module, the plan similarity is used. One exception to this rating method is the memory prober, which uses the case-independent KSAR rating function instead of the case-dependent function. No case similarity modification of the rating is needed (or possible) for the memory probers because that they are case-dependent control KSs that change the retrieved design level of the control blackboard but do not transfer particular case-dependent knowledge from any retrieved designs.

The following example using KSAR REUSE.ASTM.A36.STEEL.FROM.CASE.3 in Figure 4-13 illustrates the rating function for case-dependent KSARs:

$$\frac{R_{SELECT.MATERIAL} + R_{DESIGN.BEAM\text{-}COLUMN} + R_{FAVOR.CONTROL.ACTIONS}}{(5 \times \frac{20}{3}) + (5 \times 5) + (8 \times 10)}$$

$$= \frac{(42 \times \frac{78}{60} \times 5 \times \frac{20}{3}) + (14 \times \frac{78}{60} \times 5 \times 5) + (0 \times \frac{78}{60} \times 8 \times 10)}{(5 \times \frac{20}{3}) + (5 \times 5) + (8 \times 10)} = \frac{1820 + 455 + 0}{138.333} = 16.45$$

in which

$R_{SELECT.MATERIAL}$ = (the rating of the KSAR against SELECT.MATERIAL
x $\frac{\text{solution similarity of CASE.3}}{\text{base similarity of DDIS}}$
x the weight of SELECT.MATERIAL
x the weight of CASE.2.DESIGN.PLAN)

$= (42 \times \frac{78}{60} \times 5 \times \frac{20}{3})$

$R_{DESIGN.BEAM\text{-}COLUMN}$ = (the rating of the KSAR against DESIGN.BEAM-COLUMN
x $\frac{\text{solution similarity of CASE.3}}{\text{base similarity of DDIS}}$
x the weight of DESIGN.BEAM-COLUMN
x the weight of BEAM-COLUMN.SYSTEM.DESIGN.PLAN)

$= (14 \times \frac{78}{60} \times 5 \times 5)$

$R_{FAVOR.CONTROL.ACTIONS}$ = (the rating of the KSAR against FAVOR.CONTROL.ACTIONS
x $\frac{\text{solution similarity of CASE.3}}{\text{base similarity of DDIS}}$
x the weight of FAVOR.CONTROL.ACTIONS x 10)

$= (0 \times \frac{78}{60} \times 8 \times 10)$

The active goals on the control blackboard are stated in the previous section. Since REUSE.ASTM.A36.STEEL.FROM.CASE.3 is instantiated from REUSE. PARTIAL.SOLUTION in the solution transformer knowledge module and the source of the solution transfer is CASE.3, the solution similarity rating of CASE.3 (i.e., 78) is used in the rating adjustment.

## 4.3.3. Plan and Goal Maintenance

The plans and goals posted on the control blackboard contain the control information DDIS needs for scheduling. The applicability of plans and goals can change every cycle due to the changing blackboard state. Therefore, plans and goals need to be monitored and updated every cycle by DDIS. The structures of plan and goal objects are described in Section 4.2.1.1 and Section 4.2.1.2. This section discusses their maintenance during problem solving.

### 4.3.3.1. PLAN UPDATING

At the beginning of every execution cycle, the plans on the control blackboard are updated. The following is the procedure for updating the plans:

1. **Check applicability of plans.** The intentions of all active plans are checked. If the intention of a plan is satisfied (evaluates to true according to the current blackboard state), the plan is no longer applicable (either the intended purpose of the plan is already accomplished or it is no longer desirable).

2. **Remove inapplicable plans.** The plans that are no longer applicable are removed from the control blackboard. This triggers the goal updating stated in the next section.

3. **Check existence of retrieved designs.** The existences of the corresponding cases of every case-dependent plan are checked. Case-dependent plans should only be considered when their associated case is retrieved. The absence of the retrieved design means that it is no longer a source of case-dependent knowledge in the design session.

4. **Remove orphaned case-dependent plans.** The case-dependent plans whose originated case is no longer on the blackboard are removed from the control blackboard. This also triggers the goal updating stated in the next section.

### 4.3.3.2. GOAL UPDATING

After the plans are updated, the goal maintenance follows. It goes through the following steps:

1. **Check applicability of stand-alone goals.** The intentions of every active stand-alone goal (goals not included in any plans) are checked. If the intention of a goal is satisfied, the goal is no longer applicable (either the

intended purpose of the goal is already accomplished or it is no longer desirable).

2. **Remove inapplicable stand-alone goals.** The stand-alone goals that are no longer applicable are removed from the control blackboard.

3. **Remove orphaned goals.** The goals whose parent plans are no longer on the blackboard are removed from the control blackboard.

4. **Update active goals of plans.** Previously satisfied goals of every active plan on the blackboard are rechecked. If the intention of a previously achieved goal is not satisfied according to the current blackboard state, the goal needs to be satisfied again, and it becomes the active goal of the plan. If all the achieved goals are still satisfied, check the applicability of the currently active goals of the plan. If the current active goals are all satisfied, set the ACTIVE.GOAL of the plan to the next applicable goal(s) in the GOAL.LIST as the new active goal(s).

5. **Activate new active goals.** All the active goals of the current plans are posted on the blackboard. This new set of design goals supplies the rating objects for the new execution cycle.

6. **Remove plans with no active goals.** Plans with an empty ACTIVE.GOAL are removed from the control blackboard. When all the goals in a plan's GOAL.LIST are satisfied, its ACTIVE.GOAL is NIL.

### 4.3.4. Goal Expansion

Like the plan and goal updating mechanism, goal expansion is an important control inference method in DDIS. A global design goal can be dynamically expanded into a more detailed design plan (or plans) at run time. The individual goals of that plan may be further expanded if specific plans are available. This top-down plan decomposition with several levels of abstraction is goal expansion (see Figure 4-10 for an example). An active goal on the blackboard becomes expandable when there is at least one plan in the knowledge base that has the same intention as the goal. One exception is that the goal must not belong to the plan (usually, the intention of the last goal of a plan is the same as the intention of the plan). Goal expansion is achieved by the generic control knowledge source EXPAND.GOAL. The following is a description of this KS.

- **EXPAND.GOAL** expands design goals on the blackboard into case-independent plans with same intention in the knowledge base of DDIS. The trigger condition of EXPAND.GOAL is that at least one of the active goals on the blackboard is expandable (i.e., the intention of at least one plan in the knowledge base is the same as one of the active goals' intention). Context variables $EXPANDABLE-GOAL and $PLAN are used to generate one KSAR

for each design plan found with more specific subgoals of achieving the expandable design goals on the blackboard. The precondition is that $PLAN isnot already on the blackboard. The execution of the KS adds $PLAN on the control blackboard as an expansion of $EXPANDABLE-GOAL. In order to reflect the different importance of each expanded goal, EXPAND.GOAL gives the expansion plan a weight equal to the weight of the expanded goal.

Design goals can also be expanded to case-dependent plans. REUSE.PREVIOUS.PLAN is the case-dependent version of EXPAND.GOAL. Its intention is to expand goals to past case-dependent plans in the case memory (see Section 4.5.2.3.2.). When a case-dependent plan is reused as an expansion of a goal, the weight of the plan needs to be adjusted to account for the similarity of the design case it comes from. The following equation is used by REUSE.PREVIOUS.PLAN to calculate the weight of the case-dependent plan:

$$\text{(the weight of the goal to be expanded} \times \frac{\text{the plan similarity rating of the retrieved design}}{\text{the base similarity of DDIS}})$$

However, the weight of a plan cannot be greater than 10. If the calculated value is greater than 10, the weight of the case-dependent plan is only 10. The base similarity is the default similarity value used in various KSAR rating functions of DDIS. It is currently set to 60 on the control blackboard.

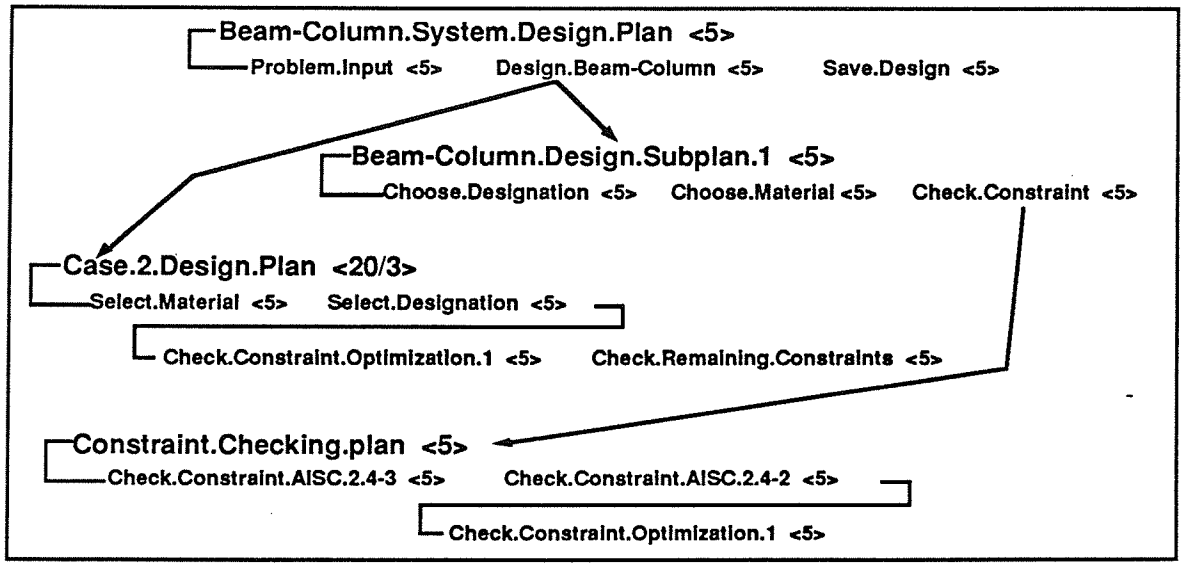



**Figure 4-10:** An Example of the DDIS's Goal Expansion

For example, when REUSE.CASE.2.DESIGN.PLAN.FOR.DESIGN.BEAM-COLUMN is executed to expand the active goal DESIGN.BEAM-COLUMN (see cycle 5 of the first beam-column design session example in Section 5.1.1), the case-dependent CASE.2.DESIGN.PLAN is posted with a weight of 20/3 resulting from the following calculation:

$$(\text{the weight of DESIGN.BEAM-COLUMN} \times \frac{\text{the plan similarity rating of CASE.2}}{\text{the base similarity of DDIS}}) = 5 \times \frac{80}{60} = 20/3$$

It is possible that several expansions (including case-dependent and case-independent expansions) are available for a given design goal. Then, we have several goal expansion KSARs in the executable agenda for that goal. We may also have KSARs for other expandable goals. Like all other KSARs, they need to be rated by DDIS and compete with other domain actions for execution. The rating of goal expansion KSs is discussed in the next section.

### 4.3.5. General Scheduling Criteria

The general scheduling rule of DDIS is to prefer control KSARs to domain KSARs and prefer case-independent KSARs to case-dependent KSARs. DDIS favors control actions using the system default goal FAVOR.CONTROL.ACTIONS (with a weight of 8), which does not show on the blackboard interface. When there are no other plans and goals on the blackboard, control actions are always favored. However, in the presence of other design plans and goals, the significance of FAVOR.CONTROL.ACTIONS in the overall design plan decreases, and control actions have to compete with other domain actions for execution.

In DDIS, each design goal rates KSARs by calling the function stored in the FUNCTION attribute of the goal. Usually, the KSAR rating function of a goal accesses the PRIORITY attribute of KSARs for ranking information. If the value stored in a KSAR's PRIORITY attribute is a constant, the priority of the KSAR is not changed with changing blackboard states. On the other hand, the KSAR's priority can be a dynamic value if the PRIORITY attribute is a function involving some blackboard objects. Dynamic rating can also be achieved by including different rating conditions in the FUNCTION attribute of goals to account for different blackboard situations. The recommended method of writing KSs and goals is to leave the dynamic rating to the PRIORITY attribute of KSARs and always get the priority value in the KSAR rating function of goals. Therefore, the general priority of KSARs can be predefined for various types of KSs and inherited by their children through the class hierarchy of DDIS. For example, case-dependent KSs have a default priority of 70, case-independent KSs have a default priority of 90, the memory probers have a default priority of 85, etc. The general rule of preferring case-independent KSARs to case-dependent KSARs is implied in those numbers.

The knowledge source EXPAND.GOAL is an example of dynamic KSAR rating. The PRIORITY of EXPAND.GOAL is described by the following conditional statements:

| | |
|---|---|
| IF | the goal in consideration has not been expanded yet |
| THEN | the priority is 80 |
| IF | the goal has only been expanded to case-dependent plan(s) |
| THEN | the priority is 60 |
| IF | the goal has only been expanded to case-independent plan(s) |
| THEN | the priority is 40 |
| IF | the goal has both case-independent and case-dependent expansions |
| THEN | the priority is 20 |

The rationale is that the priority of a goal expansion action changes with the amount and type of subplans that the expandable goal already has. However, goal expansion actions should be able to be rated according to the importance of the goal they aim to expand. Therefore, the PRIORITY of EXPAND.GOAL is modified to be the value of the conditional function described above times the *goal importance index*, which is the weight of the expandable goal divided by 8 (an experimental value from running the demonstration applications).

For instance, KSAR EXPAND.DESIGN.BEAM-COLUMN.INTO.BEAM-COLUMN.DESIGN.SUBPLAN.1 in Figure 4-13 is rated as follows:

$$\frac{R_{\text{SELECT.MATERIAL}} + R_{\text{DESIGN.BEAM-COLUMN}} + R_{\text{FAVOR.CONTROL.ACTIONS}}}{(5 \times \frac{20}{3}) + (5 \times 5) + (8 \times 10)}$$

$$= \frac{0 + 0 + ((60 \times \frac{5}{8}) \times 8 \times 10)}{(5 \times \frac{20}{3}) + (5 \times 5) + (8 \times 10)} = \frac{0 + 0 + (37.5 \times 8 \times 10)}{138.333} = \frac{3000}{138.333} = 21.69$$

in which

$R_{\text{SELECT.MATERIAL}}$ = (the rating of the goal expansion KSAR against SELECT.MATERIAL
x the weight of SELECT.MATERIAL
x the weight of CASE.2.DESIGN.PLAN)

$= (0 \times 5 \times \frac{20}{3}) = 0$

$R_{\text{DESIGN.BEAM-COLUMN}}$ = (the rating of the goal expansion KSAR
against DESIGN.BEAM-COLUMN
x the weight of DESIGN.BEAM-COLUMN
x the weight of BEAM-COLUMN.SYSTEM.DESIGN.PLAN)

$= (0 \times 5 \times 5) = 0$

$R_{\text{FAVOR.CONTROL.ACTIONS}}$ = (the rating of the goal expansion KSAR
against FAVOR.CONTROL.ACTIONS
x the weight of FAVOR.CONTROL.ACTIONS x 10)

= ((the priority of the goal expansion KSAR
x the goal importance index) x 8 x 10)

$$= \left(\left(60 \times \frac{\text{the weight of DESIGN.BEAM-COLUMN}}{8}\right) \times 8 \times 10\right)$$

$$= \left(\left(60 \times \frac{5}{8}\right) \times 8 \times 10\right)$$

The goal expansion KSAR is a control knowledge source and does not directly contribute to the design of the beam-column and the selection of its material. Therefore, it is rated zero against SELECT.MATERIAL and DESIGN.BEAM-COLUMN. On the other hand, the goal FAVOR.CONTROL.ACTIONS accesses the KSAR'S PRIORITY slot value as the KSAR's rating against favoring control actions. The PRIORITY of the goal expansion KSAR is the product of the goal importance index and the value of the conditional function described above, which is determined according to the blackboard situation at run time. The value of the conditional function is 60 because DESIGN.BEAM-COLUMN has already been expanded to case-dependent CASE.2.DESIGN.PLAN on the control blackboard. The goal importance index is the weight of DESIGN.BEAM-COLUMN (i.e., 5) divided by 8. Therefore, the PRIORITY of the goal expansion KSAR is 60 times 5 divided by 8, which is 37.5.

## 4.4. Case Memory

The case memory knowledge base is the resource of case-based design in DDIS. Newly solved designs can be stored in the memory for reuse by the case-based reasoner. The case recorder captures case-dependent knowledge in several forms. The case memory needs to be able to accommodate all of them and still let the memory prober find them effectively. Therefore, the memory is divided into three parts: a memory of solutions, a memory of plans and a memory of goals, with interrelated objects (all knowledge bases in DDIS consist of objects). Three kinds of objects are used in the case memory: design cases, case-dependent plans and case-dependent goals. Figure 4-11 shows the structure of the case memory with a sample case base.

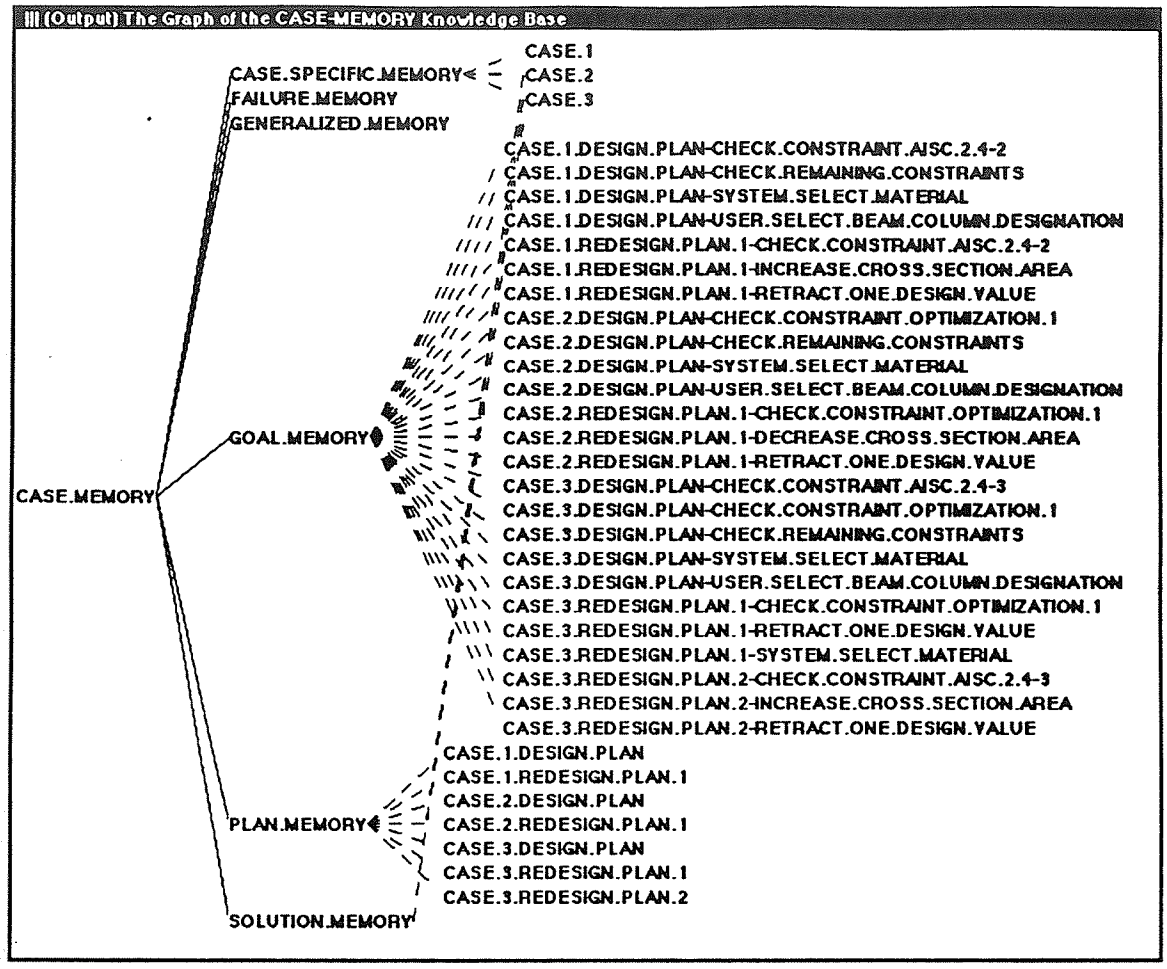



**Figure 4-11:** The Hierarchy of Case Memory Knowledge Base in DDIS

## 4.4.1. Design Cases

A design case is the primary storage element of case-dependent knowledge in DDIS. It stores information about a particular previous design. The case recorder is responsible for creating design case objects and saving them in the memory. The following are required attributes of a design case.

- **CASE.NAME**—The name of the design case.
- **DESIGN.HISTORY**—A sequence of the executed KSs and their bindings. Its value is a list of KSs and variable binding pairs.
- **DESCRIPTION**—A textual description of the design case.

- DESIGN.PLAN—A case-dependent plan object representing the control knowledge used in the previous design. The creation of the object is discussed in the next section.

- PROBLEM.INPUT—The problem specification of the design. It holds the information about the problem input variables and their values.

- REDESIGN.PLANS—Several case-dependent redesign plan objects representing the backtracking strategies used in the previous design. The creation of the objects is discussed in the next section.

- SOLUTION—The final solution of the design.

- STORAGE.LOCATION—A complete file path name representing the physical storage location of all the propositions of the design.

### 4.4.2. Case-Dependent Plans

A case-dependent plan is a subclass of the plan object discussed in Section 4.2.1.1. It stores the problem-solving or backtracking strategy of a particular previous design. It specifies the sequence of design goals achieved by the previous design steps. The case-dependent goals are discussed in the next section.

The case recorder is responsible for creating case-dependent plan objects and saving them in the memory. The control knowledge of a previous design session can be abstracted to one global design plan and several redesign plans. The process that the SAVE.DESIGN.SESSION knowledge source (see Section 4.5.3) uses to capture design plans is stated below:

- **Identify major design actions.** The design history is analyzed, and all KSs that modified the solution blackboard are gathered. This step is to filter out unnecessary design steps that do not directly contribute to the solution process (e.g., control KSs that only modify the control blackboard).

- **Create case-dependent goals.** A case-dependent goal is created for each identified major action (KS) in order to prefer the same KS in the future. This process is discussed in detail in the next section.

- **Differentiate design and redesign goals.** The major design actions can be classified into design and redesign actions. Therefore, the case-dependent goals are assigned to one design plan that represents the major design path and several redesign plans that represent the various backtracking processes. The goals then make up the GOAL.LIST of their plan.

- **State the intention of the plans.** The intention of a global plan is to generate design value for all the design attributes and to satisfy all the applicable constraints of the design. The intention of a case-dependent redesign plan is to satisfy all unsatisfied constraints that triggered the redesign process.

### 4.4.3. Case-Dependent Goals

The characteristics of case-dependent goals are discussed in Section 4.2.1.2. The case recorder is responsible for creating case-dependent goal objects and saving them in the goal memory of the case memory knowledge base. A case-dependent goal represents one step of the case-dependent plan that it belongs to. It contains a rating function to evaluate the usefulness of future KSARs for reproducing the effect that resulted from the past action taken at that step.

Three different rating conditions are created for each case-dependent goal:

1. The highest rating is given to the KSARs instantiated from the same KS that previously triggered the KSAR used to accomplish the goal (i.e., KSARs from the same KS).

2. KSARs that modify the same design objects and attributes attain a moderate rating.

3. Same type of actions (i.e., KSARs with same parent class as the previously used KSAR) score a limited rating.

In this way, each step of a case-dependent plan can be followed precisely, closely or loosely by DDIS's control mechanism when the past plan is reused by the plan transformers. However, further experiment action shows that the usefulness of previous case-dependent actions varies largely with the similarity between the new and past design. The truly valuable decisions are those that select the design objects and attributes to design for. So, only the second rating condition stays when case-dependent goals are extracted from case-dependent actions. On the other hand, the following rating condition is added to the top of the list for case-dependent goals that are extracted from previous backtracking provoking actions.

0. The best rating is given to the KSARs that are the same as previously used to accomplish the redesign goal (i.e., KSARs from the same KS with the same bindings).

The level of backtracking is the most important guidance that a previous backtracking action can give to later design sessions. The backtracking level is recorded with the bindings of the previouly executed backtracking provoker KSAR (see Section 4.5.1.3 for details of the

KS). Therefore, to capture that decision, four rating conditions are used (with highest rating given by the condition stated in item zero) instead of three.

The intention of a case-dependent goal is the negation of the trigger condition of the KS that previously accomplished the design step. A goal is not applicable when its intention is true. The action is not appropriate or is already executed when its trigger condition is not true. Therefore, when the trigger condition of a previous action is not true, the case-dependent goal denoting that action is not applicable (either the goal is already accomplished or the intended purpose of the goal is not desirable). The formulation works well with the plan and goal updating mechanism of DDIS. Inapplicable goals from past plans are delayed or skipped under this setup.

## 4.5. Knowledge Base

The knowledge base of DDIS consists of a number of knowledge sources. The KSs can be grouped into three modules: the case-independent reasoner, the case-based reasoner and the case recorder. The knowledge modules and their knowledge sources are described in this section.

### 4.5.1. Case-Independent Reasoner

Four subsidiary knowledge modules are included in the case-independent reasoner: the design generator, constraint checker, backtracking provoker and redesign adviser. The KSs included in this module are the fundamental design actions for implementing the design model discussed in Section 3.2.

#### 4.5.1.1. DESIGN GENERATOR

The function of the design generator is to generate values for design variables. The knowledge sources in this module are capable of generating different values for the same design variable (or a set of variables) using different case-independent knowledge including design heuristics, design analysis, design calculation, database information, etc. Knowledge source SYSTEM.SELECT.DESIGNATION of beam-column design problem is an example of the design generator.

- **SYSTEM.SELECT.DESIGNATION** calculates the required cross section area of the beam-column based on the heuristic equation

$$A_{required} = \frac{(P + (M \times 0.2 \times 12))}{F_y}$$

and recommends a w-shape section that has at least that area.

The trigger condition of the SYSTEM.SELECT.DESIGNATION KS is true if the designation of the beam-column is unknown, the end moment (M) is known, the axial load (P) is known, and the yield stress ($F_y$) of the material of the beam is known. One context variable is used for this KS. When the KS is triggered, the $W-SHAPE-FOUND context variable is bound to the smallest AISC W-shape structural section that satisfies the cross section area requirement calculated by the heuristic equation. The precondition is that $W-SHAPE-FOUND is not NIL and is not equal to the most recently used designation. DDIS posts $W-SHAPE-FOUND as the design value of the designation of the beam-column on the blackboard when the action is executed.

#### 4.5.1.2. CONSTRAINT CHECKER

The function of constraint checker knowledge module is to perform constraint confirmation. When constraint violations are found, it posts "fix constraint violation" goals on the blackboard to direct DDIS's attention to design modifications. The constraint checker of DDIS has three generic knowledge sources CHECK.CONSTRAINTS, ACTIVATE.CONSTRAINTS and USER.SPECIFY.MAJOR.CONSTRAINTS.

- **CHECK.CONSTRAINTS** is written with one context variable, which can be instantiated to different constraints on the blackboard to create multiple constraint checking KSARs.

  The trigger condition of the KS is the completion of problem input. DDIS generates one check-constraint KSAR for every constraint object on the blackboard at trigger time. For every KSAR generated, the $CONST context variable is bound to one unique constraint in the design problem. The precondition of the KS is that the status of $CONST is unknown and the values of all the variables involved in $CONST are known. The action of the KS is to post the status of $CONST (the possible values are satisfied and unsatisfied) on the blackboard. If $CONST is unsatisfied, a FIX.$CONST.VIOLATION goal is created and posted on the blackboard.

- **USER.SPECIFY.MAJOR.CONSTRAINTS** is an action that can be used to relax constraints. It is sometimes useful to relax secondary constraints during design, letting the system concentrate on the critical constraints and save execution time.

  The KS is triggered when there are still unchecked constraints (i.e., constraints with an unknown status) on the blackboard. No context variables are used in this KS. When the KS is executed, the user can select unchecked constraints from a table that he/she wants DDIS to focus on. The selected constraints then become the major constraints of the design session. A "check critical constraint" goal is created and posted on the blackboard for every major

constraint. Thus, DDIS can favor the actions for checking those major constraints when they are ready to be checked. At the same time, the values of the applicability attribute of all the unselected constraints are changed to "relaxed" (the default value is "activated"), which prevents the action for checking these constraints from becoming executable (because the preconditions of the corresponding KSARs are false).

- **ACTIVATE.CONSTRAINTS** is used to make previously relaxed constraints active again. Usually, it is executed when major constraints are already satisfied, and the user wants to check the secondary constraints.

  The KS is triggered when there are constraints on the blackboard that have a status of "relaxed". Since this KS does not have context variables and preconditions, only one executable KSAR is created after triggering. When the KS is executed, the user can select relaxed constraints from a table to make them active again. The relaxed applicability of all the selected constraints are removed from the blackboard. The statuses of these constraints become unknown, which can trigger more constraint checking actions.

#### 4.5.1.3. BACKTRACKING PROVOKER

The backtracking provoker knowledge module initiates the redesign process when a design failure is found. It controls the level to which DDIS backtracks. Certain design values on the solution blackboard are retracted when a backtracking provoker is executed. The truth maintenance system of DDIS then resets the blackboard state, and all other KSs compete to solve the design again. One example of the backtracking provoker is the RETRACT.ONE.DESIGN.VALUE generic KS.

- **RETRACT.ONE.DESIGN.VALUE** retracts one attribute value of a design object that is responsible for the current design failure. The assumption behind this KS is that we modify one design value at a time for one constraint violation.

  The KS is triggered when the status of at least one design constraint is unsatified. This KS uses the justifications established by the truth maintenance system during design to find out all the attribute values of design objects that are responsible for the currently unsatisfied constraints. Then, context variable $DESIGN-OBJECT and $SLOT are used in this KS to generate one KSAR for every pair of design object and attribute involved in the violated constraints. This particular KS has no precondition. The execution of the KS retracts the $SLOT value of $DESIGN-OBJECT from the solution blackboard.

#### 4.5.1.4. REDESIGN ADVISER

The function of the redesign adviser knowledge module is to give redesign advice. The backtracking provoker is different from the redesign adviser. The former suggests what design value to fix, and the latter recommends how to fix it. A redesign adviser is triggered after a backtracking provoker removes values from the solution blackboard. Analyzing the violated constraint(s) and design history, the applicable redesign adviser become executable and can be evoked to modify the design.

The redesign advisers are usually domain dependent. For example, INCREASE.CROSS.SECTION.AREA, INCREASE.BEAM.COLUMN.DEPTH and DECREASE.CROSS.SECTION. AREA are three particular redesign adviser KSs in the beam-column design knowledge base of DDIS. Each is capable of generating several possible values using context variables for redesigning certain parts of the beam-column.

### 4.5.2. Case-Based Reasoner

The case-based reasoner performs the case-based design task in the integrated design system. Three subsidiary knowledge modules are included in the case-based reasoner: the memory prober, the failure anticipator and the analogy transformer.

#### 4.5.2.1. MEMORY PROBER

The memory prober knowledge module is the retrieval mechanism of the case-base reasoner. Its function is to locate previous design cases stored in the case memory knowledge base that are similar to a new design and rank them with respect to a particular class of problems. This version of DDIS has very limited ability in this area. All previous cases that are potentially useful under the target condition are presented to the user for evaluation and selection.

Three generic knowledge sources, RETRIEVE.SIMILAR.CASES, FIND. SIMILAR.CASES. FOR.REDESIGN and RANK.CASES, are in the current DDIS knowledge base.

- **RETRIEVE.SIMILAR.CASES** searches the case memory for previous similar designs as the source of case-dependent reasoning. At this point DDIS does not have any knowledge about how to measure similarities between design cases. Therefore, RETRIEVE.SIMILAR.CASES presents all the specific designs in the case memory to the user for retrieval at the beginning of a design session.

  The trigger conditions of the KS are the completion of problem input and the lack of retrieved designs on the blackboard. No context variables are used for this KS. When the KS is triggered, only one KSAR is generated and it is

always executable (precondition is T). The execution of the KS results in the posting of retrieved design objects for each similar case selected by the user.

- **FIND.SIMILAR.CASES.FOR.REDESIGN** searches the case memory for previous design plans to fix constraint violations. It is similar to RETRIEVE.SIMILAR.CASES, but it examines the memory from a different point of view. Plans associated with design cases are the target of the action. All the cases in the memory encountered the same constraint violation are presented to the user for retrieval.

  The trigger condition of the KS is true if there are any "fix constraint violation" goals on the control blackboard. It has a $FIX-CONSTRAINT-VIOLATION-GOAL context variable in order to generate one KSAR for each constraint violation. The precondition is that the case memory has not been probed for design plans to resolve this $FIX-CONSTRAINT-VIOLATION-GOAL yet. The action of this KS is to find cases that have redesign plans to resolve the constraint violation (i.e., with the same intention as $FIX-CONSTRAINT-VIOLATION-GOAL). All the cases found are presented to the user in a pop-up window for selection, and the chosen cases are placed on the blackboard as the new retrieved designs.

- **RANK.CASES** asks the user to rank each retrieved design regarding the usefulness of reusing its solution and its affiliated plans respectively. Numerical ratings are used. A rating of 100 is the best a retrieved design can get, meaning that the case is exactly the same as the new problem. Likewise, zero rating is the worst, meaning that the case is not considered relevant to the new problem.

  The trigger condition of the KS is true if there are unrated design cases on the blackboard retrieved by the memory prober. No context variables are used for this KS. When the KS is triggered, only one KSAR is generated and it is always executable (precondition is T). The action of this KS is to rate each retrieved design and post the similarity ratings on the blackboard.

#### 4.5.2.2. FAILURE ANTICIPATOR

The failure anticipator knowledge module uses DDIS's memory of previous design failures. Remembering relevant previous design failures in a new design can help DDIS detect unsuccessful designs earlier and correct them sooner. The DDIS's failure anticipation is limited to one KS at this time.

- **REUSE.PREVIOUS.CRITICAL.CONSTRAINTS** declares the major constraints of a new design based on the critical constraints (i.e., constraints that had been violated during the previous design session) of a retrieved

similar design. This action is usually used at the beginning of a design session to let DDIS pay special attention to the previously violated constraints. Constraints that were hard to satisfy in a similar past design are likely to be troublesome in the new design. This action can help DDIS identify critical constraints and prevent important constraints being relaxed.

The KS is triggered when at least one retrieved design and no relaxed constraints are on the blackboard. Context variable $CASE is used to create one KSAR for each retrieved design. The KSAR becomes executable when design failures and modifications exist in $CASE, the status of at least one previously violated constraints is unknown in the new design, and the plan similarity rating of $CASE is greater than 25. When the KS is executed, the user can select critical constraints of the former design from a table to be the major constraints of the new design. A "check critical constraint" goal is created and posted on the blackboard for every major constraint. So, DDIS can favor the actions for checking those major constraints when they are ready to be checked. At the same time, the values of the applicability attribute of all the unselected constraints are changed to "relaxed" (the default value is "activated"), which gives these constraints a status of "relaxed" and prevents the action for checking these constraints from becoming executable (because the preconditions of the corresponding KSARs are false).

#### 4.5.2.3. ANALOGY TRANSFORMER

The analogy transformer knowledge module is the case-based reasoning mechanism of the case-based reasoner. It transfers case-dependent knowledge from the source designs (retrieved by the memory prober) to a new design. The analogy transformer is further divided into two subsidiary modules, the solution transformer and the plan transformer, corresponding to two levels of abstraction of case-dependent knowledge currently in DDIS.

##### 4.5.2.3.1. Solution Transformer

REUSE.WHOLE.SOLUTION, REUSE.PARTIAL.SOLUTION, REUSE.PREVIOUS.DESIGN.OBJECT and REUSE.PREVIOUS.DESIGN are the four generic knowledge sources in the solution transformer module.

- **REUSE.WHOLE.SOLUTION** reuses the previous solution of a similar design as the initial attempt of the new design. Frequently, previous design solutions provide good starting points for new design problems.

  The KS is triggered when the problem input is completed, all the design values are unknown, and at least one retrieved design is on the blackboard. Context variable $CASE and $SOLUTION are used to generate one KSAR for each

retrieved design and to bind $SOLUTION to its previous solution. The precondition is that the solution similarity rating of the retrieved design must be greater than 40, which means that the solutions of previous cases not very similar to the new design are not worth reusing. The execution of the KS asserts $SOLUTION from $CASE as the trial solution of the new design on the solution blackboard.

- **REUSE.PARTIAL.SOLUTION** takes part of the solution in the previous design and reuses it to achieve an active goal of the new design. This KS intends to target the granularity problem of case-dependent knowledge. The whole solution of a previous design is not always reusable. This KS helps DDIS focus on the current active goal on the control blackboard to determine what part of the previous solution can be reused.

  The trigger conditions of the KS are the completion of problem input, the presence of retrieved designs on the blackboard, and the existence of at least one unsolved design parameter. $CASE, $OBJECT, $SLOT and $VALUE are the context variables used. When triggered, the KS generates KSARs that reuse previous $VALUE of $SLOT from $CASE for $OBJECT (e.g., reuse.A36.material.from.case.2.for.column.1). The precondition is that "Design $SLOT of $OBJECT" is one of the current goals on the blackboard and the solution similarity rating of $CASE must be greater than 40. The action of this KS is to put $VALUE as $SLOT value of $OBJECT on the solution blackboard.

- **REUSE.PREVIOUS.DESIGN.OBJECT** reuses the previous solution of a similar design at the object level. The properties of a whole design object are taken from a case. Using the base plate anchor bolt design as an example, this KS can design the whole bolt group object (which includes bolt size, bolt spacing, bolt distance and number of bolts) based on an old design. The solution reuse is done by transferring the whole object, not just one parameter of the object. This KS represents another level of flexible solution reuse between of the capabilities of the two previously introduced KSs (REUSE.WHOLE.SOLUTION and REUSE.PARTIAL.SOLUTION).

  The trigger conditions of the KS are the completion of problem input, the presence of retrieved designs on the blackboard, and the existence of at least one undesigned object. $CASE, $OBJECT, $SLOT-VALUE-PAIR-LIST are the context variables used. When triggered, the KS generates KSARs that reuse the previous $OBJECT from $CASE (e.g., reuse.column.1.from.case.2) and binds $SLOT-VALUE-PAIR-LIST to all the attribute values of $OBJECT. The precondition is that "Design $OBJECT" is one of the current goals on the blackboard and the solution similarity rating of $CASE must be greater than 40.

The action of this KS is to put all the attribute values in $SLOT-VALUE-PAIR-LIST of $OBJECT on the solution blackboard.

- **REUSE.PREVIOUS.DESIGN** transfers the previous design history of a similar case into the current solution blackboard. In addition to the final solution, the intermediate analysis and calculation propositions of all alternatives (including both final and abandoned results) are also transferred to the blackboard. Therefore, DDIS can reuse the whole previous design history to skip unnecessary design calculations, constraint confirmations, etc., which have already been done in the past.

  This KS works basically the same as REUSE.WHOLE.SOLUTION with the addition of transferring over the previous design history. The trigger conditions of the KS are the same as REUSE.WHOLE.SOLUTION's. Context variables $CASE and $SOLUTION are used to generate one KSAR for each retrieved design and to bind $SOLUTION to its previous solution. The precondition is that the solution similarity rating of the retrieved design must be greater than 40. The execution of the KS asserts previous $SOLUTION from $CASE as the trial solution of the new design and recreates all the propositions in the previous design history stored with the case.

#### 4.5.2.3.2. Plan Transformer

The function of the plan transformer knowledge module is to transfer previous design plans to new designs. A design plan in DDIS is based on top-down decomposition. Several levels of plan abstraction are possible. In order to transfer plans at different levels of abstraction whenever appropriate, the plan transformer needs to work with the goal expansion mechanism of DDIS (discussed in details in Section 4.3.4.). The plan transformer of DDIS has one generic knowledge source REUSE.PREVIOUS.PLAN.

- **REUSE.PREVIOUS.PLAN** is both a case-dependent KS and a control KS because it transfers case-dependent knowledge (plans) from retrieved designs to the control blackboard. Global design plans, subplans and redesign plans can all be transferred when they are applicable. REUSE.PREVIOUS.PLAN does not preform an explicit plan adaptation task because the plan and goal maintenance mechanism of DDIS ensures the applicability of transferred plans. Plan and goal maintenance is discussed in details in Section 4.3.3.

  REUSE.PREVIOUS.PLAN is triggered when retrieved designs have design plans or redesign plans to expand active goals on the blackboard. Context variables $EXPANDABLE-GOAL, $PLAN and $CASE are used to generate one KSAR for each design plan of the retrieved designs that can provide more detailed steps of achieving the expandable design goals on the blackboard. The precondition is that the plan similarity rating of the retrieved $CASE must be

greater than 25 (a similarity threshold for case-dependent plan transformation) and $PLAN is not already on the blackboard. The execution of the KS adds $PLAN on the control blackboard as an expansion of $EXPANDABLE-GOAL. Goal expansion is discussed in details in Section 4.3.4, and an example of the use of REUSE.PREVIOUS.PLAN is given there.

### 4.5.3. Case Recorder

The case recorder knowledge module is the recording mechanism of DDIS. Its function is to capture case-dependent knowledge from design cases produced by DDIS itself. SAVE.DESIGN.SESSION is the generic KS in this module.

- **SAVE.DESIGN.SESSION** saves the final solution, intermediate propositions and control strategies of the design in the case memory knowledge base at the end of a DDIS design session. The solutions and propositions are recorded on the blackboard and can be easily saved in forms that the case-based reasoners can recognize. However, the design strategies do not explicitly exist on the blackboard. The KS processes the design history and abstracts one design plan along with several redesign plans to be stored with the case as its case-dependent plans. The plan extraction process is discussed in detail in Section 4.4.2 and 4.4.3.

  The trigger condition of the KS is the completion of the design session. No context variables are used for this KS. When the KS is triggered, only one KSAR is generated and it is always executable (precondition is true). The action of this KS is to make an entry in the case memory, analyze the design session, create the objects that need to be stored with the case (e.g., design plans, redesign plans, goals, etc.), and provide processed values for the attribute of the objects.

## 4.6. Implementation

At the beginning of the study, experimental prototypes were developed using KEE and BB1. The experience with the two systems allowed us to evaluate their feasibility as the implementation environment of the project and to gain more practical knowledge about certain techniques for transferring various types of case-dependent knowledge across designs. KEE [Kunz 84] is favored for its rich and flexible frame-based knowledge representation, object-oriented programming paradigm, data-driven reasoning, and truth maintenance system. On the other hand, the blackboard control architecture of BB1 [Hayes-Roth 84] is ideal for the integrated design paradigm. Consequently, the resulting solution was to use KEE as the underlying development tool and build our own blackboard system, which is a reimplementation of BB1 with specific modifications (e.g., solution

blackboard maintenance and plan and goal updating) to facilitate the integrated design model. The blackboard architecture of DDIS is not as sophisticated as BB1, but it supports cooperative problem solving by means of a global data structure that records the evolving solution and control plans contributed by different knowledge sources.

DDIS is implemented in IntelliCorp's KEE running on a Texas Instruments MicroExplorer installed in an Apple Macintosh II. KEE is a flexible and general-purpose commercial AI development system produced by IntelliCorp. It is a hybrid environment, which integrates frame-based knowledge representation, rule-based reasoning, data-driven inference and object-oriented programming. It also has interactive graphics and a truth maintenance system (TMS).

All the elements in DDIS, as described earlier, are all implemented as classes of objects using KEE's frame-based representation. However, the rule system in KEE is not used (except for a few TMS justifications created by rules). DDIS is mainly implemented using an object-oriented programming approach.

The blackboard reasoning paradigm is built in the KEE frame system utilizing its data-driven capabilities (implemented as active values) and its object-oriented programming facility. Every major blackboard component is represented as an object with attached methods that define its own procedural characteristics. The blackboard activities are sequences of data-driven message passing actions. For example, the plan and goal maintenance procedures described earlier are invoked when the values of the blackboard plan level (a slot of the control object) is changed, and the updating procedure is accomplished by sending several messages to the plans and goals involved.

KEEworlds and the TMS are used to implement the solution information blackboard. KEEworlds is a facility provided by KEE for modeling and exploring different hypothetical situations that might arise in knowledge bases. A world represents an alternative state of knowledge bases. Objects can have different attribute values in different worlds and can inherit values from their parent worlds. DDIS creates a new child world for every design cycle. The changes made on the solution information blackboard in that cycle are recorded in the world. Design contributions from previous cycles are not stored locally, but are inherited from the parent worlds. Therefore, the most recent world represents the latest state of the solution information blackboard, and its parent worlds record the whole design history.

KEE's Truth Maintenance System (TMS) is a set of facilities for establishing and maintaining dependencies between facts in the knowledge bases. The justifications created during design by the TMS procedures associated with data items are also maintained by KEE's TMS. Figure 4-12 shows the dependence graph of a variable during design. The dependencies play an important role in supporting backtracking, maintaining design consistency and propagating redesign modifications in DDIS. Although both KEEworlds

and TMS are used, DDIS does not support parallel design. Multiple partial designs cannot exist simultaneously; DDIS only follows one line of reasoning.

The user interface of DDIS uses the graphic facility provided by both Explorer and KEE. The input, output and blackboard interface are implemented in the windowing system of MicroExplorer and the ActiveImages package of KEE. Figure 4-13 shows the blackboard interface. All the case-dependent and case-independent executable actions and their rating are presented on the right, and design plans and their currently active goals are shown on the left.

The DDIS program consists of 690 KEE objects. In addition to the functions associated with the objects, there are approximately 80 Lisp functions (about 66 KB in size with comments). DDIS has 17 generic knowledge sources and 25 domain specific knowledge sources (for the two demonstration applications described in Chapter 5). The knowledge sources can generate temporary objects during design. Most of the knowledge sources are written with context variables, which enable multiple actions (KSARs) to be generated from a single knowledge source. The domain knowledge base is not complete. More design and backtracking knowledge sources can be added to the knowledge base. However, the domain knowledge sources are adequate for the illustrative purpose as they are used in this study.

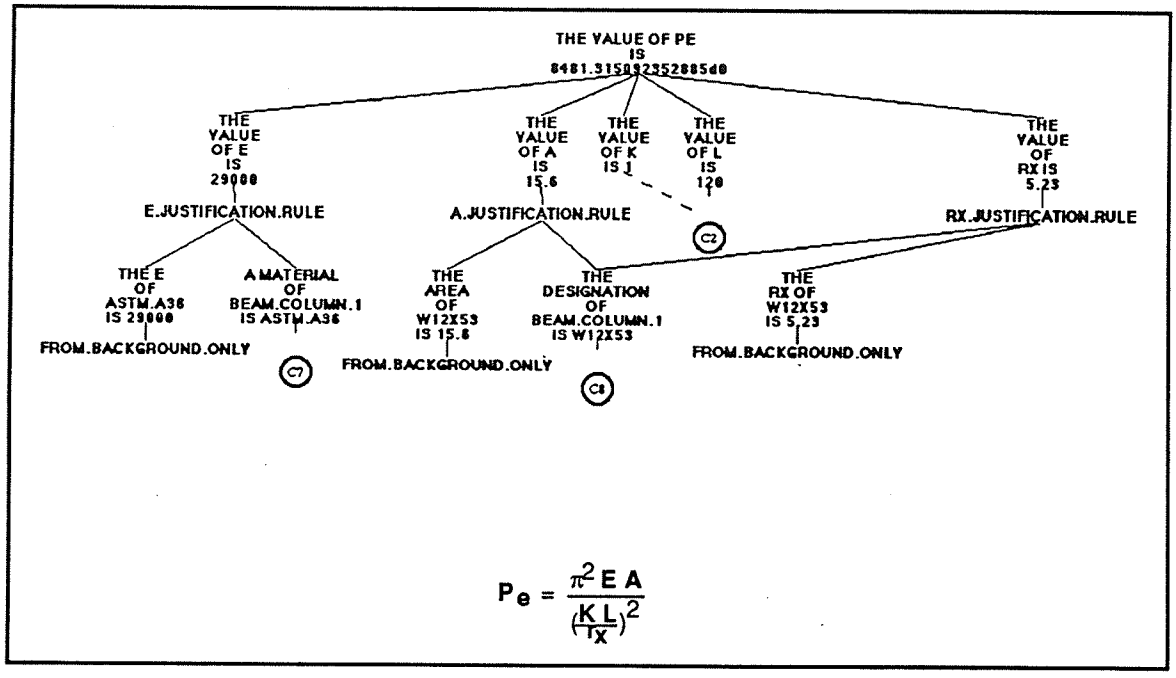



**Figure 4-12:** Variable Dependence Graph

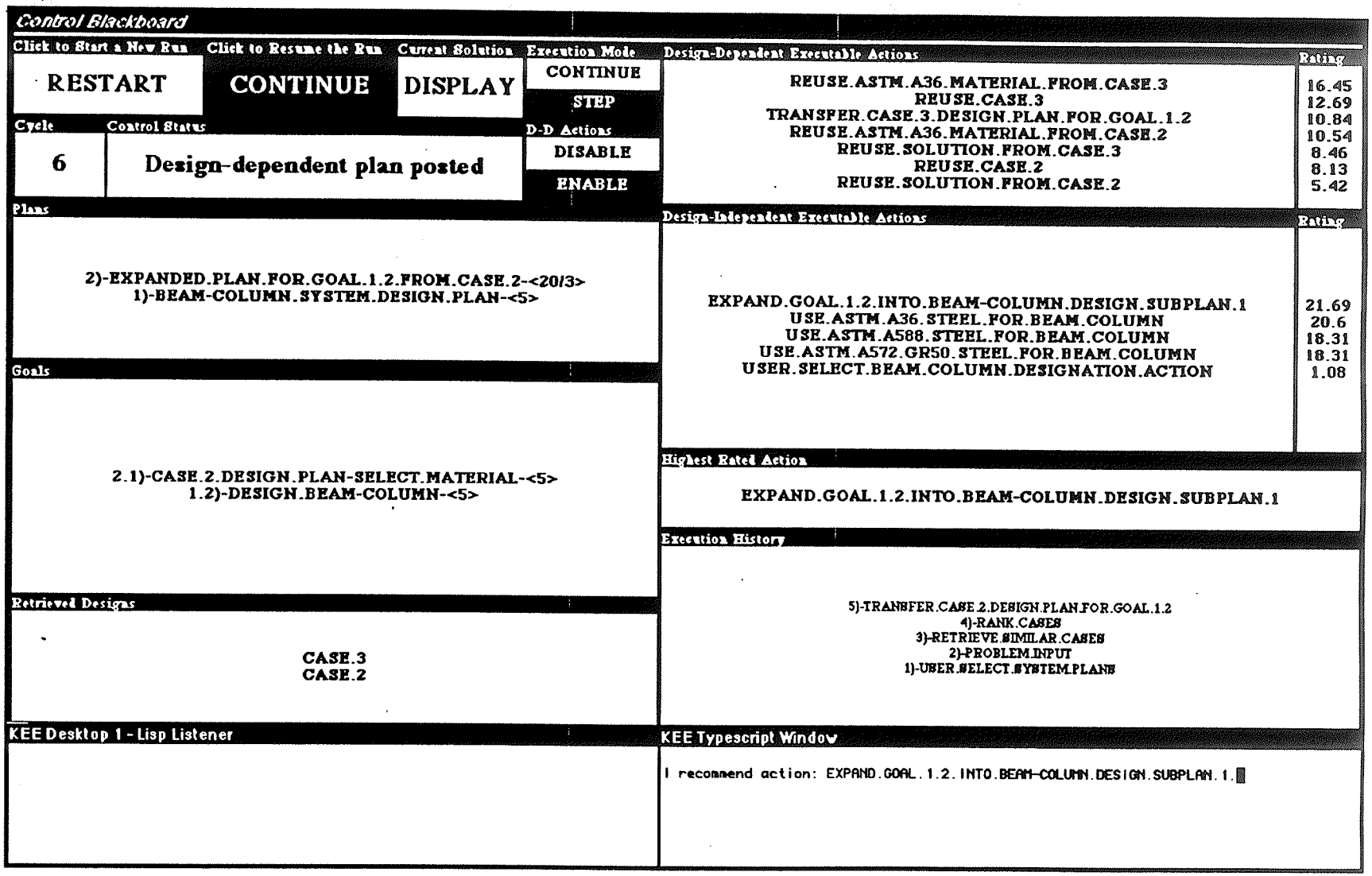



**Figure 4-13:.** The Blackboard Interface of DDIS

The computational speed of DDIS depends largely on the size of the problem. For example, DDIS is mush faster in solving the beam-column design problem (described in Section 5.1) than the base plate problem (described in Section 5.2). A typically 40 cycle run with the base plate problem (e.g., the example given in Section 5.2.2) takes about 20 minutes, and a 40 cycle run with the beam-column problem takes about 8 minutes. The truth maintenance system in KEE is relatively slow. Therefore, a large number of variables and constraints (as in the problem space of the base plate design) slows down the system dramatically. Speed was not a concern in the programming of the DDIS prototype. There is room for improving the performance of the blackboard architecture of DDIS.

# Chapter 5
# Illustrative Examples

A domain knowledge base for structural steel beam-column design was built as the initial test bed for the DDIS prototype. This problem served as a valuable vehicle for learning about the concepts, formalisms, techniques and devices needed to integrate case-dependent and case-independent knowledge in knowledge-based design systems. The simplicity of the problem made the prototyping faster and the debugging easier. It is also used as an example to easily communicate with people about DDIS's ideal for integrating case-dependent and case-independent reasoning. To provide a basis for further testing and refinement of the prototype, the design of anchor base plates for electrical transmission poles was implemented later. This problem allowed us to extend the architecture of DDIS for more case-based design ability.

To demonstrate the integrated design approach of DDIS, this chapter describes the two domain knowledge bases and gives several illustrative examples with comments. The two design examples from the beam-column problem explain the basic problem-solving behavior of DDIS and the different uses of case-based design methods in DDIS (including solution reuses and plan reuses). The base plate design examples show how DDIS captures design plans from a user and reuses them in a new design session.

## 5.1. Beam-Column Design

The beam-column design problem is shown in Figure 5-1. The column is restrained only at the ends and is subjected to an axial compression load and two end moments. The design task is to find an efficient wide-flange structural section and the steel material for the given load using the AISC plastic design method [AISC 80]. Figure 5-2 shows the beam-column design knowledge base of DDIS.

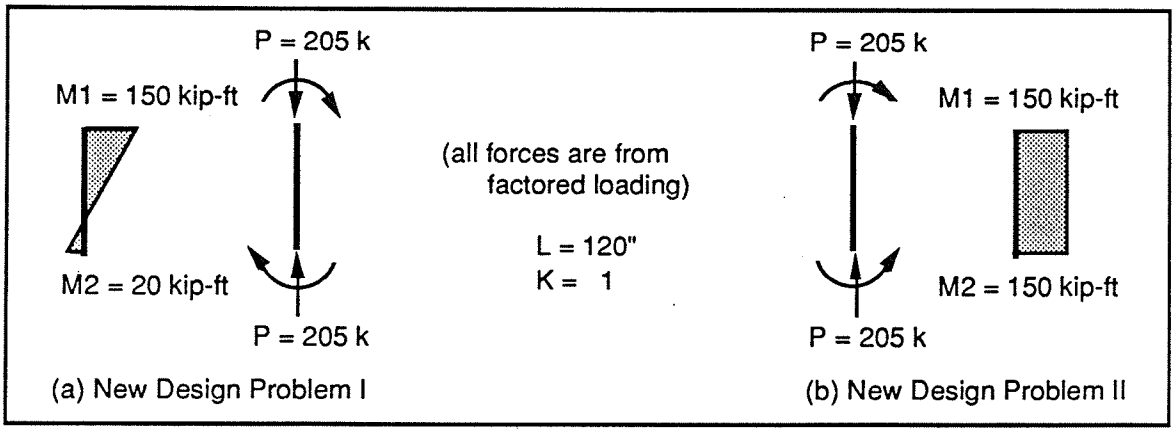



**Figure 5-1:** Beam-Column Design Examples

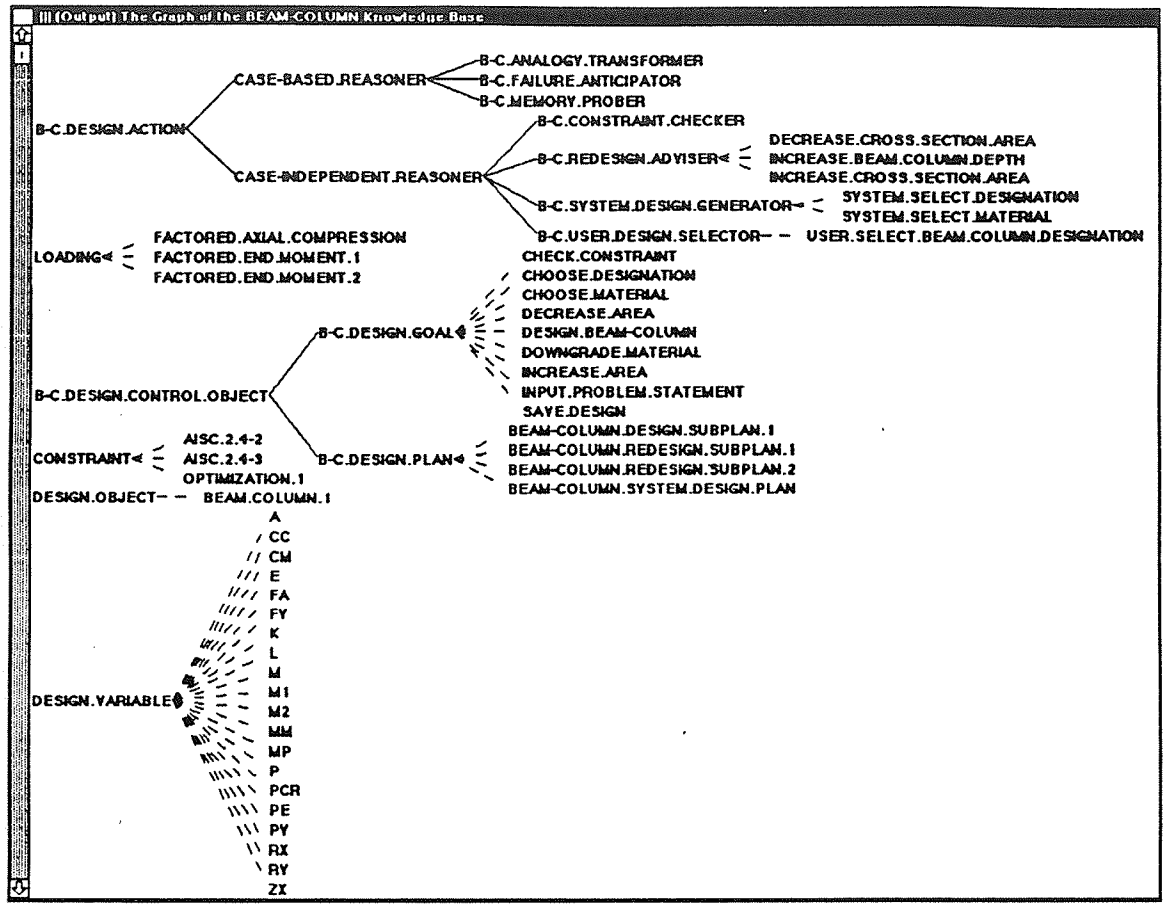



**Figure 5-2:** The Beam-Column Knowledge Base

The design object of the problem is BEAM.COLUMN.1, which is an instance of the W.SHAPE parts in the concept knowledge base. The problem inputs are P (factored axial loading in kips), K (effective length factor), L (length of the beam column in inches), M1 and M2 (factored bending moment at two ends in kip-ft). The design constraints in the knowledge base are the two interaction equations from AISC and an optimization constraint:

$$\frac{P}{P_y} + \frac{M}{1.18\,M_p} \le 1 \qquad \text{AISC Formula (2.4-3)}$$

$$\frac{P}{P_{cr}} + \frac{C_m\ M}{(1 - P/P_e)\,M_m} \le 1 \qquad \text{AISC Formula (2.4-2)}$$

$$\text{Max}\left[\left(\frac{P}{P_{cr}} + \frac{C_m\ M}{(1 - P/P_e)\,M_m}\right), \left(\frac{P}{P_y} + \frac{M}{1.18\,M_p}\right)\right] \ge 0.9$$

in which

| | |
|---|---|
| P | factored axial compression load. |
| M | factored primary bending moment. |
| $P_{cr}$ | ultimate strength of an axially loaded compression member. |
| $M_p$ | plastic moment. |
| $M_m$ | maximum resisting moment in the absence of axial load. |
| $P_e$ | Euler buckling load. |
| $P_y$ | plastic axial load. |
| $C_m$ | column curvature factor. |

The above mentioned variables are all represented as design variable objects in the beam-column knowledge base instantiated from appropriate type of data item. There are three domain specific design generators (SYSTEM.SELECT.DESIGNATION, SYSTEM.SELECT.MATERIAL and USER.SELECT.BEAM-COLUMN.DESIGNATION) and three domain specific redesign advisers (INCREASE.CROSS.SECTION.AREA, INCREASE.BEAM-COLUMN.DEPTH and DECREASE.CROSS.SECTION.AREA) in the beam-column knowledge base.

### 5.1.1. Beam-Column Design Session I

At the beginning of the run, the case memory has three design cases as shown in Figure 5-3 and Figure 4-11. Figure 5-1 (a) illustrates the new problem, and Table 5-1 summarizes the design session. The next section describes the DDIS problem solving session step by step.

| Cycle | Action | Cycle | Action |
|---|---|---|---|
| 1 | state problem | 9 | check critical constraint OPT.1 * |
| 2 | input | *10* | *import redesign plan from CASE.2* |
| *3* | *retrieve similar designs* | 11 | redesign beam-column designation |
| *4* | *rank CASE.2 and CASE.3* | 12 | decrease cross-sectional area |
| *5* | *reuse case-based plan from CASE.2* | 13 | recheck OPTIMIZATION.1 const. |
| 6 | expand goal to system heuristic plan | 14 - 15 | check remaining constraints |
| 7 | select beam-column material | 16 | end design session |
| *8* | *reuse designation from CASE.3* | 17 | save design |

**Table 5-1:** Action Overview of Beam-Column Design Session I
(Case-dependent actions are in italics and
constraint violations are marked with asterisks)

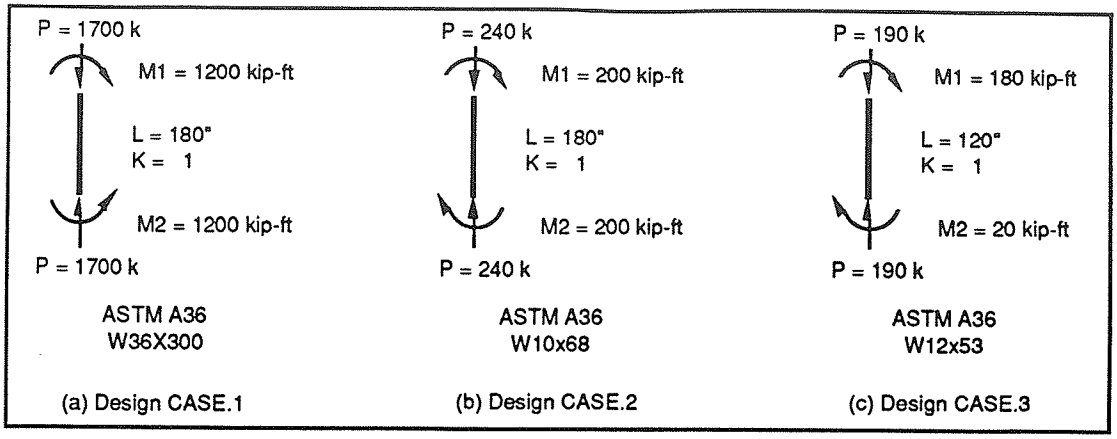



**Figure 5-3:** Design Cases in the Case-Memory Knowledge Base

### 5.1.1.1. DESCRIPTION OF THE DESIGN SESSION

***Cycle 1—state problem.*** The design plan BEAM-COLUMN.SYSTEM.DESIGN.PLAN is posted on the blackboard as the global control strategy of the design session. The plan has three sequential goals: INPUT.PROBLEM.STATEMENT, DESIGN.BEAM-COLUMN and SAVE.DESIGN.

***Cycle 2—input.*** The beam-column problem input action is executed. All the values of the variables that need to be provided by the user are entered through a pop-up window.

***Cycle 3—retrieve similar designs.*** Once the new problem inputs have been entered, the global plan moves to its second goal—DESIGN.BEAM-COLUMN, and the RETRIEVE.SIMILAR.CASES action is recommended. The user decides to retrieve CASE.3 and CASE.2 because the loading condition of CASE.3 is closer to the new requirement and the design plan of CASE.2 has fewer redesign cycles compared to CASE.3.

***Cycle 4—rank cases.*** Based on the above justification, the user assigns 50 and 80 respectively to the solution similarity and plan similarity of CASE.2. On the other hand, the solution similarity and plan similarity of CASE.3 are ranked 78 and 60, respectively. Note that the similarity ratings are provided by the user based on his/her judgement on the usefulness of the previous design solutions and plans in the current design.

***Cycle 5—reuse case-dependent plan.*** The retrieval of CASE.2 and CASE.3 triggers several case-dependent actions at cycle 5. However, the two retrieved designs are not similar enough that their solutions can be reused as a whole. The ratings of the two REUSE.WHOLE.SOLUTION actions are low. Therefore, DDIS recommends the reuse of the design plan of CASE.2.

***Cycle 6—expand goal to case-independent plan.*** Figure 4-13 shows the blackboard state of this cycle. All the executable actions and their rating are presented on

the right, and two plans (one case-independent and one case-dependent) and their currently active goals are shown on the left. The DESIGN.BEAM-COLUMN goal was expanded into CASE.2.DESIGN.PLAN at cycle 5. However, the case-dependent plan is not considered good enough to lead the design session due to the moderate plan similarity ratings of CASE.2 (the evaluations of the three highest rated KSARs are shown in the KSAR rating examples in Section 4.3.2.1, 4.3.2.2 and 4.3.5). Therefore, the case-independent BEAM-COLUMN.DESIGN.SUBPLAN.1 becomes the second subplan of DESIGN.BEAM-COLUMN.

***Cycle 7—select material.*** Using the two plans, two conflicting goals are on the blackboard: CHOOSE.DESIGNATION and CASE.2.DESIGN.PLAN-SELECT.MATERIAL. Since the case-dependent plan has a higher weight (i.e., 20/3 vs. 5), DDIS follows the case-dependent plan from CASE.2 and executes the case-independent USE.ASTM.A36.STEEL action.

***Cycle 8—reuse partial solution.*** The execution of the action satisfies the case-dependent goal and invokes the next goal in CASE.2.DESIGN.PLAN, which is CASE.2.DESIGN.PLAN-SELECT.DESIGNATION. Now the two plans agree on selecting the designation for the beam-column. There are three executable actions on the blackboard for this task, one case-independent action that estimates the required section based on heuristic rules and two case-dependent actions to reuse the previously used section in CASE.2 and CASE.3. In this case, DDIS chooses the highest rated action, which is REUSE.W12X53.DESIGNATION.FROM.CASE.3.

***Cycle 9—check critical constraint.*** The new blackboard state triggers DDIS's constraint checker. The order of the constraint checking is influenced by the case-dependent CASE.2.DESIGN.PLAN-CHECK.CONSTRAINT.OPTIMIZATION.1 goal. The constraint OPTIMIZATION.1 is checked first because it was the more critical one in CASE.2, and it is found unsatisfied. Therefore, a FIX.CONSTRAINT.OPTIMIZATION.1.VIOLATION goal is posted on the blackboard by DDIS.

***Cycle 10—employ previous redesign plan.*** The constraint violation triggers two backtracking provoker actions as well as several goal expansion actions. However, the two backtracking actions, REDESIGN.BEAM-COLUMN.1.DESIGNATION and REDESIGN.BEAM-COLUMN.1.MATERIAL, are ranked the same. To modify the design intelligently, DDIS first finds the redesign plan in CASE.2 that had corrected the same constraint violation before and then posts it on the blackboard to guide the redesign process.

***Cycle 11—invoke backtracking.*** The designation of the beam-column should be redesigned according to the first goal of the redesign plan from CASE.2. Therefore, DDIS executes the highest rated REDESIGN.BEAM.COLUMN.1.DESIGNATION backtracking provoker, which removes the W12X53 designation from the design information blackboard.

*Cycle 12—decrease cross-sectional area.* The removal of the value of the beam-column's designation invokes the next goal in the redesign plan (i.e., DECREASE.CROSS.SECTION.AREA) and reactivates the CHOOSE.DESIGNATION goal in the two still-active design subplans. Although all goals participate in the rating, the previous redesign plan has a much higher weight to focus the current task on the redesign. By following the old plan again, DDIS decreases the cross-sectional area of the beam-column and uses a W14X48 section.

*Cycle 13—recheck constraint.* The modified design of the beam-column is an ASTM A36 W14X48 wide-flange structural steel. The modified design is checked against the previously violated OPTIMIZATION.1 constraint, and the constraint is satisfied this time.

*Cycle 14 to 15—check remaining constraints.* Then, the other two constraints are checked, and they are satisfied.

*Cycle 16—end design session.* Since all the constraints are satisfied, the design is completed at cycle 16.

*Cycle 17—save design.* DDIS saves the design in the memory as CASE.4 at the end.

#### 5.1.1.2. ANALYSIS OF THE DESIGN SESSION

This example showed that DDIS is capable of utilizing more than one design case and combining different design approaches. Two cases were retrieved. One was for reusing its solution and the other was for reusing its design and redesign plans. The two design values for material and designation were generated by case-independent design generator and case-dependent solution transformer, respectively. Goal expansions were also used in three occasions (cycle 5, 6 and 10) to provide DDIS with more detailed design plans. The way DDIS works with multiple goals was also described.

### 5.1.2. Beam-Column Design Session II

To give another example, consider the beam-column problem shown in Figure 5-1 (b). The only difference between this problem and the previous example is that the end moment at bottom is increased to 150 kip-ft counterclockwise. Table 5-2 summarizes the design session. Note that the case-memory knowledge base now has an additional case (i.e., CASE.4 ) just saved after the previous session.

| Cycle | Action | Cycle | Action |
|---|---|---|---|
| 1 - 2 | input problem | *11* | *employ CASE.1 redesign plan* |
| *3* | *retrieve similar design—CASE.4* | 12 | redesign beam-column designation |
| *4* | *rank CASE.4* | 13 | increase cross-sectional area |
| *5* | *reuse case-based plan from CASE.4* | 14 | recheck AISC 2.4-2 constraint * |
| *6* | *import CASE.4* | 15 -17 | repeat design modification steps |
| 7 | check OPTIMIZATION constraints | *18* | *reuse case-based plan from CASE.1* |
| 8 | check constraints AISC 2.4-2 * | 19 - 20 | check remaining constraints |
| *9* | *search memory for redesign plans* | 21 | end design session |
| *10* | *rank CASE.1* | | |

**Table 5-2:** Action Summary of Beam-Column Design Session II
(Case-dependent actions are in italics and
constraint violations are marked with asterisks)

### 5.1.2.1. DESCRIPTION OF THE DESIGN SESSION

***Cycle 1 to 2—input problem.*** Once again the design starts by selecting the global BEAM-COLUMN.SYSTEM.DESIGN.PLAN and by entering the problem input variables.

***Cycle 3—retrieve similar design.*** CASE.4 is retrieved this time because of its very close loading resemblance to the new problem.

***Cycle 4—rank case.*** 90 and 80 are assigned to CASE.4's solution similarity and plan similarity rating, respectively.

***Cycle 5—reuse case-dependent plan.*** Because of the lack of detailed design plans on the control blackboard and the high plan similarity rating of CASE.4, DDIS suggests the reuse of the design plan from CASE.4.

***Cycle 6—import previous design.*** The close similarity of the loadings in CASE.4 and the new problem make the previous design solution an excellent initial trial for the new design. Therefore, the REUSE.CASE.4 action is highest rated and is executed. The execution of the action posts the previous solution (i.e., ASTM A36 W14X48) on the blackboard and loads in the previously saved world of CASE.4, which contains the intermediate solutions and propositions involved in the old design.

***Cycle 7 to 8—check constraints.*** The new blackboard state triggers DDIS's constraint checker. Only the coefficient $C_m$ changes during the calculation of the new constraint values, so AISC formula 2.4-3 is still satisfied. Therefore, DDIS only needs to

recheck stability according to AISC formula 2.4-2 and efficiency according to the OPTIMIZATION.1 constraint. That is why the check AISC 2.4-3 constraint action is not shown as one of the executable actions on the blackboard. The OPTIMIZATION.1 constraint is satisfied. However, the AISC 2.4-2 constraint is not satisfied. Therefore, the goal FIX.CONSTRAINT.AISC.2.4-2.VIOLATION is posted on the blackboard by DDIS.

***Cycle 9—search case memory for redesign plans.*** The constraint violation triggers two backtracking provoker actions. However, the current control knowledge on the blackboard cannot differentiate which one is better. Therefore, redesign plans are needed for this constraint violation. Since neither heuristic redesign plans nor case-based redesign plans are available from the knowledge base and from CASE.4, DDIS searches the case memory knowledge base again to find cases that have experienced the same constraint violation before. CASE.1 is found and retrieved for guiding the design modification. Note that replacing CASE.4 with CASE.1 on the retrieved design level of the control blackboard also removes all the case-dependent plans and goals associated with CASE.4.

***Cycle 10—rank newly retrieved design.*** CASE.1 is retrieved only for its redesign plan. Its solution should not be considered because of its very different loading condition. Therefore, 35 and 80 are assigned to its solution similarity and plan similarity rating, respectively. CASE.1 could not be judged as a similar case if only the surface similarities between the two problems were compared. This illustrates how DDIS's case memory can be searched from a different angle.

***Cycle 11—employ previous redesign plan.*** The previous redesign plan for fixing the AISC 2.4-2 constraint violation is reused to control the backtracking point.

***Cycle 12—invoke backtracking.*** The designation of the beam-column should be redesigned according to the first goal of the old redesign plan. Therefore, DDIS executes the highest rated REDESIGN.BEAM.COLUMN.1.DESIGNATION backtracking provoker, which removes the W14X48 designation from the design information blackboard.

***Cycle 13—increase cross-sectional area.*** The removal of the value of the beam-column's designation invokes the next goal in the redesign plan (i.e., INCREASE.CROSS.SECTION.AREA). DDIS increases the cross-sectional area of the beam-column and uses a W14X53 section.

***Cycle 14—recheck constraints.*** AISC 2.4-2 is checked again and found still unsatisfied.

***Cycle 15 to 17—repeat redesign steps.*** The W14X53 section is still underdesigned because the constraint is still violated. The CASE.1.REDESIGN.PLAN.1 is reused again to start another round of design modification. DDIS repeats the redesign steps in cycle 12, 13, and 14 again and settles down with a W12X58 section that satisfies the AISC 2.4-2 constraint.

*Cycle 18— post case-dependent plan.* The purpose of CASE.1.REDESIGN.PLAN.1 and FIX.CONSTRAINT.AISC.2.4-2.VIOLATION is achieved because of the success of the redesign. DDIS's goal and plan updating mechanism removes them from the control blackboard. With no specific plans for the rest of the design session, DDIS refers to the past design plan of CASE.1 to guide its actions.

***Cycle 19 to 20—check remaining constraints.*** The other two constraints are rechecked, and they are satisfied.

***Cycle 21—end design session.*** Since all the constraints are satisfied, the design is completed at cycle 21. The final solution of the design is an ASTM A36 W12X58 wide-flange structural steel.

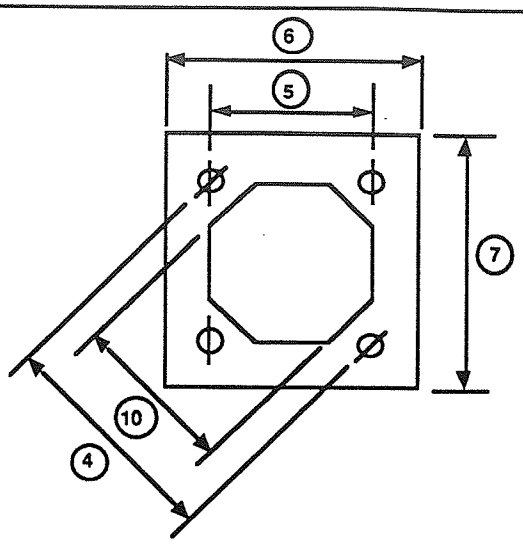
#### 5.1.2.2. ANALYSIS OF THE DESIGN SESSION

This example showed that DDIS is capable of reusing not only previous design solutions, but also the propositions in an old design (e. g., intermediate design solutions, calculations and variable dependences). Steps with known results can be skipped. At cycle 9, the case memory was searched for redesign plans to fix AISC 2.4-2 violation. This illustrates the flexibility of DDIS in terms of changing the source of case-dependent knowledge and searching the memory from a different perspective.

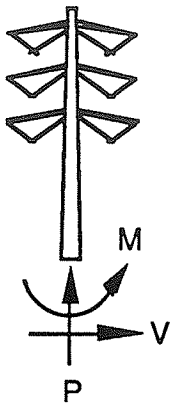
## 5.2. Pole Anchor Base Plate Design

The second demonstration application developed with DDIS is anchor base plate design for electrical transmission pole. Douglas Phan provided this study with necessary design documentation and expertise. Pole anchor steel base plate design is a design problem involving many variables and general control knowledge. Figure 5-4 shows a typical 4-bolt pattern base plate and its parameters. The design process consists of three major design tasks: weld design, bolt design and plate design. Therefore, the base plate design task may be considered as a subsystem design problem with interaction between the bolts and the plate as well as the plate and the weld.

The inputs of the base plate design problem (see Figure 5-4) are the end reactions at the base of the pole, which include P (axial load), V (resultant shear) and M (resultant moment), and the geometry of the pole base, which includes POLE.BASE.DIA (flat-to-flat diameter of the pole base shaft section) and POLE.SIDES (number of sides of the pole base shaft). Because the implementation focuses on designing the plate and the bolts, the weld information (the type and size of the weld) is also treated as input. The design variables (see Figure 5-4) include the plate dimensions (length, width, and thickness) and the bolt configuration (bolt pattern, number, size, and distance). The design objects on the design information blackboard that collect all these values are: BASE.PLATE.1, BOLT.1 and




Top View of a 4-Bolt Base Plate




Transmission Pole

| | | |
|---|---|---|
| 1. | Bolt Pattern: | 4-bolt |
| 2. | Number of Bolts per Quadrant: | 1 |
| 3. | Bolt Diameter: | Diameter of the selected bolt |
| 4. | Bolt Distance: | Diagonal Bolt Spacing Distance |
| 5. | Horizontal Bolt Distance: | Horizontal Bolt Spacing Distance |
| 6. | Plate Length: | |
| 7. | Plate Width: | |
| 8. | Plate Thickness: | Not shown in this view |
| 9. | Pole Base Shape: | Octagonal |
| 10. | Pole Base Diameter: | Flat-to-flat dimension of pole base |

**Figure 5-4:** Design Variables of a 4-Bolt Base Plate

BOLT.GROUP.1. The base plate design knowledge base (shown in Figure 5-5) has eleven design constraints in three categories: strength constraints, geometric constraints and user specified constraints.

The strength constraints are:

$$\frac{P_{max} + V_{max}}{P_{all}} \leq 1.0$$ Bolt Interaction Equation

$$\frac{Diam_{plate} - Dist_{bolt}}{2} \geq \frac{V_{max}}{0.75 \times F_u \times t}$$ Minimum Edge Distance

$$S \geq \frac{V_{max}}{0.75 \times F_u \times t} + \frac{Diam_{bolt}}{2}$$ Minimum Bolt Spacing II

$$\frac{6 \times d_{eff} \times P_{max}}{b.single_{eff} \times t^2} \leq 0.66\ F_y$$ Plate Bending Single Bolt

$$\frac{6 \times d_{eff} \times \sum P_{actural}}{b.group_{eff} \times t^2} \leq 0.66\ F_y$$ Plate Bending Bolt Group

The geometric constraints are:

$$Dist_{bolt} \leq (Diam_{pole} + Weld.Size \times 2 + Nut.Width + 1)$$ Minimum Bolt Circle Distance

$$S \geq 3 \times Diam_{bolt}$$ Minimum Bolt Spacing

$$\frac{\pi \times Dist_{bolt}}{4} \geq S \times NOB_{quad}$$ Max. Number of Bolts

$$\frac{Diam_{plate} - Dist_{bolt}}{2} \geq$$ the applicable minimum edge distance in AISC Table 1.16.5.1

The user specified constraints are:

$$\frac{P_{max} + V_{max}}{P_{all}} \geq 0.9$$ Bolt Optimum Unity Factor

$$Dist_{bolt} \leq (Diam_{pole} + Weld.Size \times 2 + Nut.Width + 1) + 6$$ Maximum Bolt Distance

in which

| | |
|---|---|
| $P_{max}$ | maximum axial bolt load. |
| $P_{all}$ | Bolt allowable load. |
| $P_{actural}$ | axial load on individual bolt. |
| $V_{max}$ | maximum shear bolt load. |
| $NOB_{quad}$ | number of bolts per quadrant. |
| $Dist_{bolt}$ | bolt circle distance. |
| $Diam_{bolt}$ | diameter of anchor bolt. |
| S | bolt spacing. |
| Nut.Width | width of bolt nut. |
| $Diam_{plate}$ | diameter of base plate. |
| t | thickness of base plate. |
| $d_{eff}$ | effective bending distance of base plate. |
| $b.single_{eff}$ | effective width of base plate for single-bolt bending. |
| $b.group_{eff}$ | effective width of base plate for multiple-bolt bending. |
| $F_y$ | specified minimum yield strength of base plate. |
| $F_u$ | specified minimum tensile strength of base plate. |
| $Diam_{pole}$ | point-to-point diameter of the pole base shaft. |
| Weld.Size | width of pole base weld. |

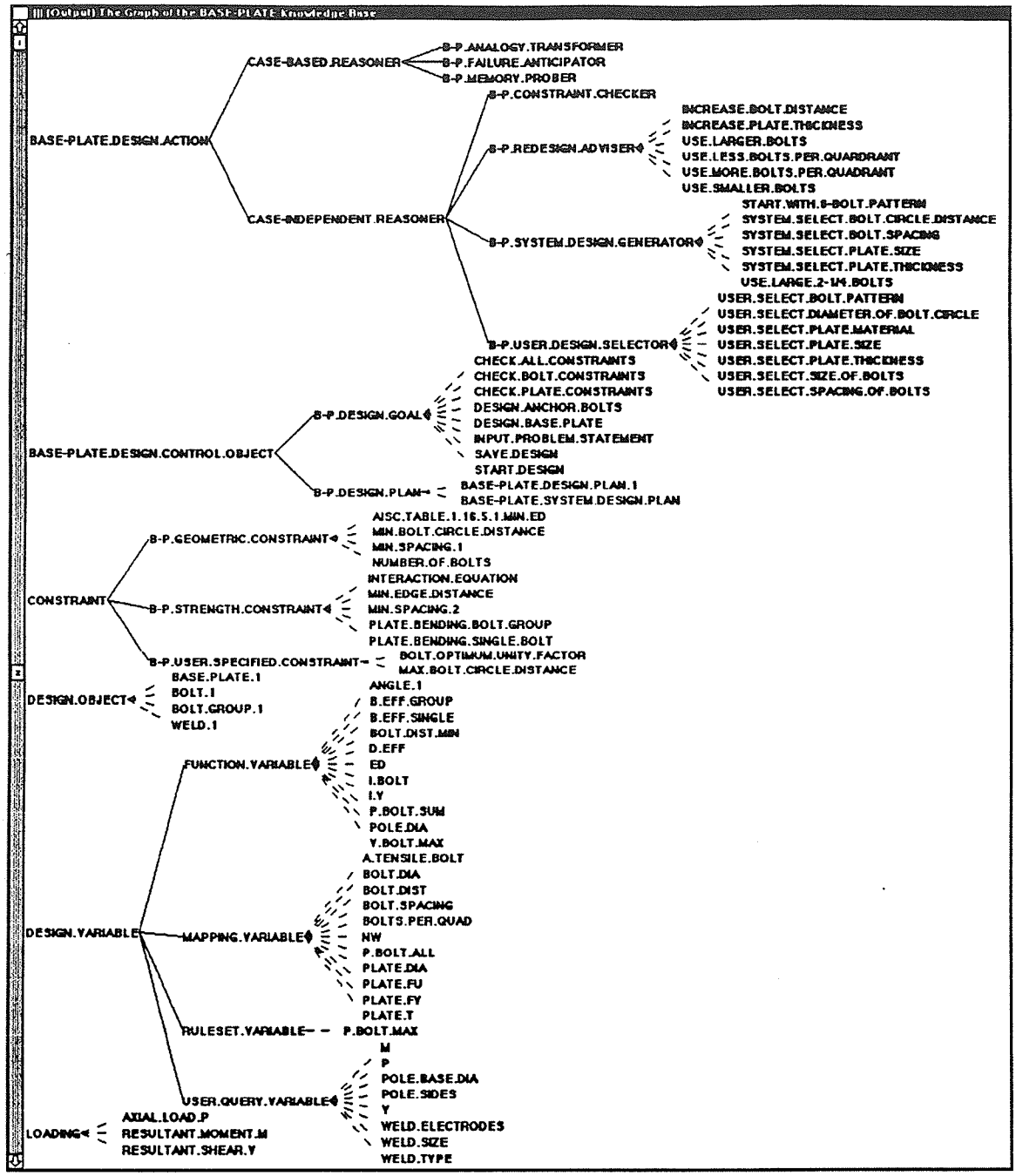



**Figure 5-5:** The Base Plate Knowledge Base

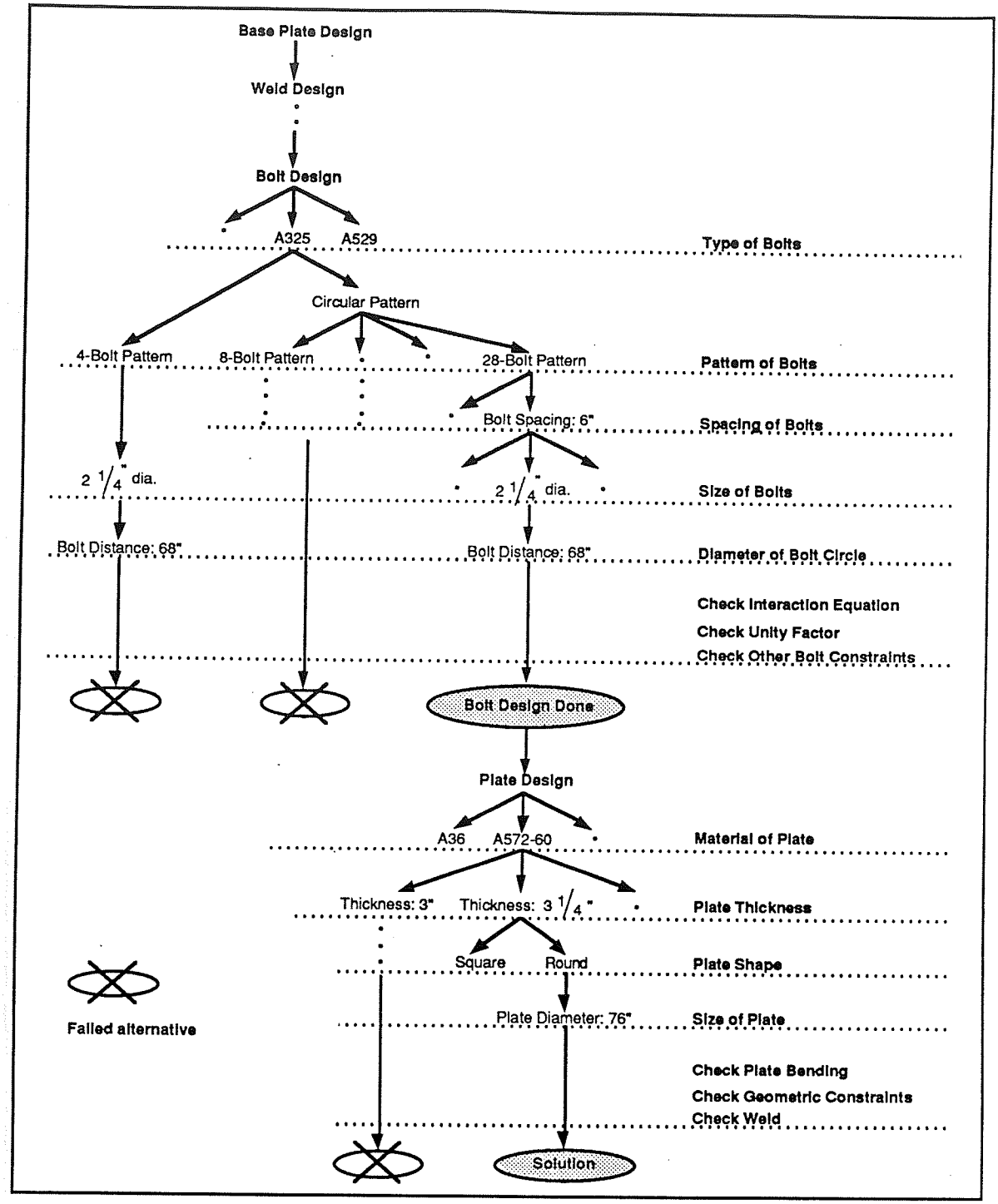



**Figure 5-6:** A Solution Path of the Base Plate Design Problem

The current knowledge base contains thirteen design generators and six redesign advisers representing the domain-specific actions in anchor base plate design. The design task is not a fixed procedure (see Figure 5-6 for one example of the possible solution paths). Because of the interaction between design steps, evaluation and subjective decision-making are included in the design cycle. Decisions have to be made to construct design plans to minimize search and guide the design process. A good design should achieve compact plate size, small plate thickness and feasible bolt distance. In addition to those considerations, local availability (as equipment and expertise), and preferences may also influence the design. Frequently, there is more than one configuration that is acceptable. Therefore, base plate design needs much more high-level guidance during the design process than does the design of simple beam-columns, which makes the base plate design problem a very interesting and feasible problem domain to investigate how to apply case-based design strategy.

## 5.2.1. Base Plate Design Example I

This example demonstrates how DDIS acquires design strategies from the user, and the next example reuses them in a new design. The two design problems are illustrated in Figure 5-7. The sample problem is given to DDIS first. The case-dependent actions of DDIS are turned off, and an experienced designer helps the design. When necessary, the designer overides DDIS's decisions to guide the design to his/her prefer path. The design actions at each cycle are listed in Table 5-3, and the design is summarized in the following section.

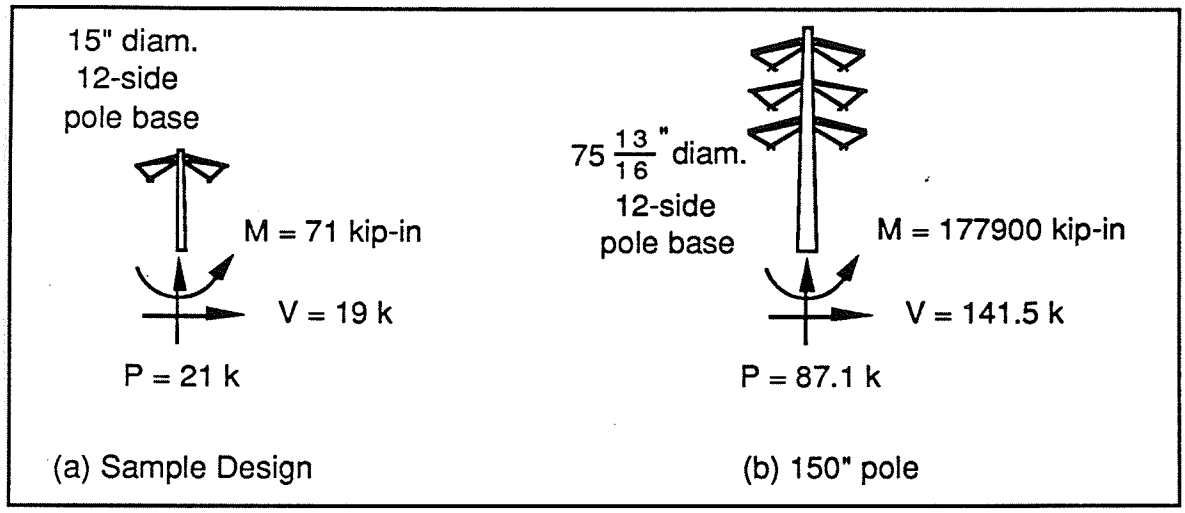



**Figure 5-7:** Anchor Base Plate Design Examples

| Cycle | Action | Cycle | Action |
|---|---|---|---|
| 1 - 3 | input problem & expand global goal | 27 | ASTM.A572.GR60 plate selected |
| 4 | user specifies major constraints | 28 | use 21.25" diam. plate |
| 5 | use 2-1/4" bolts | 29 | use 0.5" thick plate |
| 6 | start with 8-bolt pattern | 30 | check single-bolt plate bending |
| 7 | use 24" bolt distance | 31 | check multiple-bolt plate bending |
| 8 | use 6.75" bolt spacing | 32 - 37 | check other constraints |
| 9 | check interaction.equation | 38 | check MIN.BOLT.CIRCLE.DIST * |
| 10 | check BOLT.OPTIMUM.UNITY * | 39 | redesign bolt distance |
| 11 | redesign bolt diameter | 40 | 19.5" bolt distance selected by user |
| 12 | select 1/2" diam. bolts by user | 41 | check MIN.BOLT.CIRCLE.DIST |
| 13 | check BOLT.OPTIMUM.UNITY * | 42 | recheck INTERACTION.EQUATION |
| 14 | redesign bolt pattern | 43 | recheck single-bolt plate bending * |
| 15 | use fewer bolts per quadrant (4-bolt) | 44 | redesign plate thickness |
| 16 | check BOLT.OPTIMUM.UNITY | 45 | increase plate thickness to 0.75" |
| 17 | recheck INTERACTION.EQUATION * | 46 | check single-bolt plate bending |
| 18 | redesign bolt diameter | 47 | check multiple-bolt plate bending |
| 19 | use larger bolts (5/8" diam.) | 48 - 51 | check remaining constraints |
| 20 | check INTERACTION.EQ | 52 | check AISC min. edge dist. table * |
| 21 | check BOLT.OPTIMUM.UNITY * | 53 | redesign plate diameter |
| 22 | redesign bolt distance | 54 | use 21.75" diam. plate |
| 23 | use 19" bolt distance | 55 | check AISC min. edge dist. table |
| 24 | check BOLT.OPTIMUM.UNITY | 56 - 58 | check remaining constraints |
| 25 | check INTERACTION.EQUATION | 59 | end design session |
| 26 | activate the plate constraints | 60 | save design as EXPERT.1 |

**Table 5-3:** Design Overview of the Base Plate Design Example I
(Constraint violations are marked with asterisks)

#### 5.2.1.1. DESCRIPTION OF THE DESIGN SESSION

***Cycle 1 to 3—input problem.*** As usual, the design session starts by entering the problem inputs and posting system design plans. The system plan calls for anchor bolt design after the problem input is completed.

***Cycle 4— specify major constraints.*** The user selects the INTERACTION.EQUATION and BOLT.OPTIMUM.UNITY.FACTOR as the critical constraints to which DDIS should pay special attention. Two stand-alone CHECK.CRITICAL.CONSTRAINT.GOALs are posted on the control blackboard. In order to speed up the system, DDIS also deactivates other constraints that are not selected.

***Cycle 5 to 8—design anchor bolts.*** Various parts of the bolt group are generated by the system design generators. Note that the particular order of these design actions is chosen by the designer. DDIS does not prefer one action over the other because the design plans on the control blackboard are not detailed enough to guide the design at this level.

***Cycle 9 to 10—check critical constraints.*** The INTERACTION.EQUATION is checked and satisfied. However, the BOLT.OPTIMUM.UNITY.FACTOR is unsatisfied. The anchor bolts are overdesigned.

***Cycle 11 to 16—fix BOLT.OPTIMUM.UNITY.FACTOR constraint violation.*** All backtracking provokers are invoked, and no particular preferences are given to them because DDIS has no knowledge (i.e., redesign plans) that can apply. The designer chooses to redesign the diameter of the bolt first, and the smallest bolt (1/2" diam.) is selected. DDIS rechecks the optimum constraint and finds out that the bolts are still overdesigned. This time, the designer wants to modify the bolt pattern. He/she selects the backtracking provoker that redesigns the pattern of bolts, and the redesign adviser suggests a 4-bolt pattern (the original design is an 8-bolt pattern). The two design modifications resolve the constraint violation.

***Cycle 17—recheck INTERACTION.EQUATION.*** The INTERACTION.EQUATION is no longer satisfied after the design modifications. The bolts are underdesigned.

***Cycle 18 to 20—fix INTERACTION.EQUATION violation.*** To overcome the design failure, the designer modifies the bolt diameter by using a larger bolt (5/8" instead of 1/2").

***Cycle 21—recheck BOLT.OPTIMUM.UNITY.FACTOR.*** Although the INTERACTION.EQUATION is satisfied now, the bolts are overdesigned again.

***Cycle 22 to 23—fix BOLT.OPTIMUM.UNITY.FACTOR violation.*** The designer chooses to redesign the bolt circle distance this time because the unity factor is not

too faraway from the requirement. Then, DDIS suggests a value of 19" for the bolt distance based on the smaller bolt size used by the current design.

***Cycle 24 to 26—recheck critical constraints and active more constraints.*** The two constraints are satisfied now, so the designer actives the other constraints related to the base plate.

***Cycle 27 to 37—design base plate.*** The bolt design is complete. The next goal of the BASE.PLATE.SYSTEM.DESIGN.PLAN is DESIGN.BASE.PLATE. The designer again selects a sequence of actions from DDIS's executable agenda from cycle 27 to 37 according to his/her judgment. Up to this point, the design is fine. Nine of eleven constraints are satisfied.

***Cycle 38—check MIN.BOLT.CYCLE.DISTANCE constraint.*** The constraint is unsatisfied. Backtracking is needed.

***Cycle 39 to 41—fix MIN.BOLT.CYCLE.DISTANCE constraint violation.*** This design failure is fixed by increasing the bolt distance to 19.5". Note that the bolt design is modified. Therefore, many constraints need to be rechecked.

***Cycle 42—recheck INTERACTION.EQUATION.*** The INTERACTION.EQUATION is still satisfied.

***Cycle 43—recheck SINGLE-BOLT PLATE BENDING constraint.*** The constraint PLATE.BENDING.SINGLE.BOLT is not satisfied. The design needs to be modified again.

***Cycle 44 to 46—fix PLATE.BENDING.SINGLE.BOLT constraint violation.*** There are a few design modifications that we can do to resatisfy the constraint (e.g., change the bolt spacing, the bolt distance or even the bolt pattern). The designer chooses to redesign the plate thickness in order to isolate the effect of the changes. The redesign adviser suggests increasing plate thickness to 3/4", and this resolves the problem.

***Cycle 47 to 52—recheck other constraints.*** Several constraints are rechecked to make sure they are still satisfied after the above design modification. They are all satisfied except AISC.TABLE.1.16.5.1.MIN.ED.

***Cycle 53 to 55—fix AISC.TABLE.1.16.5.1.MIN.ED constraint violation.*** The edge distance is not enough according to AISC table 1.16.5.1. Therefore, the diameter of the plate is increase to 21.75".

***Cycle 56 to 59—recheck constraints.*** The three constraints that are influenced by the latest change are rechecked. All are satisfied. Therefore, the design is finished at cycle 59.

*Cycle 60—save design.* DDIS saves the design in its case memory knowledge as case EXPERT.1 at the end. The primary attributes of the case are shown in Figure 5-8. The control strategies of the whole design process are captured in one design plan and six redesign plans. Figure 5-10 shows the design plan and Figure 5-9 shows one of redesign plans.

```
CASE.NAME:   EXPERT.1

DESIGN.PLAN:   EXPERT.1.DESIGN.PLAN

PROBLEM.INPUT:
    ((P 21) (M 71) (V 19) (POLE.BASE.DIA 15)
     (POLE.SIDES 12) (WELD.ELECTRODES E70XX)
     (WELD.TYPE FILLET) (WELD.SIZE 0.75))

REDESIGN.PLANS:
    EXPERT.1.REDESIGN.PLAN.1    EXPERT.1.REDESIGN.PLAN.2
    EXPERT.1.REDESIGN.PLAN.3    EXPERT.1.REDESIGN.PLAN.4
    EXPERT.1.REDESIGN.PLAN.5    EXPERT.1.REDESIGN.PLAN.6

SOLUTION:
    ((THE DIAMETER OF BASE.PLATE.1 IS 21.75)
     (THE MATERIAL OF BASE.PLATE.1 IS ASTM.A572.GR60)
     (THE THICKNESS OF BASE.PLATE.1 IS 0.75)
     (THE DIAMETER OF BOLT.1 IS 0.625)
     (THE BOLT.SPACING OF BOLT.GROUP.1 IS 6.75)
     (THE NUMBER.OF.BOLTS.PER.QUADRANT OF
          BOLT.GROUP.1 IS 1)
     (THE BOLT.DISTANCE OF BOLT.GROUP.1 IS 19.5))

STORAGE.LOCATION:    261MT1:KEE:WANG:CASE-
   MEMORY:EXPERT.1.W
```

**Figure 5-8:** The Primary Attributes of Case EXPERT.1

```
GOAL.LIST:
    (EXPERT.1.REDESIGN.PLAN.4-REDESIGN.BOLT.GROUP.1.
              BOLT.DISTANCE
     EXPERT.1.REDESIGN.PLAN.4-USER.SELECT.DIAMETER.
              OF.BOLT.CIRCLE
     EXPERT.1.REDESIGN.PLAN.4-CHECK.CONSTRAINT.
              MIN.BOLT.CIRCLE.DISTANCE)

INTENTION:
    (CONSTRAINTS-SATISFIED-P 'MIN.BOLT.CIRCLE.DISTANCE)

ORIGINATED.CASE: EXPERT.1
```

**Figure 5-9:** EXPERT.1.REDESIGN.PLAN.4

```
GOAL.LIST
    (EXPERT.1.DESIGN.PLAN-USE.LARGE.2-1/4.BOLTS
     EXPERT.1.DESIGN.PLAN-START.WITH.8-BOLT.PATTERN
     EXPERT.1.DESIGN.PLAN-SYSTEM.SELECT.
              BOLT.CIRCLE.DISTANCE
     EXPERT.1.DESIGN.PLAN-SYSTEM.SELECT.BOLT.SPACING
     EXPERT.1.DESIGN.PLAN-CHECK.CONSTRAINT
              BOLT.OPTIMUM.UNITY.FACTOR
     EXPERT.1.DESIGN.PLAN-CHECK.CONSTRAINT
              INTERACTION.EQUATION
     EXPERT.1.DESIGN.PLAN-CHECK.CONSTRAINT.
              BOLT.OPTIMUM.UNITY.FACTOR
     EXPERT.1.DESIGN.PLAN-USER.SELECT.PLATE.MATERIAL
     EXPERT.1.DESIGN.PLAN-SYSTEM.SELECT.PLATE.SIZE
     EXPERT.1.DESIGN.PLAN-SYSTEM.SELECT.PLATE.THICKNESS
     EXPERT.1.DESIGN.PLAN-CHECK.CONSTRAINT.
              MIN.BOLT.CIRCLE.DISTANCE
     EXPERT.1.DESIGN.PLAN-CHECK.CONSTRAINT.
              AISC.TABLE.1.16.5.1.MIN.ED
     EXPERT.1.DESIGN.PLAN-CHECK.CONSTRAINT.
              PLATE.BENDING.SINGLE.BOLT
     EXPERT.1.DESIGN.PLAN-CHECK.REMAINING.CONSTRAINTS)

INTENTION:
    (VALUE-DESIGNED-P
            '(BOLT.1 DIAMETER)
            '(BOLT.GROUP.1 NUMBER.OF.BOLTS.PER.QUADRANT)
            '(BOLT.GROUP.1 BOLT.DISTANCE)
            '(BOLT.GROUP.1 BOLT.SPACING)
            '(BASE.PLATE.1 DIAMETER)
            '(BASE.PLATE.1 MATERIAL)
            '(BASE.PLATE.1 THICKNESS))

ORIGINATED.CASE: EXPERT.1
```

**Figure 5-10:** EXPERT.1.DESIGN.PLAN

### 5.2.1.2. ANALYSIS OF THE DESIGN SESSION

Without detailed design plans, DDIS does not have much control knowledge to schedule actions. The design path that the designer chose is graphically shown in Figure 5-11. Compared with Figure 5-6, Figure 5-11 took a different design approach. The initial ordering of different design tasks is different. The backtracking levels used are different. All these are captured in the saved case EXPERT.1 representing the individual design judgment and style of that particular designer. The following example will reuse this case and reproduce the same decision when appropriate conditions appear.

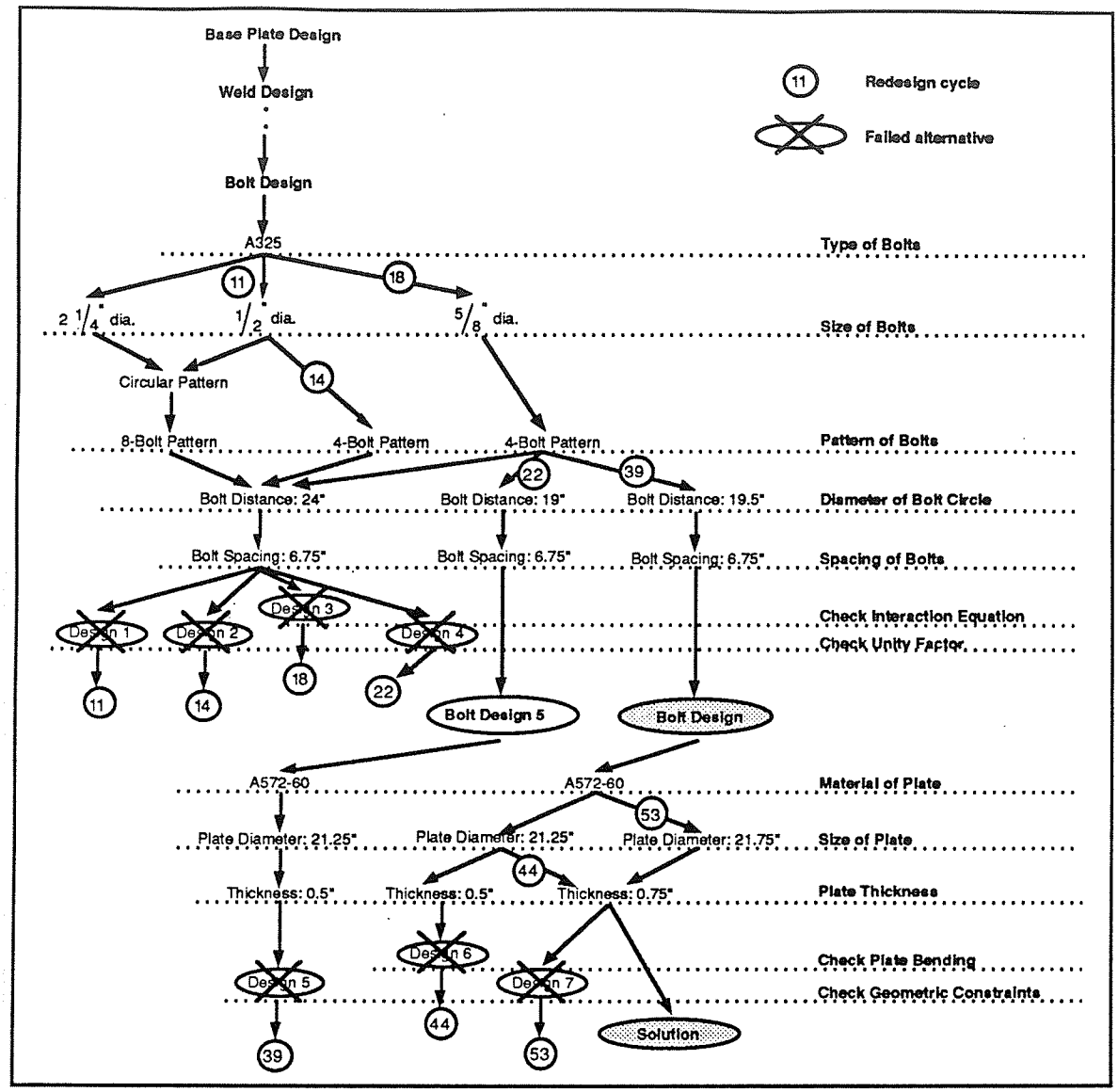



**Figure 5-11:** The Solution Path of the Base Plate Design Example I

## 5.2.2. Base Plate Design Example II

Now, we give DDIS a very different problem. The inputs for a new base plate design are shown in Figure 5-7 (b). The loadings are much higher and the dimension of the pole is also much larger than the sample problem. Apparently the previous design solution is not feasible for the new design. However, the design plans used by the experienced designer can still be applied to the new problem to produce an entirely different design. Let's see how DDIS takes advantage of its case-based design ability using its newly acquired case-dependent knowledge. Table 5-4 describes the design session, and the explanation of the design session follows.

| Cycle | Action | Cycle | Action |
|---|---|---|---|
| 1 - 2 | input the new problem | 20 | use more bolts (20-bolt pattern) |
| *3 - 4* | *retrieve and rate case EXPERT.1* | 21 | check INTERACTION.EQUATION * |
| *5* | *reuse critical consts. of EXPERT.1* | 22 | redesign bolt pattern |
| 6 | expand goal to heuristic plan | 23 | use 36-bolt pattern |
| *7* | *reuse EXPERT.1.DESIGN.PLAN* | 24 | check INTERACTION.EQUATION |
| 8 | use 2-1/4" bolts | 25 | check BOLT.OPTIMUM.UNITY |
| 9 | start with 8-bolt pattern | *26* | *reuse EXPERT.1.DESIGN.PLAN* |
| 10 | use 85" bolt distance | *27* | *reuse ASTM.A572.GR60 material* |
| 11 | check MIN.BOLT.CIRCLE.DIST * | 28 | use 95" diam. plate |
| *12* | *reuse EXPERT.1.REDESIGN.PLAN.4* | 29 | check AISC min. edge dist. table |
| 13 | redesign bolt distance | 30 | use 4" thick plate |
| 14 | increase bolt distance to 87" | 31 | check single-bolt plate bending |
| 15 | check MIN.BOLT.CIRCLE.DIST | 32 | activate all constraints |
| 16 | use 6.75" bolt spacing | 33 - 38 | check remaining constraints |
| 17 | check INTERACTION.EQUATION * | 39 | end design session |
| *18* | *reuse EXPERT.1.REDESIGN.PLAN.2* | 40 | save design session |
| 19 | redesign bolt pattern | | |

**Table 5-4:** Design Overview of the Base Plate Design Example II
(Case-dependent actions are in italics and
constraint violations are marked with asterisks)

### 5.2.2.1. DESCRIPTION OF THE DESIGN SESSION

***Cycle 1 to 2—input problem.*** The new design session starts from selecting the global BASE-PLATE.SYSTEM.DESIGN.PLAN and entering the problem input variables.

***Cycle 3 to 4—retrieve and rate case EXPERT.1.*** Case EXPERT.1 is retrieved for its potentially reapplicable design plans although the solution is not reusable. DDIS wants to transfer the design steps of the experienced design to the new design. 45 and 70 are assigned to the solution similarity and plan similarity rating of EXPERT.1, respectively.

***Cycle 5—reuse previous critical constraints as major constraints.*** During the previous design process, five constraints (INTERACTION.EQUATION, BOLT.OPTIMUM.UNITY.FACTOR, PLATE.BENDING.SINGLE.BOLT, MIN.BOLT.CIRCLE.DISTANCE and AISC.TABLE.1.16.5.1.MIN.ED) were violated. Therefore, DDIS's REUSE.PREVIOUS.CRITICAL.CONSTRAINTS action marks them as the major constraints of the new design and places five CHECK.CRITICAL.CONSTRAINT.GOALs on the blackboard accordingly. All other constraints are deactivated at this time. This action assures that the constraints most likely to be critical are checked as soon as they become checkable. Moreover, the performance of DDIS does not suffer because it does not have to handle too many constraints at the same time.

***Cycle 6—expand START.DESIGN goal.*** The START.DESIGN goal of BASE-PLATE.SYSTEM.DESIGN.PLAN is expanded into BASE-PLATE.DESIGN.PLAN.1 whose first goal is DESIGN.ANCHOR.BOLT. At this point, we can also reuse the EXPERT.1.DESIGN.PLAN.1 to expand START.DESIGN. However, the moderate plan similarity rating of EXPERT.1 makes DDIS think that it is better to transfer the old plan piece by piece rather than take it as a whole.

***Cycle 7—transfer EXPERT.1.DESIGN.PLAN.1 for DESIGN.ANCHOR.BOLT.*** DDIS transfers the relevant goals for anchor bolt design from EXPERT.1.DESIGN.PLAN.1 to the control blackboard. The lack of detailed heuristic design plans in the knowledge base is remedied by the plan transformer.

***Cycle 8 to 10—design anchor bolts.*** With the case-dependent plan on the blackboard, DDIS follows the previous design procedure step by step. First, it selects the largest bolt size (2-1/4" diam.) and the 8-bolt pattern design. Then, the same knowledge source used in the sample design is used again to generate the bolt distance. Because of the different problem inputs, the design value turns out to be different even the same heuristics are used to design the bolts.

***Cycle 11—check MIN.BOLT.CYCLE.DISTANCE constraint.*** This constraint is found unsatisfied late in the earlier design session, which causes a great deal of trouble when modifying that design. This experience is transferred into this design by declaring the constraint as a critical constraint at cycle 5. Therefore, DDIS knows to check the constraint earlier to avoid the same mistake. Indeed, this constraint is violated.

***Cycle 12 to 15—reuse earlier redesign plan.*** The constraint violation triggers three backtracking provoker actions. However, the current control knowledge on the blackboard cannot differentiate which one is better. In order to reuse the previous design modification steps, DDIS posts EXPERT.1.REDESIGN.PLAN.4 on the blackboard. The redesign goals in the case-dependent plan (see Figure 5-9) guide DDIS through the design modification. At the end of the redesign process, DDIS recovers from the design failure by choosing a bolt distance of 87 inches and is back on the course of its original design plan.

*Cycle 16—resume the anchor bolt design.* The system generator selects 6.75" bolt spacing for the new design.

*Cycle 17—check INTERACTION.EQUATION.* The anchor bolt design is completed now. Another critical constraint, the INTERACTION.EQUATION, is checked and found unsatisfied.

*Cycle 18—reuse earlier redesign plan.* DDIS tries to reuse EXPERT.1.REDESIGN.PLAN.2, which encountered the same constraint violation earlier. However, the plan calls for a bolt diameter increase. This is not a feasible plan under the current situation since the failed design already uses the largest available bolts (2-1/4").

*Cycle 19 to 21—redesign bolt pattern.* The user steps in to tell DDIS to redesign the bolt pattern. Because of the big gap between the design value and the requirements, a 20-bolt pattern is selected. However, the INTERACTION.EQUATION is still violated. Further redesign is necessary.

*Cycle 22 to 24—repeat redesign steps.* The 20-bolt pattern is still underdesigned since the constraint is still violated. The same actions are reused again to start another round of design modification. DDIS repeats the redesign steps in cycle 19, 20, and 21 and settles on a 36-bolt pattern that satisfies the INTERACTION.EQUATION.

*Cycle 25—check BOLT.OPTIMUM.UNITY.FACTOR.* Another critical constraint, the BOLT.OPTIMUM.UNITY.FACTOR, is checked and found satisfied. The bolt design is completed now. All the critical constraints related to the anchor bolts are satisfied.

*Cycle 26—transfer case-dependent plan for base-plate design.* The goal DESIGN.ANCHOR.BOLT is achieved. Therefore, it is removed from the control blackboard as well as the case-dependent plan from EXPERT.1 that expands it. With no specific plans for the base plate design, DDIS refers to the design plan of EXPERT.1 to guide its actions again. The goals for plate design are transferred to the control blackboard to expand DESIGN.BASE.PLATE.

*Cycle 27 to 31—design base plate.* To design the base plate, DDIS follows the steps that the experienced designer used when solving the sample problem and interrupts the design only to check the two previously violated critical constraints. This time everything goes well. The completely designed base plate satisfied the two constraints.

*Cycle 32—activate all constraints.* Since all the critical constraints are satisfied, the user has the option to end the design or to make more constraints active. The rest of the unchecked constraints are made active.

*Cycle 33 to 38—check remaining constraints.* Six newly activated constraints are checked and all satisfied.

*Cycle 39 to 40—end and save the design session.* Since all the constraints are satisfied, the design is completed at cycle 39 and saved in the case memory as POLE.150 at cycle 40. The final design of the pole base is:

```
((THE DIAMETER OF BASE.PLATE.1 IS 95.0)
 (THE MATERIAL OF BASE.PLATE.1 IS ASTM.A572.GR60)
 (THE THICKNESS OF BASE.PLATE.1 IS 4.0)
 (THE DIAMETER OF BOLT.1 IS 2.25)
 (THE BOLT.SPACING OF BOLT.GROUP.1 IS 6.75)
 (THE NUMBER.OF.BOLTS.PER.QUADRANT OF BOLT.GROUP.1 IS 9)
 (THE BOLT.DISTANCE OF BOLT.GROUP.1 IS 87.0))
```

#### 5.2.2.2. ANALYSIS OF THE DESIGN SESSION

The final solution of this problem is very different from the solution of the EXPERT.1 case although the previous design control knowledge was used to schedule actions in this session. Design plans, which are previous design strategies, can be applied to a large range of problems to generate completely different designs. Figure 5-12 shows the new design path. It follows the previous line of reasoning except that constraint checking was done more wisely. However, even when the same KS is used to perform a specific task, the design value generated can be different. For example, SYSTEM.SELECT. BOLT.CIRCLE.DISTANCE KS was used at cycle 10 to generate a bolt distance. The bolt distance generated for the new design was 85 inches instead of 24 inches as used before. The rules encoded in KSs can adjust to different blackboard contexts to produce adequate value for new designs. A complete design plan covers several aspects of a design problem. This example showed that DDIS can flexibly transfer part of a plan to perform a subtask of a new design. For example, the EXPERT.DESIGN.PLAN.1 (which is the whole design plan of the previous base plate design session) was used for the subtask of designing the anchor bolts in cycle 7. This example also showed that DDIS anticipated potential design failures by declaring critical constraints at the beginning based on past design experience. At the end of this design session, DDIS increased its case-dependent knowledge by saving the design in the case memory. In addition to capturing one more design solution and plan, DDIS also learned one more way to repair the underdesign of anchor bolts (i.e., INTERACTION.EQUATION violation).

## 5.3. Comments on the Two Demonstration Applications

The beam-column design problem is adequate for design solution reuse. Previous design cases can provide shortcuts and make the design cycle converge a little faster. However, the limited design space does not make the use of previous design plans an important capability. On the other hand, the base plate design problem can really take advantage of DDIS's ability to transfer past solutions and plans. The complicated design

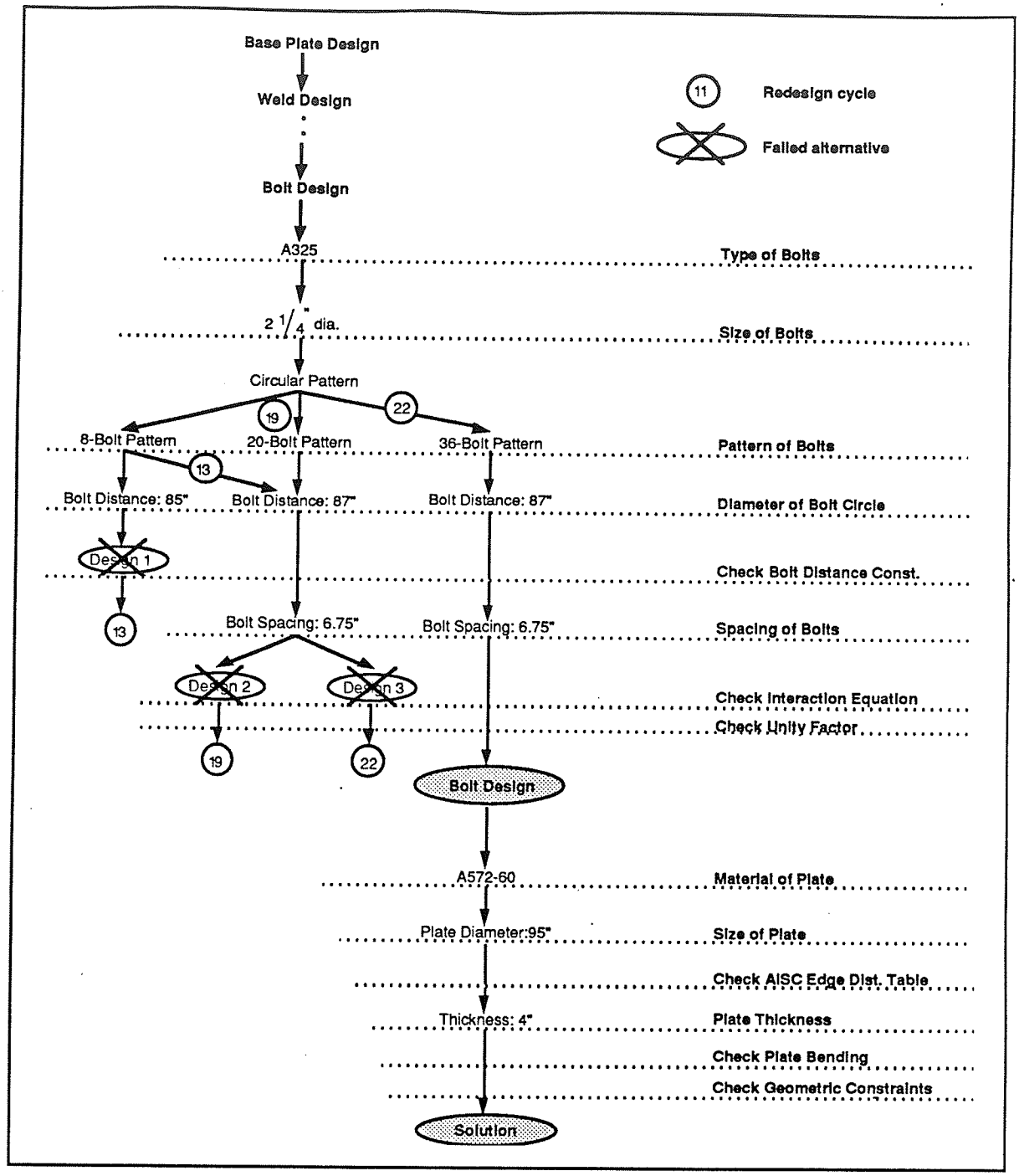



**Figure 5-12:** The Solution Path of the Base Plate Design Example II

process and large number of design options make the base plate design a knowledge intensive problem. While it can benefit from a better starting point, it can also benefit from a systematic design plan. A good initial trial design helps DDIS reach the solution faster, and an effective design plan reduces the design search space. Most importantly, plans about how to recover from design failures are critical to the generate-test-modify design

paradigm. A good redesign plan repairs the defect promptly and confines the scope of design modification. As shown in the examples, DDIS's plan transformer can be effectively used in these areas.

Further study of the problem shows that the large number of design constraints and variables slow down DDIS dramatically. So, constraint relaxation was added to DDIS's generic design actions. In addition to relying on case-independent knowledge to declare critical constraints, case-dependent knowledge about previous design failures can also be used to anticipate possible design problems. Therefore, a failure anticipator with one KS—REUSE.PREVIOUS.CRITICAL.CONSTRAINTS—was added to the system.

Very limited heuristics are in DDIS's knowledge base for base plate design. The experiment (as shown in the examples) demonstrates the memory-oriented learning ability of DDIS. Giving DDIS an incomplete domain knowledge base, it is probable that DDIS can improve its performance through design sessions with human interaction.

The blackboard architecture of DDIS and its case-independent and case-based knowledge modules provide a cooperative design environment to integrate case-base reasoning with knowledge-based design. The two design problems validate the power of DDIS's integrated reasoning paradigm. They also reveal the difficulties of capturing case-dependent knowledge in the form of design strategies. While recording the designer's steps as design plans, DDIS can capture the explicit basis of the design plan, such as the ingredient design attributes, satisfied constraints, unsatisfied constraints, performed actions, etc. However, the implicit supporting reasoning behind design plans can not be acquired (e.g., judgment based on unrepresented design considerations and analysis of values from several objects, etc.). To more effectively use case-dependent plans, DDIS needs to capture the undeclared reasons associated with individual human design decision. Furthermore, DDIS can not justify design steps of its user. It only records the designer's steps and assumes their correctness.

# Chapter 6
# Summary and Conclusions

The intent of this study was to investigate a new approach for implementing knowledge-based structural design systems using both case-dependent and case-independent knowledge. The resulting system, DDIS, combines case-based reasoning with case-independent knowledge in a blackboard framework very similar to BB1 [Hayes-Roth 84 and 85]. In the blackboard model, the knowledge needed to solve a problem is partitioned into independent knowledge sources that are grouped into several knowledge modules in the knowledge base. The knowledge sources modify only the global data structure (blackboard) and respond opportunistically to the changes on the blackboard. DDIS has two major knowledge modules: case-dependent and case-independent. The case-independent module represents abstract knowledge about the problem domain and problem solving strategies. The case-dependent module uses case-based reasoning techniques to transfer knowledge from previous designs chosen by the human designer to current design tasks. Using the blackboard control mechanism and the two knowledge modules, DDIS can apply both case-dependent and case-independent knowledge to perform collaborative and opportunistic design.

The development of DDIS involved research in producing a formal representation for structural engineering case-dependent knowledge, developing an architecture for applying case-dependent knowledge to design problems, implementing a prototype computer system based on the architecture developed, and demonstrating the prototype system with structural engineering examples.

Many issues of case-based reasoning, case-dependent design in particular, are raised by the work. The rest of this chapter discusses these general issues and examines how well DDIS addresses each of them. This discussion also includes the limitations of DDIS and how DDIS could be expanded in these areas.

## 6.1. Design Recording

In addition to the standard elements that must be described in blackboard systems, such as control knowledge, states, actions, constraints, etc., DDIS needs to acquire and retain past designs for future reuse. The questions are: what information about the original design is needed? when should it be recorded? and how can it be expressed?

The most important elements of case-dependent knowledge identified in this study are: design solutions, justifications, constraints, failures, plans and goals (see Section 3.3).

Design solutions include final solutions, intermediate solutions and partial solutions. Design justifications are the calculations of previous design variables and the dependences of their values. Design constraints are used to evaluate designs and are very important information for understanding the history of a design case. Previous design failures can be used to avoid unsuccessful design alternatives in the future. Design plans are the strategies used to solve a design problem. Specific knowledge about how to achieve a particular design step is contained in a design goal. Design goals are included in plans to form complete case-dependent control knowledge of a case.

A design session in DDIS can be divided into two steps, a design phase and a memory-oriented learning phase. The design case to be recorded is captured at the end of the design phase. Therefore, the design performance of DDIS is not significantly influenced because of its additional learning task. Only a minimum amount of necessary information (e.g., design history—the sequence of design actions) is recorded during the design phase. Most of the case-dependent knowledge is captured during the learning phase by processing the recorded design information on the blackboard.

The case-based design capabilities of DDIS are fundamentally supported by the underlying model of the system, and the model directly influences the representation of design solutions and the description of solution strategies (i.e., the plans). The solution of a recorded design includes the final solution and intermediate propositions. The final solution is saved as object attribute values, and the intermediate solutions, value justifications and constraint statuses are recorded as variable dependency graphs in KEEworlds format. Case-dependent design strategies are captured as case-dependent plans and goals. Plans and goals are created after designs using design history and constraint statuses, as well as other information on the blackboard (see Section 4.4.2 and 4.4.3). The case-dependent control knowledge captured is of little use unless it can be recognized by DDIS's control mechanism. Therefore, the plans and goals are expressed in a format that can later be directly posted on the control blackboard by case-dependent control actions to change the behavior of DDIS.

## 6.2. Design Retrieval

Given a new design problem, the concerns of design retrieval are: how can relevant previous designs be identified? when should the memory be searched? and how effective should the search be?

DDIS relies on users to retrieve relevant designs from the case memory and to decide how similar they are to the new design, although it does have the ability to find the relevant redesign plans based on the constraint violations. Our intent is to rely heavily on human interaction during design retrieval since the focus of the research was on the integration of case-dependent and case-independent reasoning.

In order to maintain the effectiveness of DDIS, the retrieved designs on the blackboard (which are retrieved from the memory by the memory probers with the assistance from the user) are the only eligible sources for case-dependent knowledge transfer. The analogy transformer knowledge module does not directly access the larger case memory. This focuses the case-dependent reasoning on the chosen designs, minimizes time-consuming memory search, and ensures uninterrupted design actions. However, either the user or a design failure can invoke the memory prober knowledge module to search for other designs to transfer more case-dependent knowledge.

The case memory of DDIS has multiple indexes and can be searched for different purposes. A design case can be retrieved for its solutions or its design plans. Design solutions are indexed by their problem statements and can be searched to find a match for a new design specification. Design plans and redesign plans are indexed by their intentions. When the intention of a case-dependent plan matches a current design goal, its associated case can be retrieved.

## 6.3. Appropriateness

Recorded designs do not always fit into a new problem perfectly. Frequently, it is necessary to modify previous solutions and plans in order to reuse them appropriately in new situations. For instance, we may want to omit the execution of unnecessary steps in an old plan or repeat the execution of the old plan until the desired subgoal is achieved. Therefore, the question is: how do we appropriately fit a previous design into a new problem?

Design solutions are reused in DDIS without modifications. The correctness of a reused solution is checked by DDIS's constraint checkers. Previous solutions are reused as the initial trial of the generate-test-modify design cycle. The appropriateness of the reuse depends on the high solution similarity rating needed to trigger the reuse solution case-dependent action.

Although case-dependent plans are reposted on the blackboard without explicit modifications, their appropriateness for reuse is determined by DDIS's plan and goal maintenance mechanism. The plan and goal maintenance mechanism and the structure of the case-dependent goals allow plans that don't fit exactly into a new design to behave correctly under the new situation. For instance, inapplicable goals in a plan can be delayed or skipped. Goals can be revisited when they need to be reachieved. Also, plans are killed automatically when they become inapplicable.

## 6.4. Flexibility

Recorded design solutions and plans cannot always be reused as a whole. A previous design solution may not be reused in its entirety, but parts of the solution can still be useful in a new problem. Furthermore, a very detailed recorded sequence of design actions that satisfies a special order of goals can rarely be followed entirely in a new design session. This is a granularity problem. Therefore, we have to ask: how can design cases in the case memory be reused even though they are only partly useful?

DDIS has a very flexible architecture and representation that can use any subset of a past design. Past design solutions can be applied to very similar new designs while previous design plans can be applied to guide designs with less surface similarity. The flexibility of solution reuse is achieved by the solution transformers, which apply part of a previous design solution to satisfy a new goal. The flexibility of plan reuse comes from several places. Previous design steps of a recorded design are decomposed into a main design plan and several redesign plans in the case memory, so that they can be reused separately. The different levels of generalization built into case-dependent plans enable them to be followed very flexibly according to the new situations they encounter (see Section 4.4.3). Also, DDIS's goal expansion allows previous plans to be reused at the desirable level of abstraction in the new design.

Furthermore, DDIS can work with multiple design cases simultaneously. For instance, DDIS can follow a case-dependent plan from one case by satisfying one of its goals with a design solution of another case or DDIS can reuse a partial solution from one case to satisfy a goal and reuse a solution from another case to satisfy another goal.

Another level of flexibility comes from the integrated reasoning paradigm of DDIS. Case-dependent actions compete with case-independent actions. Case-dependent and case-independent plans can work together competitively or in a complementary fashion. Case-based design actions are not always available or competent. DDIS has the flexibility to use its case-independent actions instead. It should be noted that the selection between heuristic-based and case-based design approach can be altered from one design step to another since the design is divided into generic actions and subgoals, and the blackboard architecture provides a framework for integrating multiple sources of knowledge to solve a shared problem.

## 6.5. Scope and Limitations

DDIS is a research prototype intended to demonstrate the idea of integrating case-dependent and case-independent knowledge in knowledge-based structural engineering design systems. It is not meant to be a production tool.

The scope of this study is limited to the design of structural steel beam-columns and electrical transmission pole anchor base plates. The beam-column design example is a component design problem. It was implemented to facilitate the prototyping and to demonstrate DDIS's idea for integrating case-dependent and case-independent reasoning easily. In a practical sense, beam-column design cannot really benefit from the case-dependent design approaches offered in DDIS (except for design solution reuses). The base plate design problem is at the subsystem design level. It provided adequate complexity to study the DDIS architecture. However, to explore the capture and reuse of the other kinds of case-dependent knowledge, a more complicated design problem at the system design level (such as conceptual structural design) is more desirable (and very challenging). Furthermore, the assessment of the utility of DDIS in terms of design speed, accuracy and optimization requires a more challenging design problem.

The current case memory of DDIS has only limited number of cases generated by the system. The design cases were saved after controlled sample design sessions. They were produced to validate and demonstrate different features of DDIS. To further evaluate the priority functions used in the generic knowledge sources and the heuristic weights of rating functions, more cases are needed. For example, the priority function of knowledge source EXPAND.GOAL (see Section 4.3.5) can be refined by running more cases. The base similarity (see Section 4.3.2.2) and goal importance index (see Section 4.3.5) implemented in DDIS can be polished with a large number of cases in the memory. It is necessary to study and compare more design cases to refine the knowledge source rating scheme, to improve the plan and goal weighting strategy, and to discover other potential utility of case-dependent knowledge. Furthermore, additional research in case indexing, similarity matching and memory generalization require a large case memory to begin with.

Other weaknesses of DDIS will be addressed in the following section along with their possible enhancements.

## 6.6. Directions for Future Research

The long-term goal of investigating the integrated, case-dependent and case-independent system is not to build a "black-box" designer. There are many aspects of problem-solving that humans perform very well, and our goal is to improve the interaction of computer tools and human designers by constructing intelligent design assistants that support rather control the design process. The DDIS project serves as a starting point for the development of those intelligent design assistants. Much more work is required to accomplish the larger integrated design model. DDIS can be enhanced and improved in many areas. Future research directions might focus on the following areas:

- **Similarity Matching Algorithm**—This study did not address the similarity problem. DDIS relies on users to retrieve relevant designs from the case memory and to decide how similar they are to the new design. The design

retrieval process can be improved by employing matching heuristics and similarity metric techniques that judge partial matches and serve to choose between potential cases. In [Carbonell 82], a similarity metric is used in the retrieval process for judging partial matches. It compares differences between the (a) initial state, (b) final state, (c) path constraints, and (d) operator preconditions of the new problem and the previously solved problems. A value hierarchy is used in the CHEF program [Hammond 86] to determine relative importance of a matched feature with respect to a set of features that have been compared. The similarity metric applies for single aspect comparison. The metric determines whether two cases are similar with respect to that aspect, while a value hierarchy is used to determine how much that aspect is worth in the overall comparison. However, the matching criteria and their relative importance heavily depend on the domain and the type of problems in the domain [Rafiq 89]. In order to automate DDIS's design retrieval process, further research in the area of indexing and similarity matching is needed. On the other hand, the design retrieval does not have to be fully automated. Human interaction can still play an important role during the retrieval process.

- **Design Decision Capture**—The experiments with the two sample applications revealed the difficulties of capturing case-dependent knowledge in the form of design strategies. While recording the designer's steps as design plans, DDIS can capture the explicit basis of the design plan, such as the ingredient design attributes, satisfied constraints, unsatisfied constraints, performed actions, etc. However, the implicit supporting reasoning behind design plans can not be acquired (e.g., judgment based on unrepresented design considerations and analysis of values from several objects, etc.). DDIS assures the fundamental appropriateness of transferring previous plans. However, the good result of plan reuse is not promised. The prediction of the results of plan transfer needs more high level information. To more fully assess the correctness of plan reuse, DDIS needs to capture the undeclared reasons associated with individual human design decision. Furthermore, DDIS can not justify design steps of its user. It only records the designer's steps and assumes their correctness. A possible solution to these problems is to build an interface to let the user provide justifications whenever he/she overrides the system's decisions. If case-dependent knowledge of this kind can be captured through the interface (as mentioned above), it can be presented to the designer and let the designer help DDIS to make evaluations. A more robust solution, involving acquiring design knowledge during design through decision justifications, is being studied by Garcia [Garcia 91].

- **Anticipating and Avoiding Design Failures**—The steel anchor base plate design problem showed that failure anticipation is very important when multiple levels of design process abstraction and interrelated design constraints

are involved. Therefore, past design failures are another type of case-dependent knowledge that should be captured while recording a design. A memory of failures could be added to the case memory of DDIS. Since previous bad design solutions and solution strategies can be used to eliminate unfavorable alternatives, they are as valuable as previous good solutions and strategies. Therefore, it is important to capture this kind of case-dependent knowledge and to develop the failure anticipator knowledge module to check for potential design failures and so avoid them in the new design. The current failure anticipator of DDIS has only one knowledge source that predicts the critical constraints of a new design based on a past similar design. Another solution for failure anticipation may be the use of elimination plans and goals to give negative ratings to bad alternatives.

- **Design Adaptation**—DDIS lacks the ability to modify previous design solutions and plans before reusing them. More analogy transformer knowledge sources can be added to perform case adaptations. General knowledge and heuristic rules can be used to write domain specific knowledge sources that adapt previous design solutions and plans to fit the specifications of new designs.

- **Memory Generalization**—DDIS does not have the capability to generalize the design cases in its memory knowledge base. A possible improvement is to develop a better categorized memory structure to support memory indexing and generalization. The MOP [Schank 79] memory structure is a potential candidate for this improvement.

- **Learning and Knowledge Acquisition**—DDIS focused on the case-based design aspect and its cooperation with conventional heuristic-based techniques, but a major original motivation behind the integrated design paradigm developed in this research is its ability for incremental expertise acquisition. In addition to the potential improvement of design performance, the knowledge representation and case-based design strategies developed during this research can support various processes of capturing design knowledge from experts and by automated machine learning. The research can be extended to include a learning component and an expert knowledge acquisition interface to surmount the current difficulty of acquiring domain-specific rules from experts. The new design approach enables the knowledge acquisition to be done as a case-by-case process and the results to be recorded as case-dependent knowledge. Furthermore, a learning component can extract design rules (i.e., case-independent knowledge) from this case-dependent knowledge incrementally.

# References


[AAAI 88] *Proceedings*, AAAI-88 Case-Based Reasoning Workshop, Minneapolis - St. Paul, Minnesota, American Association for Artificial Intelligence, August, 1988.

[AAAI 90] *Working Notes,* AAAI-90 Spring Symposium Series—Case-Based Reasoning, Stanford University, Stanford, California, March 27-29, American Association for Artificial Intelligence, March, 1990.

[Adeli 86a] Adeli, H. and Paek, Y., "Computer-Aided Design of Structures Using LISP," *Journal of Computer and Structures*, Vol. 22, No. 6, pp. 939-956, 1986.

[Adeli 86b] Adeli, H. and Al-Rijleh, M. M., "A Knowledge-Based Expert System for Design of Roof Trusses," *Microcomputers in Civil Engineering*, Vol. 2, No. 3, pp. 179-195, September 1986.

[AISC 80] American Institute of Steel Construction, *Manual of Steel Construction*, Eighth Edition, AISC, Chicago, IL, 1980.

[Biswas 87] Biswas, Mrinmay and Welch, James G., "BDES: A Bridge Design Expert System," *Engineering with Computers*, Vol. 2, No. 3, pp. 125-136, 1987.

[Brachman 85] Brachman, Ronald J. and Levesque, Hector J., Eds., *Readings in Knowledge Representation*, Morgan Kaufmann Publishers, 1985.

[Brown 84] Brown, D. C. and Chandrasekaran, B., "Expert Systems for a Class of Mechanical Design Activity," *Knowledge Engineering in Computer-Aided Design*, IFIP WG 5.2 Working Conference on Knowledge Engineering in Computer-Aided Design, Budapest, Hungary, September, 1984.

[Carbonell 82] Carbonell, Jaime G., *Learning by Analogy: Formulating and Generalizing Plans from Past Experience*, Technical Report CMU-CS-82-126, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, June 1982.

[Carbonell 85] Carbonell, Jaime G., *Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition*, Technical Report CMU-CS-85-115, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, March 1985.

[Chan 87] Chan, Weng Tat and Paulson, Boyd C., Jr., "Exploratory Design Using Constraints," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, Vol. 1, No. 1, pp. 59-71, 1987.

[DARPA 89] *Proceedings*, DARPA Workshop on Case-Based Reasoning, Pensacola Beach, Florida, May 31 - June 2, Defense Advanced Research Projects Agency, Kluwer Academic Publishers, May, 1989.

[Daube 89] Daube, Francois and Hayes-Roth, Barbara, "A Case-Based Mechanical Redesign System," *Proceedings*, The Eleventh IJCAI, Eleventh International Joint Conference on Artificial Intelligence, Detroit, Michigan, pp. 1402-1407, August, 1989.

[Dixon 84] Dixon, John R., Simmons, M. K. and Cohen, P. R., "An Architecture for Application of Artificial Intelligence to Design," *Proceedings*, 21th Design Automation Conference, 1984.

[Garcia 91] Carcia, Ana Cristina Bicharra and Howard, H. Craig, "Building a Model for Augmented Design Documentation", *Artificial Intelligence in Design '91*, First International Conference on Artificial Intelligence in Design, Edinburgh, UK, June 25-27, 1991.

[Garrett 86] Garrett, James H. and Fenves, S. J., *A Knowledge-Based Standards Processor for Structural Component Design*, Technical Report R-86-157, Department of Civil Engineering, Carnegie-Mellon University, Pittsburgh, PA, September 1986.

[Garrett 89] Garrett, James H., "An Object-Oriented Representation of Design Standards", *Computing in Civil Engineering*, Barnwell, Thomas O., ed., Sixth Conference on Computing in Civil Engineering, Atlanta, Georgia, 1989, American Society of Civil Engineers, pp. 267-274, 1989.

[Gentner 83] Gentner, Dedre, "Structure-Mapping: A Theoretical Framework for Analogy," *Cognitive Science*, Vol. 7, No. 2, pp. 155-170, 1983.

[Hammond 86] Hammond, Kristian J., *Case-based Planning: An Integrated Theory of Planning, Learning and Memory*, Technical Report YALEU/CSD/RR#488, Department of Computer Science, Yale University, October 1986.

[Hayes-Roth 84] Hayes-Roth, Barbara, *BB1: An architecture for blackboard systems that control, explain, and learn about their own behavior*, Heuristic Programming Project Report HP-84-16, Stanford University, December 1984.

[Hayes-Roth 85] Hayes-Roth, Barbara, "A Blackboard Architecture for Control," *Artificial Intelligence*, Vol. 26, pp. 251-321, 1985.

[Howard 89] Howard, H. C., Wang, J., Daube, F., and Rafiq, T., "Applying Design-Dependent Knowledge in Structural Engineering Design," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, Vol. 3, No. 2, pp. 111-124, 1989.

[Howe 86] Howe, Adele E., Cohen, Paul R., Dixon, John R. and Simmons, Melvin K., "Dominic: A Domain-Independent Program for Mechanical Engineering Design," *The International Journal for Artificial Intelligence in Engineering*, Vol. 1, No. 1, pp. 23-28, July 1986.

[Huhns 87] Huhns, Michael N. and Acosta, Ramon D., *Argo: An Analogical Reasoning System for Solving Design Problems*, MCC Technical Report AI/CAD-092-87, Microelectronics an April 1987.

[Kedar-Cabelli 85] Kedar-Cabelli, Smadar, *Purpose-Directed Analogy*, Technical Report ML-TR-1, Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903, July 1985.

[Kolodner 88] *Proceedings,* DARPA Workshop on Case-Based Reasoning, Kolodner, Janet L., Eds., Clearwater Beach, Florida, May 10-13, Defense Advanced Research Projects Agency, Kluwer Academic Publishers, May, 1988.

[Kunz 84] Kunz, John C., Kehler, Thomas P. and Williams, Michael D., "Applications Development Using a Hybrid AI Development System," *AI Magazine*, Vol. 5, No. 3, pp. 41-54, Fall 1984.

[Maher 85] Maher, Mary Lou and Fenves, Steven J., *HI-RISE A Knowledge-Based Expert System for the Preliminary Structural Design of High Rise Buildings*, Technical Report R-85-146, Department of Civil Engineering, Carnegie-Mellon University, Pittsburgh, PA, January 1985.

[Maher 87] Maher, Mary L., Ed., *Expert Systems for Civil Engineers: Technology and Application, American Society of Civil Engineers,* 345 East 47th Street, New York, New York 10017-2398, 1987.

[Maher 88] Maher, M. L., "Engineering Design Synthesis: A Domain Independent Representation," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, Vol. 1, No. 3, 1988.

[Maher 91] Maher, M. L. and Zhang, D. M., "CADSYN: Using Case and Decomposition Knowledge for Design Synthesis", *Artificial Intelligence in Design '91*, First International Conference on Artificial Intelligence in Design, Edinburgh, UK, June 25-27, pp. 137-150, 1991.

[Mittal 86] Mittal, S., Dym, C. L. and Morjaria, M., "PRIDE: An Expert System for the Design of Paper Handling Systems," *Computer*, Vol. 19, No. 7, pp. 102-114, 1986.

[Mostow 87] Mostow, Jack and Barley, Mike, *Automated Reuse of Design Plans*, Technical Report ML-TR-14, Department of Computer Science, Rutgers University, May 1987.

[Navinchandra 88] Navinchandra, D., "Case Based Reasoning in CYCLOPS, a Design Problem Solver," *Proceedings*, DARPA Case-Based Reasoning Workshop, Kolodner, Janet, Ed., Morgan Kaufmann Publishers, May, 1988.

[Prieditis 88] Prieditis, Armand, Ed., *Analogica*, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1988.

[Quillian 68] Ouillian, M. R., "Semantic Memory," in *Semantic Information Processing*, Minsky, M. Ed., MIT Press, Cambridge, MA, 1968.

[Rafiq 89] Rafiq, Taufiq, *Similarity in Structural Component Case Bases*, unpublished Engineer's degree thesis, Department of Civil Engineering, Stanford University, 1989.

[Schank 79] Schank, R. C., *Reminding and Memory Organization: An Introduction to MOPs*, Technical Report 170, Yale University Department of Computer Science, 1979.

[Schank 81] Schank, R. C., "Failure-driven Memory," *Cognition and Brain Theory*, Vol. 4, No. 1, pp. 41-60, 1981.

[Shortliffe 76] Shortliffe, Edward H., *Computer-Based Medical Consultations: MYCIN*, New York: American Elsevier, 1976.

[Slade 91] Slade, Stephen, "Case-Based Reasoning: A Research Paradigm," *AI Magazine*, Vol. 4, No. 1, pp. 42-55, 1991.

[Sriram 86] Sriram, D., "*Knowledge-Based Approaches for Integrated Structural Design*," unpublished Ph.D. Dissertation, Department of Civil Engineering, Carnegie-Mellon University, Pittsburgh, PA, February 1986.

[Tulving 83] Tulving, E. *Elements of Episodic Memory*, Oxford University Press, Oxford, 1983.

[Waterman 86] Waterman, Donald A., *A Guide to Expert Systems*, Addison-Wesley, Reading, MA, 1986.

[Winston 80] Winston, Patrick H., *Learning and Reasoning by Analogy: The Details*, Technical Report AIM 520, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1980.

[Zhao 88] Zhao, F. and Maher M. L., "Using Analogical Reasoning to Design Buildings," *Engineering with Computers*, Vol. 4, No. 3, pp. 107-119, 1988.

# Abbreviations and Acronyms

| | |
|---|---|
| BB1 | A domain-independent blackboard architecture conceived and developed by Dr. Barbara Hayes-Roth at Stanford University |
| CBR | Case-Based Reasoning |
| DDIS | Design-Dependent and Design-Independent System ("Design-dependent" and "design-independent" are the terms used for "case-dependent" and "case-independent" in the early stage of this research.) |
| KBES | knowledge-based expert system |
| KEE | Knowledge Engineering Environment developed by IntelliCorp |
| KS | knowledge source |
| KSAR | knowledge source activation record |
| LHS | left-hand side of a rule |
| RHS | right-hand side of a rule |
| TMS | truth maintenance system |