INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
An extensible, task-specific shell for routine design problem-solving

Herman, David Joseph, Ph.D.
The Ohio State University, 1992
AN EXTENSIBLE, TASK-SPECIFIC SHELL FOR ROUTINE DESIGN PROBLEM SOLVING

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

David Joseph Herman, B.S., M.S.

The Ohio State University

1992

Dissertation Committee:
B. Chandrasekaran
John R. Josephson
Tom Bylander

Approved by
Adviser
Department of Computer
and Information Science
ACKNOWLEDGEMENTS

I wish to thank my advisor, B. Chandrasekaran, who provided the conceptual framework for this dissertation as well as the research and working environment that made it possible; John Josephson and Tom Bylander, who provided valuable comments as members of my reading committee. I would also like to thank David C. Brown, who helped me make the transition from industry to academia, and on whose work in design much of this work is based. I am also indebted to the members of the LAIR and the LAIR staff, past and present, who made working at the LAIR such a memorable experience. In particular, my gratitude goes out to Chris Putnam, and my former LAIR-mates: Mike Tanner, Dean Allemang, Anne Keuneke and Bill Punch.

Many people associated with the CIS Department also deserve recognition for their support, in many different forms, of my work at Ohio State. I would especially like to thank Dr. Merv Muller, the Department Chairperson, for his encouragement and advice; Marty Marlatt and the rest of the office staff for being more than just the efficient, helpful individuals that they certainly are; Carl Phillips and his staff, who work very diligently to provide a truly remarkable computing environment.

I would also like to thank the many graduate students in the CIS department I have come to know during my own tenure as a grad student, especially Deborah
Shands and my juggling partner Doug Roble, who helped me celebrate my triumphs, sympathized with my setbacks, and generally distracted me when I got too serious. Thanks.

Many friends outside the university have contributed greatly to my mental health during this seemingly endless task. Thanks to all my training partners at Aikido, in Ohio and California, for providing me with a refreshing release from a very intense cognitive task. I am especially indebted and grateful to Larry Linder for his friendship, which I value highly. I thank Paula Rojo-Vega, both for her unswerving faith and friendship in me and for encouraging me to finish my degree through blunt intimidation.

Closer to home, I thank Stephanie and her family, and also my own family for their patience, support, and unquestioning belief in my abilities during my long adventure in grad school. Your love and support made the difference, for which I am forever grateful.

This work was supported in part by DARPA/AFOSR contract #AF F 49620-89-C-0110, the Air Force Office of Scientific Research grants 87-0090 and 89-0250, and the RAMCAD Contract number F33615-87-C-001. I would also like to acknowledge the Boeing Computer Services and Ted Kitzmiller for their cooperation and interest in this work.
VITA

March 13, 1954 .................. Born—Sandusky, Ohio

1976 ............................. B.S. Computer and Information Science
                                The Ohio State University, Columbus, Ohio

                                Columbus, Ohio

                                Columbus, Ohio

1977–1980 ...................... Lecturer, Computer Science Department
                                The Ohio State University, Columbus, Ohio

1980–1982 ...................... Graduate Teaching Assistant
                                Computer Science Department
                                The Ohio State University, Columbus, Ohio

1982 ............................. M.S. Computer and Information Science
                                The Ohio State University, Columbus, Ohio

1984–1990 ...................... Graduate Research Assistant
                                Laboratory for AI Research
                                The Ohio State University, Columbus, Ohio

1988–1989 ...................... Consultant, Battelle Memorial Institute
                                Columbus, Ohio

1990–1991 ...................... Instructor, Computer Science Department
                                The Ohio State University, Columbus, Ohio
PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in

- Artificial Intelligence: Prof. B. Chandrasekaran
- Compiler Theory: Prof. J. Gourlay
- Cognitive Science: Prof. T. Milburn
# Table of Contents

ACKNOWLEDGEMENTS ............................................................. ii

VITA ......................................................................................... iv

LIST OF FIGURES ..................................................................... xi

LIST OF TABLES ...................................................................... xv

CHAPTER PAGE

I Introduction ............................................................................. 1

1.1 The Design Task ................................................................. 3
1.2 Previous Work on Architectures for Routine Design ............. 5
1.3 The Vocabulary of Design ................................................... 8
1.4 A Flexible Design Shell ...................................................... 11
1.5 An Overview of the Dissertation ......................................... 12

II A View of Design Problem Solving ....................................... 14

2.1 A Style of Analysis ........................................................... 14
2.2 The Structure of the Design Task ....................................... 17
2.3 The Input to the Design Task ............................................. 18
2.3.1 A Simple Taxonomy of Design Requirements ................. 21
2.4 The Output of the Design Task ........................................... 23
2.5 Structures That Mediate the Design Process ....................... 26
2.5.1 Representing Devices .................................................... 26
2.5.2 Representing the Functions of Devices ........................... 31
2.6 Methods for the Design Process ........................................ 32
2.6.1 A Task Breakdown of the Design Process ....................... 32
2.6.2 Methods for Proposing Design Solutions ....................... 37
## 7.2 The Kernel Architecture

7.2.1 Future Work

## 7.3 The User Interface

7.3.1 Future Work

## 7.4 System Building Aids

7.4.1 Future Work

### APPENDICES

A The User Interface

A.1 Loading DSPL++

A.2 The DSPL++ Icon

A.2.1 Running DSPL++

A.3 The Knowledge Base Browser

A.4 The Design Database Browser

A.5 The Case Browser

A.5.1 Attribute Displays

A.6 The Goal Stack Browser

A.6.1 Controlling the Interpreter

B User's Manual

B.1 Support Functions

B.1.1 Method Selection

B.1.2 Accessing The Design Database

B.2 Language Primitives

B.3 The Goal Behavior Definition Language

B.3.1 Primitives For Controlling Execution Flow

B.4 The Method Behavior Definition Language

B.4.1 Plan Language Extensions

B.5 Accessing the Goal Stack

C Implementation Notes

C.1 Goal State Transitions

C.2 Method Syntax and the DSPL++ Compiler

C.2.1 Lexical Processing in the Expression Grammar
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>2</td>
<td>65</td>
</tr>
<tr>
<td>3</td>
<td>67</td>
</tr>
<tr>
<td>4</td>
<td>69</td>
</tr>
<tr>
<td>5</td>
<td>76</td>
</tr>
<tr>
<td>6</td>
<td>78</td>
</tr>
<tr>
<td>7</td>
<td>85</td>
</tr>
<tr>
<td>8</td>
<td>86</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>94</td>
</tr>
<tr>
<td>11</td>
<td>97</td>
</tr>
<tr>
<td>12</td>
<td>99</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>102</td>
</tr>
<tr>
<td>15</td>
<td>106</td>
</tr>
<tr>
<td>16</td>
<td>115</td>
</tr>
</tbody>
</table>
The PVCM method ........................................ 116
The Verify Design behavior definition .................. 122
The AskUser verification method .......................... 123
The Verify Design selector .................................. 124
A strategy for modifying a faulty design .................. 126
The Critique Design behavior definition ................. 127
The Modify Design behavior definition ................... 131
A strategy to pursue a faulty design ...................... 133
A strategy for discarding a faulty design ................. 136
The structure of a decomposition ........................... 144
A decomposition ............................................ 145
A specification mapper ..................................... 145
A solution mapper ......................................... 146
A plan ....................................................... 148
A primitive plan .......................................... 149
A case ....................................................... 151
A step ....................................................... 156
A constraint ................................................. 158
A verification plan ......................................... 159
A suggestion map ......................................... 162
A step redesigner .......................................... 165
A modification plan ....................................... 167
<table>
<thead>
<tr>
<th>Page</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>Levels of extensibility</td>
</tr>
<tr>
<td>40</td>
<td>A solution map from STILL</td>
</tr>
<tr>
<td>41</td>
<td>The simulation of a solution map</td>
</tr>
<tr>
<td>42</td>
<td>Steps in Acquiring a Decomposition</td>
</tr>
<tr>
<td>43</td>
<td>Manually initiating the acquisition of a new method</td>
</tr>
<tr>
<td>44</td>
<td>Selecting the type of entity to acquire</td>
</tr>
<tr>
<td>45</td>
<td>Acquiring a decomposition method</td>
</tr>
<tr>
<td>46</td>
<td>Acquiring the condenser description</td>
</tr>
<tr>
<td>47</td>
<td>Acquiring a specification mapper</td>
</tr>
<tr>
<td>48</td>
<td>The simulation of a specification map</td>
</tr>
<tr>
<td>49</td>
<td>The symbolic simulation of the final decomposition</td>
</tr>
<tr>
<td>50</td>
<td>A template for defining a new method type</td>
</tr>
<tr>
<td>51</td>
<td>The definition of the Plan method type</td>
</tr>
<tr>
<td>52</td>
<td>A spectrum of generality</td>
</tr>
<tr>
<td>53</td>
<td>The basic Blackboard architecture</td>
</tr>
<tr>
<td>54</td>
<td>The DSPL++ icon</td>
</tr>
<tr>
<td>55</td>
<td>The DSPL++ knowledge base browser</td>
</tr>
<tr>
<td>56</td>
<td>The structure of a plan in DSPL++</td>
</tr>
<tr>
<td>57</td>
<td>A DSPL++ database browser</td>
</tr>
<tr>
<td>58</td>
<td>The DSPL++ case browser</td>
</tr>
<tr>
<td>59</td>
<td>A DSPL++ device's attribute display</td>
</tr>
<tr>
<td>60</td>
<td>An example of a DSPL++ goal stack browser</td>
</tr>
</tbody>
</table>
61 A typical DSPL++ working environment .............................................. 256
62 DSPL++ goal transitions. ............................................................ 273
63 The structure of the DSPL++ modular grammar ................................ 276
64 The final goal stack ................................................................. 283
65 Part of the design results .......................................................... 284
66 Prompting for a missing attribute value ....................................... 287
67 Prompting for a missing attribute value ....................................... 287
68 Verifying the result of design ...................................................... 298
69 A detailed display of the stripping section goal stack ..................... 300
70 Redesign within the plate design ................................................ 301
71 An option to stub out missing knowledge ..................................... 303
72 Allowing the user to add a method .............................................. 303
73 Verifying the result of design ...................................................... 304
74 Verifying the result of design ...................................................... 306
75 Determining the reboiler throughput .......................................... 307
76 Checking the cost with the user .................................................. 310
# List of Tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Suitability classes for methods</td>
</tr>
<tr>
<td>2</td>
<td>The main grammar</td>
</tr>
<tr>
<td>3</td>
<td>The decision body grammar</td>
</tr>
<tr>
<td>4</td>
<td>The plan body grammar</td>
</tr>
<tr>
<td>5</td>
<td>The suggestion grammar</td>
</tr>
<tr>
<td>6</td>
<td>The expression grammar</td>
</tr>
</tbody>
</table>
CHAPTER I

Introduction

The research documented in this volume explores architectures for the construction of knowledge-based systems that perform design. This work builds on previous work in this area, in particular an analysis of the design task developed by Chandrasekaran [Chandrasekaran, 1989, Chandrasekaran, 1990a], as well as previous efforts by Brown [Brown and Chandrasekaran, 1983, Brown and Chandrasekaran, 1985, Brown and Chandrasekaran, 1986, Brown, 1984] to develop a language describing the automatic design of mechanical components by computer. Similar to the effort by Brown, one goal of this work is to produce a task-specific architecture for describing design problem solving. However, the design analysis proposed by Chandrasekaran [Chandrasekaran, 1989, Chandrasekaran, 1990a] suggests that an extensible architecture capable of supporting a large number of design methods is a more appropriate tool for building design support systems. Thus, the direction of the work described in this volume is to show how multiple design strategies can be integrated into an overall design process by a design task-specific architecture. The focus of this work is to demonstrate a system with enough flexibility and generality to take advantage of many different forms design knowledge and expertise, and still retain the leverage of an architecture specialized to the design task.
Our attraction to task-specific structures for problem solving arises from their many advantages [Chandrasekaran, 1987]. Task-specific structures are inherently modular. They match both representation and control strategies appropriately to the problem task. Knowledge acquisition and explanation are both facilitated by the structured representations of the architecture. Not the least important is the ability of task-specific structures to provide tractable solutions to problems.

However, by their very nature, task-specific structures are incomplete in that they represent only partial (i.e., task-specific) descriptions of problem solving phenomena. Any architecture depending on the task-specific view of problem solving will be correspondingly incomplete, making extensibility of the architecture an essential feature.

Furthermore, a task as multi-faceted as design will require an underlying architecture capable of flexibly recombining task-structures to suit its problem solving needs. A single task-structure controlling the overall process will be hard pressed to provide for design's inherent variability. The issue is how general should the underlying architecture be.

One approach is to use architectures for general cognition, such as SOAR [Laird et al., 1987] or the BB1 system [Hayes-Roth and Hewett, 1988, Hayes-Roth, 1985] to control the problem solving process. While not specifically addressing design, these architectures can still be usefully applied to design by virtue of their generality. For example, R1/Soar [Rosenbloom et al., 1985] uses SOAR's general model of intelligence to implement the R1 expert system. The emphasis is on demonstrating the
generality of the underlying architecture rather than on facilitating the design process.

General purpose architectures by their nature do not make any commitments to the characteristics of a particular task not shared by all other tasks, and thus they provide no task-level guidance for building practical, real-world systems. On the other hand, using a task-specific structure to drive problem solving tends to produce excessively rigid and brittle behavior that operates well only in a relatively narrow context. This work attempts to find a middle ground by providing an architecture that flexibly integrates task-structures derived from an analysis of the design task, and gracefully allows extending the set of core task-structures as needed. Beginning with an analysis of the design task encourages a more insightful view of design expert system building tools.

1.1 The Design Task

Chandrasekaran [Chandrasekaran, 1989, Chandrasekaran, 1990a] minimally describes design as the complete specification of some set of “primitive” components and their relations to meet a set of constraints. The “primitives” are often implicit in the domain, e.g., an engineer who designs circuits works in terms of resistors, capacitors, etc., as well as higher-level constructs such as filters and amplifiers. Constraints may be specified in part in the design requirements of the object to be designed. Many constraints may be implicit in the design primitives to be used. The design primitives themselves may be specified as part of the design requirements, or, more rarely, may need to be discovered as part of the design process. Constraints may refer to characteristics of the object to be designed (e.g. its total weight or size), or to some
aspect of the design process itself (e.g. constraining the parts used to be existing, off-the-shelf parts, or perhaps manufacturing limitations on the machinability of a material used in making one of the parts).

This description is a very general, domain-independent characterization of design. Many activities from planning to programming are described equally well by this definition. The MPA system [Chandrasekaran et al., 1989, Chandrasekaran et al., 1987, Herman et al., 1986], for example, is a system that performs tactical counter-air mission planning using a paradigm of planning fitting this definition. KNOBS is a constraint-based planning system [Brown et al., 1982] fitting this definition of design. Friedland’s MOLGEN [Friedland, 1979] performs a kind of design that fits this description. There are many examples in various design domains ([Frayman and Mittal, 1987], [Marcus and McDermott, 1986], [Birmingham et al., 1988], etc.). Other examples abound.

However, the above definition alone does not give sufficient guidance to build systems that can handle specific, practical problems in design. This domain-independent view must be augmented with domain-specific knowledge about a particular domain as well as strategies for efficiently exploring the domain. For example, mechanical design may call for significant amounts of spatial reasoning, while the electrical domain may involve topological reasoning. A single mode of problem solving is insufficient for describing the totality of design.

Chandrasekaran has proposed ([Chandrasekaran, 1989, Chandrasekaran, 1990a]) that design is based on a variety of types of knowledge, each of which helps solve a
portion of the design problem in a computationally efficient way. This is essentially the generic task approach to problem solving applied to the domain of design. Expertise in design problem solving consists of an accumulation of a repertoire of such knowledge. Unlike traditional expert systems, we do not view this expertise as a collection of pieces of knowledge manipulated by a single, uniform inference technique. Instead, each form of knowledge is uniquely structured and used in characteristic ways appropriate to that particular form. A plan, for example, represents a pre-determined method for achieving a goal. It has a representation specific to planning knowledge, e.g. a list of plan items, that is used in a particular fashion, typically involving the sequential execution of each plan item. Additional knowledge may be attached to the plan to assist when problems arise during the use of the plan. A list of Suggestions, for example, may be used to guide backtracking. Each different type of knowledge can produce some information that is needed or useful during design, or can generate some part of the design solution. Conversely, each type of knowledge requires certain types of information to be available before it is used.

1.2 Previous Work on Architectures for Routine Design

The DSPL language[Brown, 1984, Brown and Chandrasekaran, 1985] represents a first attempt to develop a language for describing routine design knowledge based on the generic task approach to building knowledge-based architectures for problem solving[Chandrasekaran, 1985, Chandrasekaran, 1986]. The DSPL++ architecture is an attempt to build on the strengths of DSPL and eliminate some of its weaknesses, still along the lines of generic task theory.
One characteristic of the generic task approach to knowledge based architectures for problem solving is a commitment to problem solving strategies that manipulate specific types of knowledge. DSPL is no exception. DSPL uses design plans to organize steps that select the values of attributes of the device being designed. The plans are organized around specialists associated with specific tasks in the domain. A plan-refinement strategy operates on these structures, and works to drastically reduce the search involved in designing a device. When design failures are detected by constraints, a highly constrained form of backtracking known as redesign by suggestion is used to attempt recovery.

One benefit of the generic task approach is that a knowledge-based system may use appropriately compiled knowledge during problem solving to reduce or even eliminate the need for explicit search in computationally expensive spaces. In DSPL, for example, the set of design plans to consider at each point during design is precompiled as a list within the specialist. The rating of each plan is precompiled in plan sponsors attached to each plan. Even the selection of a single plan from a set of design plans is compiled in plan selectors. Each of these forms of knowledge work to limit the complexity of search during design, and introduce search into the design process only in narrow, highly constrained circumstances.

Another feature of generic task architectures is predictability of operation, due in part to the close association between knowledge forms and strategies for manipulating them. Such a linkage is demonstrated in DSPL, where problem solving alternates between plan selection within a specialist and plan refinement within each plan within
the specialist. Depending on one's point of view this control scheme can be seen as a benefit because it produces very predictable behavior, or as a drawback because it severely limits the runtime options available during problem solving and thus appears somewhat over constraining and inflexible.

However useful the above characteristics of generic task architectures, there are still drawbacks. Search cannot be avoided completely, and locking a knowledge type to a single inference mechanism can be excessively constraining. The DSPL++ architecture attempts to expand on the strengths of DSPL and generic task architectures, while addressing some of the weaknesses.

The first change in DSPL++ is to "fracture" DSPL's rigid knowledge organization and allow pieces of design knowledge to be applied to the design process as they become applicable. This is a step away from the highly structured, static organization typical of many generic task tools and a step toward more opportunistic, dynamic problem solving.

The second change in DSPL++ is the replacement of the fixed control mechanism with a goal-driven control scheme to drive problem solving behavior during the design process. This relaxes the rigid control scheme of plan-selection/plan refinement forced by the DSPL architecture. Instead, problem solving behavior shifts according to the demands of the problem and the knowledge sources available to solve it. A goal stack within the interpreter additionally allows a clearer view of the internal state of the design process as the design progresses. These changes provide a more modular and extensible system, allowing new design behaviors to be easily integrated into the shell.
The machinery required to implement the changes to DSPL mentioned above are similar to the basic machinery and default knowledge provided by SOAR [Laird et al., 1987] to allow goal processing and subgoal generation. Many of the same issues arise in both cases, issues such as how and when knowledge retrieval should occur and deciding when goals are satisfied. SOAR automatically handles certain forms of goal termination, and knowledge retrieval, such as the set up of appropriate problem spaces and the selection of operators, is handled by SOAR’s under-lying production architecture. The gross goal-oriented behavior of SOAR is similar to the core of the DSPL++ design architecture.

Since architectures such as SOAR or BB1 have many of the desirable features of an expanded generic task architecture, one obvious line of development would be to build the necessary design abstractions using, for example, the SOAR system as an implementation layer. However, in its attempt to represent a general model of cognition, these architectures make commitments which, while important to aspects of cognition, become burdensome with respect to developing an architecture for design. In developing a useful technology for design system building, we want to only make the commitments necessary for allowing a flexible, extensible architecture, without the performance penalty associated with general purpose architectures in general.

1.3 The Vocabulary of Design

The emphasis in DSPL++ is on a design level architecture capable of coordinating multiple methods during problem solving, rather than using a single mechanism such as plan selection and refinement. This level may use problem solving mechanisms such
as plan selection and refinement, or others such as classification, or even analytical
methods such as linear programming as appropriate to the design task at hand. Many
of the features of the design task explored by earlier work such as DSPL appear
again in this broader context. The design process still exhibits properties such as
recursion. The concept of a community of cooperating specialists found in DSPL and
elsewhere [Minsky, 1986] is still appropriate, but requires some modification.

As an illustration of the many modes of problem solving in which a designer
engages, consider the problem in process engineering of the design of a distillation
column.¹ A distillation column is a complex device used to separate a mixture of
chemical compounds into its constituent compounds by utilizing differences in the
phase change characteristics of the individual compounds. Such a device is often found
in the oil refining industry, where a mixture of hydrocarbon compounds (e.g. crude
oil) is separated (refined) into many products. The design of distillation columns is an
activity associated with considerable expertise and many different kinds of engineering
knowledge. The interested reader may refer to [Economopoulos, 1987], [Henley and

Briefly, the process engineer begins with a description of components to be sepa­
rated. By examining the physical properties of each of the components, it is possible
to predict how each of the components will behave (e.g. the likelihood that it is in
either the gaseous or liquid phase) as it proceeds through an ideal column. This in­
formation is subsequently used to determine the precise physical properties of a real

¹The potential for automating this particular process was first suggested by Rose in [Rose, 1985].
Portions of this process for designing a distillation column have been captured in STILL [Myers
et al., 1988], an expert system written in DSPL.
distillation column that will approximate these results.

The simulation of the distillation column is done through various mathematical models, known generically as state equations. The process engineer needs to select an appropriate set of state equations in order to properly model the distillation process. Different mathematical models are appropriate under different circumstances. While the evaluation of the model is not an issue relative to design problem solving per se, the process of selecting of the correct model certainly is. The engineer must examine the requirements for the distillation column and classify the design problem on the basis of the column's features. Each set of state equations have their own applicability, and must be selected to match the particular design problem. Classification is one problem solving method used to help generate plausible design candidates in the early stages of distillation column design.

During the design proper, the column designer addresses the task of design subsystem by subsystem. First the characteristics of the column's trays are determined for each section of the column, then the parameters of the reboiler and condenser subsystems are determined. In other words, the designer uses a decomposition of the design task into smaller sub-tasks in order to manage the complexity of the design problem. For each sub-problem the designer may take advantage of plans, i.e., predetermined sequences of actions, to organize the design process.

The analysis of the design process in terms of sub-processes with well-defined information processing responsibilities helps in identifying types of knowledge and inference needed. This suggests an architecture for design with these sub-processes
as building blocks. Because each sub-process uses characteristic types of knowledge and inference, a mini-shell can be associated with it. Design expertise can be directly encoded using that shell. Since each task has a clear information processing responsibility, the sub-processes can communicate with each other in terms of the information that defines the input and outputs of the processes.

1.4 A Flexible Design Shell

A problem solving architecture based a task analysis such as that described by Chandrasekaran [Chandrasekaran, 1989, Chandrasekaran, 1990a] also suggests a principled way to define the human-machine interaction during design. This interaction has been previously noted by Chandrasekaran [Chandrasekaran, 1985, Chandrasekaran, 1986]. First, whenever knowledge and control can be explicitly stated for one of the design processes, that module can be built directly, by using a knowledge and control representation that is appropriate to that task. Second, if knowledge for a module is not explicitly available, the human can be part of the loop for providing information that that module would have been responsible for. For example, failure analysis and common sense reasoning involving space and time are difficult tasks to automate. The human/machine division of responsibility may be done in such a way that the machine turns to the user for the performance of these tasks. As these tasks are better understood, they can be incrementally brought into the machine side of the human/machine division of labor.

Another kind of human machine interaction is possible in this framework. Note that each sub-process is characterized both by specific types of knowledge and by
inference and related control problems. Certain control problems, such as ones that
search in the space of problem decompositions, can become quite complex. One way
that a module can interact with a domain expert is by proposing available knowledge
and letting the human make the control choices by using knowledge that has not been
made explicit in the problem solving theory. As a practical matter, this can be an
effective way of using the module as a knowledge source, even without a complete
theory of problem solving using that knowledge. The VEXED system [Steinberg,
1987], for example, works in this mode. It proposes possible decompositions, and the
user is asked to choose the one he would like to pursue. Similarly, when a design
system’s choice of design plan fails, it may turn to the user for choosing alternative
plans.

1.5 An Overview of the Dissertation

Chapter II discusses the task-level analysis of the design process. This analysis is the
foundation for all of the major architectural features in the DSPL++ system.

An architecture for building knowledge-based systems for design is presented in
Chapters III and IV. Chapter III contains the descriptions general purpose portions
of the architecture, including the underlying goal-processing mechanism, an abstract
description of goals and methods, and the database that contains the primitive objects
of a domain. This chapter also describes various types of failures and how failures
are handled in the general architecture. Chapter IV describes the taxonomy of the
design goals and methods that make the shell specific to the design task.

Chapter V illustrates some useful features of the design system that arise from
the architecture, including aides to knowledge acquisition, the ability to "symbolically simulate" aspects of design knowledge, and other benefits that arise from a system built on an analysis of the design task. Chapter V also contains an example of the ability to add onto the basic design system.

Chapter VI contains a look at other architectures for building design expert systems. This chapter compares the DSPL++ problem solving architecture to its immediate predecessor, DSPL. Several existing problem solving systems for design are also analyzed. Finally, general purpose problem solving architectures that have been adapted to the design task are examined.

Chapter VII summarizes this investigation of the design task and the DSPL++ problem solving architecture and outlines future directions of the research.
CHAPTER II

A View of Design Problem Solving

This chapter describes an analysis of design problem solving, based on previous work by Chandrasekaran [Chandrasekaran, 1989, Chandrasekaran, 1990a]. This analysis is the foundation for all of the major architectural features found in the DSPL++ system.

2.1 A Style of Analysis

To begin, let’s examine an information processing description of the top-level design task. This style of analysis is made most vivid in the work of Marr [Marr, 1982], where it was used to tie together many previously unconnected empirical studies of the human vision system. This style of analysis has also been used by Chandrasekaran (see, for example, [Chandrasekaran, 1985], [Chandrasekaran, 1983], and especially [Chandrasekaran, 1990b] for details) as the basis for the “generic task” theory of problem solving. This analysis has also been applied to the design task, first by Brown and Chandrasekaran in [Brown and Chandrasekaran, 1989], and again by Chandrasekaran in [Chandrasekaran, 1989] and [Chandrasekaran, 1990a]. It is through this lineage that this present analysis of design is descended.
Marr suggests that there are three levels that a machine performing an information processing task must be understood. They are:

• Computational theory: What is the goal of the computation, why is it appropriate, and what is the logic of the strategy by which it can be carried out?

• Representation and algorithm: How can this computational theory be implemented? In particular, what is the representation for the input and output, and what is the algorithm for the transformation?

• Hardware implementation: How can the representation and algorithm be realized physically? [Marr, 1982]

Marr used this framework in his analysis of the human visual system. The generality of the approach makes it applicable to setting the standards for understanding any complex information processing system. In our case, however, we are attempting to understand a complex cognitive system, or a portion of it, as it engages in the purposeful design of some artifact. Our model for design is the human engineer. Our Computational theories arise in large part from observing the engineer's behavior and reports. However our representations are motivated by the limitations of building and running expert systems on computers. The result should be a better understanding of the capabilities of that medium.

The following sections describe design in light of these levels. For the sake of clarity the scope of the discussion is limited to the design of a physical artifact, although the principles will likely apply to other domains related to design, such
as programming (see, for example, [Johnson and Soloway, 1985], [Rich, 1981], or [Allemang, 1990]) and certain forms of planning ([Herman et al., 1986]).

A distinction should be made at this point regarding the use of the word "design". The term "design" is used in engineering to describe two very different classes of phenomena, one describing a rather broad set of activities in general, and another, narrower set of specific activities. In the broader sense, the term "design" covers any situation in which an artifact is generated, by whatever means, to satisfy some functionality. This is what we refer to as the design task. In the narrower sense, the term "design" refers to any of the specific methods used to produce a portion of an artifact. In this case we understand design to indicate a particular process. This distinction was first identified and elaborated by Chandrasekaran [Chandrasekaran, 1989].

Marr's first two levels of analysis make clear the differences between these different uses. The Computational theory asks for a statement of what the task is, under what circumstances it can be carried out and a high-level account of the strategy of carrying out the task. This level of theory does not consider the specifics of how the task is performed. The specifics of the performance of the task, i.e., how it can be carried out, is the domain of the Representation and Algorithm level.

We start by describing the structure of the design task, and follow by addressing the issue of how the design process can be carried out.
2.2 The Structure of the Design Task

The purpose of the following sections is to describe the task of design at the level of the Computational theory. Broadly speaking, what are the inputs and outputs of design? What is represented? Why do we design, and what are our expectations of design? What are its results? We are not concerned, at this point, with any particular method for performing design, but only with structure of the overall design task.

Intuitively, we design to a purpose. An engineer\(^1\), at least one who has any hope for successfully completing a design problem, does not work in a vacuum. He is instead presented with a problem to solve, and finite resources to solve it. The engineer certainly does not approach a design problem \textit{tabula rase}, a blank slate with no prior knowledge of the issues, but rather draws on previous experience.

Rarely does an engineer engage in design without having considerable prior knowledge of the form of the final object being designed. Instead he works with “primitives” with known properties, and proceeds by combining and parameterizing these primitives to meet the given input requirements. Although each design is novel in the sense of being a unique combination of parts and parameter settings, most of the design rests on well understood and well trodden concepts and components. The development of new primitives in a design domain is usually the result of long-term efforts, and is probably better classed as a kind of research rather than part of the design process.

\(^1\)Many agents perform tasks that can reasonably be considered to be “design”, from a programmer writing a program to a homemaker planning his daughter’s birthday party. However, in order to improve readability, we shall use the term “engineer” or “designer” to indicate a human agent engaging in some form of design.
The following section describes the form of the input to a design problem as it is presented to an engineer.

2.3 The Input to the Design Task

Perhaps the simplest form of design, which can arguably even be called "design," occurs when an engineer is requested to select an instance of a known device from a predetermined set of devices that satisfies some requested parameters. In the simplest case, no mention need be made to the device's functionality, and no new components need to be created. The design task is reduced to a mapping from required features of a device to a member of a set of known objects that might satisfy the requirements. The input to the problem is simply a set of features that describe the characteristics of the desired device.

In this first case, the input features describe some observable or testable aspect of the desired device. An engineer may be asked, for example, to "design" a bolt that meets certain requirements. The requirements may be directly available, such as the bolt's length, thread size, or diameter, or may involve some inferred property, such as the bolt's ability to withstand a certain load. The method for design might be to simply to select an off-the-shelf part by examining existing examples of bolts, and choosing one that matches the features required. A variation on this theme is to select from previously proven designs, which are not strictly "off-the-shelf" but that still can be considered immediately available to the engineer. The difference in problem solving is slight. These are very simple forms of "design", but they do provide a useful endpoint for our analysis.
Note that describing the input requirements as a simple list of features of the final device to be designed does not necessarily require that the final device be selected from a pre-enumerated set, as described above. Issues of how the design might proceed are discussed in later sections.

Describing the design input requirements as a list of features of the desired device may be equivalently viewed as satisfying a list of implicit constraints on the design output. That is, each of the features of the desired device are constrained to have the values of the corresponding features mentioned in the input requirements. One might restate the bolt selection problem as "Find me a bolt, as long as the one you select has a length is 3.25 inches and has a diameter of .875 inches." The intent of the requirements is the same, only the form of the input differs.

A second form of input to a design problem, then, are explicit constraints. Constraints, as discussed above, may state limitations on features of the desired device to be designed. Constraints may also refer to limitations on the fabrication of the device, such as which machinery to use during fabrication or material selection, or even to the process of design itself. For example, an engineer may be requested to deliver a design by a certain time, or within certain resource limitations, completely aside from the characteristics of the device being designed. More likely, input constraints will refer to some derived or inferred aspect of the device, such as its cost or weight.

The most likely form of input to a design problem is a description of a functionality to be achieved. The object to be designed is expected to be capable of achieving the required functionality. While this may be the most common form of design
requirement, it also represents the most broad category of design problems. The act of creating an artifact to perform a specified function might be considered the essential definition of design.

All design, either implicitly or explicitly, must address functionality. In earlier examples the requirements were specified "below" the level of functionality, and the design requirements already implied the design to a function was completed. The design task in this case, as will be seen below, is more closely related to an instantiation task rather than the broader common sense idea of design.

Important classes of design exist that do not explicitly address satisfying a functionality. The DSPL language, for example, addresses such a class of design problems and as such did not provide any capability for representing functions as requirements. When using DSPL, the level of design dealing with functionality must be complete before an AIR-CYL-like system can even be created. In the AIR-CYL system, a list of features describe a desired air cylinder. By the time the AIR-CYL system is invoked to perform its form of design, it has already been determined (presumably by the engineer) that the functionality demanded in the current problem will be most appropriately satisfied by an instance of an air cylinder. The design problem is to produce the most appropriate instance of this particular class of device.

A knowledge engineer creating an expert system in DSPL could encode functions as requirements, but there are no provisions that support this in the language. Representing functions in the feature space provided by DSPL, while possible, is not in the spirit of DSPL's intended field of application. Using DSPL in this fashion confuses
the distinction between these different forms of design requirements. These observations are not intended to belittle the capabilities of the language or the diminish the importance of the class of design represented by DSPL, but rather to clearly delineate its scope.

Designing to satisfy a specified functionality encompasses the most creative forms of design, where little may be known beforehand about the form of the finished device. However, not all design-to-function problems need be so open ended. Must simpler examples of design-to-function problems exist where a specific functionality is mapped directly to a known device, which can subsequently be handled by, for example, DSPL-like techniques.

2.3.1 A Simple Taxonomy of Design Requirements

The device must in some sense satisfy a set of requirements. The forms of the requirements described thus far are:

- **features**: a set of desired features of the desired device. In this case, the input to the design problem is the (implicit) constraint that the features of the output match those listed at input.

- **constraints**: a set of constraints, including
  
  - constraints on the form of the desired device, e.g., its dimensions, material, etc.
  
  - constraints relating to the device's intended operating environment.
— constraints on "secondary" properties of the desired device, such as its maintainability, reliability, supportability, etc.

— constraints on the manufacturing process of the device, including scheduling, machining, availability of parts, etc.

— constraints on the design process, that is, constraints on how the computation of the desired output is performed.

• functions: the functionality that the designed device is meant to satisfy.

Note that constraints, in their weakest form, can express any kind of input requirement. It might appear at first that representing all input requirements as constraints is an improvement because it suggests a uniform representation. Instead, reducing the diversity of knowledge types blurs important distinctions and leads to a reduction in the effectiveness of methods that might otherwise take advantage of the individual structures of each knowledge type. The lesson of global mechanisms working on uniform representations has been made apparent many times before, and is just as applicable in this case.

A design problem begins by collecting the requirements that describe the design problem. The amount of data collected during the requirements phase may be substantial, and may in fact represent a substantial portion of the design effort. The importance of this work, however, should not be underestimated. A robust understanding of the design requirements, both in form and content, can be used to advantage throughout the design process. It is useful to make the requirements as detailed as possible since this is the information needed to set up the problem spaces
for the design proper. The design requirements are initially the only clues to selecting operators to satisfy the goals of design. When a candidate device is proposed for satisfying the design, the requirements can be used to aid in verifying that the device does indeed satisfy the design goals.

2.4 The Output of the Design Task

This section presents a spectrum of the possible outputs of the design task. Often, aspects of the output are implied by the form of the requirements. That is, the mere act of describing the setup to a design problem may determine the design task's output and in some cases even the process by which the requirements are mapped to the output.

In the case of "off-the-shelf" design, described in the previous section, the output of the design task was a device that satisfies the design requirements and that was selected from a pre-determined set of known devices. In the most simple scenario, the device and all its attributes are known in advance, and the device is a "primitive" device, in the sense that it consists of no sub-devices or components. The output of the design task is the identification of an existing device; no additional elaboration of the device is necessary since the device and all its attributes are known in advance.

Another primitive example of design involves producing an instance of a known class of devices where certain parameters of the device need to be set in order to completely specify the device. The output of the design task in this case is the values of the parameters of the device.

These two cases, selection of a completely specified device from a set of devices,
and instantiation of a single class of device, are often found in some combination. A design task may require selection of one of several classes which is in turn instantiated to generate the solution to a design problem. The output in this combined case consists of the identification of the device class and the elaboration of the attributes of a particular instance of that class.

The previous situations only involve producing a single, simple, completely specified device that satisfies the initial requirements. Another step up in complexity comes when the desired device consists of an assembly of sub-components. Still, all of the devices and sub-devices are known in advance, and each must be parameterized to satisfy the requirements on the desired device. However, we now have the additional complexity that comes with insuring that the sub-devices are parameterized so that they are consistent both with the input requirements and with each other. The output of the design task now consists of a set of devices, and the relationships among the devices.

The next level of complexity is an arbitrary collection of assemblies to form a system. This may be considered just a scale-up of an assembly as the result of design; however the desired device is a complex assembly of other assemblies and devices.

The necessary output for any design task is the complete specification of a device and its attributes that satisfies the initial input requirements. An assembly or system must additionally specify all of its sub-components and assemblies, and their relationships to each other. This specification describes a physical device purporting to satisfy the functionality of the input requirements, in sufficient detail that the
engineer is convinced that the device can be rendered in the real world.

Parts of the output specification, beyond the device's functional specification, may be available before the start of design, and this impacts the difficulty or ease with which the problem may be approached. If a device, its attributes and attribute values are known before problem solving begins, then the design task becomes a problem of selecting which candidate object to use to implement the design. When attribute values of a device are not known beforehand, the design task must include some sort of parameterization. When new device classes must be generated, or when a device class is incompletely known before the start of design, then some sort of generator must be available to complete the task. In general, the less structure that is available before problem solving begins, the more difficult the design task becomes. This difficulty can be reflected in the need to traverse larger search spaces, that is, an increase in computational expense, or in the need for a larger body of design knowledge to generate a solution.

It is additionally useful to include in the output description of an artifact exactly how the artifact satisfies the functionality specified in the input requirements. The association between functionality and aspects of the device provides important clues to certain types of diagnostic and maintenance problems, as well as providing a means for indexing the device for later case-based design.

The design task so far: We are attempting to create a device to some specifications. The device may be known in advance, (ie, in some sense a “primitive” device), or consist of a collection or assembly of other devices, or (in some extreme cases) require
the generation or discovery of new primitive devices. This description does not tell us what goes on during design problem solving, only basics of its input and output behavior. The next section describes some of the representational primitives needed for the design task.

2.5 Structures That Mediate the Design Process

By examining the preceding descriptions of the input and output of design, we can hypothesize how the mapping from input to output is made, and suggest what knowledge structures might be prerequisite to completing such a mapping. Without referring to particular implementations for performing the design task, we can still get a handle on the preconditions for design. This level of description gives a view of the high-level strategy, or logic of the design task.

This section begins with a description of the representation of the devices used throughout the description of the design task. These device descriptions are used to represent the output of the design process, as well as knowledge of ancillary devices used during design and for input.

2.5.1 Representing Devices

The goal of the design tasks described in this work is the specification of a physical object that satisfies some requirements. These objects have various attributes, such as the physical dimensions of the object, the material the object is made of, the orientation and relation of sub-components, etc. The objects and their attributes are represented as structures similar to Minsky's frames [Minsky, 1975]. Each class of
object is represented in its own frame, with a particular object represented by an instance of the frame. The frame organizes the collection of attributes that describe the physical parameters of a device.

At the end of a design session, the implementation language contains a structured representation of the object that was designed in a design data base. A Design Object is a structure in the design data base that represents a class of objects to be designed. At this level of abstraction, the design process consists of instantiating and filling in values for appropriate design objects. An instance of a design object is the result of a design process, that is, it is the result of an abstract object that was instantiated and specialized by the design process.

There are several categories of object corresponding to the different roles that an object assumes in the design task.

**Primitive Objects.** A primitive object describes the lowest level of detail engaged by an engineer performing design within a domain. Such an object may correspond to a simple, physical object such as a nut or bolt. Such an object has no hidden structure or detail. Other objects may be more complex. For example, a primitive object may have significant inner detail, as does a transistor or an amplifier for an electrical engineer. In either case, the primitive object is irreducible for the purposes of the design task at hand.

The attributes that elaborate a primitive object completely specify, either implicitly or explicitly, the physical properties that describe the object. For most forms of design, the physical attributes of these objects are provided by the domain expert.
prior to the start of the design process. In more "creative" forms of design, primitive classes and their attributes may be generated as part of the design process.

The lowest level of design detail is the computation of the values of the attributes of the primitive object. It seems highly unlikely that the values of the attributes of an object are determined concurrent with the determination of the attributes themselves. Instead, enumerating the attributes of an object are likely done prior to design, if not, at least, during an earlier phase of the design process. The static representation of the primitive objects, including their attributes, need to be explicit somewhere in the design knowledge base.

Generally, the attributes of an object are indexed by, and organized around, the object that they describe. That is, the length of a rod, in some design context, will typically only be used in the context of using that rod as a part in the design process. It is unlikely that the length of the rod would be of use outside of that context. Other organizations, such as organizing attributes by their type (ie, all length attributes, or all attributes that specify some physical dimension) are rarely used and are not high priority in the representation.

The values of the attributes of primitive objects are generally synthesized from previously computed values in the design state. These values are made available in the form of requirements to the design of the primitive object.

**Assemblies of Primitive Objects.** Primitive objects are combined and collected by various means during design to form new units, or assemblies, implementing new or additional functions than the initial primitives. Assemblies are often created from
existing primitive components by a specialized design process that we call configuration.

Assemblies of primitive objects may act as new primitives in the engineer's vocabulary of design. They have their own functionality, dependent on but different from the functions of their primitive components. The attributes of an assembly are derived from both the attributes of the primitives that make it up, as well as the manner that the primitives are configured.

Although an assembly's attributes are determined from the parts that comprise it, the constraints on those features of the assembly are derived from the current design state. Requirements are inherited from parent design problems, and mapped into the requirements for design and specification of the primitives and sub-assemblies that comprise the assembly.

The detailed levels of the design process focus on the selection and specification of primitive components. The important issues revolve around how primitives are selected or generated, and how attribute values are chosen. This level of design is often very domain-specific in nature, and taken in isolation, are often relatively simple problems to solve.

At the next level of generality, the level of assembly synthesis, an additional set of problem solving issues come into play. At the higher levels the attributes of the assembly are still important, but these values are no longer directly determined. Rather they are determined as the result of combinations of results of the components of the assemblies. The problems that are handled directly by this level of design are
the mapping of requirements to set up lower level problems, and insuring that the result of primitive specification and selection can be properly integrated into the assembly.

System Descriptions. Collections of device assemblies and subassemblies to perform a complicated function or set of functions is called a system.

At this level, design descriptions primarily represent some sort of functional decomposition of the desired device.

Note that at each level of device complexity, the use of efficient, effective problem solving becomes more critical. At the primitive device level, the design task is usually well behaved, and easily handled by domain-specific methods for generating primitive specifications. At the level of assembly specification, the potential problem space involves not only the problem space of the specification of each primitive that appears as a component of the assembly, but also the selection of each primitive, the composition of the primitives into the assembly, and the verification that the assembly as a whole satisfies the local design requirements. At the system level, vast search efforts that may have been required in the determination of a system's characteristics can be condensed and compiled, and then referenced by each aspect of the system functionality. Each level of description provides a correspondingly higher leverage opportunity to prune a unwieldy search space into a manageable one.
2.5.2 Representing the Functions of Devices

An engineer may approach a problem with a particular functionality in mind, but not necessarily with any \textit{a priori} idea of a device to achieve that functionality. At the Computational level, then, the design problem begins with a description of a function, and finishes successfully when a device is presented that is capable of providing that functionality. At the Representation level, it would be useful to provide an index of known devices by the functions that they can provide. Such a representation has been suggested by Sembugamoorthy and Chandrasekaran in [Sembugamoorthy and Chandrasekaran, 1986].

Briefly, Sembugamoorthy and Chandrasekaran’s Functional Representation language represents devices by their functions. In this language, functions are the defining characteristic of a device. Devices are made up of other devices, known as \textit{components}. A device achieves its functionality by using the features of its components. How a component in a device implements its function within the device is represented separately from the specification of the function. In this way the component may be replaced by another that provides the same functionality but in a different fashion.

A functional representation of a device, when available, provides an engineer with a flexible, high level map of the tasks of designing the device. In many cases, the representation suggests a structural decomposition of the device into a set of simpler components, and defines the relationships that must hold between those components. The design of each of the simpler components can be addressed individually, and finally recombined by again using the functional representation, this time to verify
that the characteristics of the individual components together fit the functionality described.

The functions of a device may not always correspond to apparent physical components. In this case, the representation of a device may suggest less than a complete or completely detailed structural decomposition. In this case, the functional representation may provide the engineer with a functional breakdown of the device, allowing the engineer to set up several simpler design problems each addressing a portion of the overall functionality of the device. The functional representation allows the design process to shift between either a functional or structural emphasis as appropriate.

2.6 Methods for the Design Process

Once the structure of a task is understood, we can begin to examine how such a task can be carried out; by what methods and using what forms of knowledge to achieve the necessary results. This section addresses these connections between the inputs and outputs of design described so far. This is Marr’s Representation and Algorithm Level.

2.6.1 A Task Breakdown of the Design Process

One strategy for design is to propose candidates and verify that one satisfies the design requirements. The goals of proposing and verifying candidate designs are very different from the top level ones of “design artifact”. If the proposed solution does not meet the design requirements, the reason for that failure is analyzed and either a new design is proposed, or else a modification to the existing solution is suggested.
Abstractly, this strategy can be seen as a specialization and extension of the Generate and Test method of problem solving, applied to the task of design.

Each of the tasks in this breakdown can be analyzed separately, in terms of their unique needs and contributions to the design process. Each task has its own methods for achieving its portion of the design task, and each has a unique role in accessing and modifying the design state.

Proposing Candidate Designs. Given a design goal, that is, a goal of producing an artifact that satisfies some functionality or other requirements, propose a solution. This task succeeds if any reasonable candidate that addresses all or part of the requirements can be proposed, and fails when no candidate can be suggested, for whatever reason.

At this point the design of a large system may be broken down into several smaller design problems. In some cases, a method for satisfying the Propose goal may break the Propose goal into one or more new design goals, each of which must be completed to produce a solution to its own goal requirements.

The constraints on the candidates proposed at this point are rather weak. Complete candidates (ones that address all of the design requirements), or consistent candidates (ones that satisfy all of the requirements that they do address), are not necessary. Later tasks are responsible for determining if, in fact, the solution satisfies all the necessary requirements, or for suggesting modifications if requirements are not met.

A large body of work recognizable as design research is related to the propose
task, since most of the responsibility for producing finished designs falls on this task. The other tasks mentioned here support the design process, but not by producing solutions as such.

Specific methods for proposing design candidates are discussed in the next section (2.6.2).

Verification of a Proposed Candidate. Once a candidate design is available, it is verified for consistency with the input requirements for that portion of the design. If all requirements are met, then the goal is satisfied.

The inputs to this task are the partial or complete design, and the corresponding design requirements. The task may be effected by constraints that directly check the value of relevant attributes in the design. In more complicated situations, methods such as simulations can be used to decide if the requirements are satisfied. The output to the task is simply a statement of whether or not design may proceed.

Note that it may not be possible to verify that the proposed candidate either satisfies or fails to satisfy the input requirements. In this case, the engineer must decide whether to investigate some form of testing that can finally verify the candidate, or appeal to some higher authority on the matter, or perhaps suspend judgment on the candidate and request a new candidate to be proposed.

Critique of an Unsuitable Candidate. A candidate design may be discarded when it has been shown to be lacking in some aspect, provoking a new candidate design to be proposed. However, since the candidate may represent an arbitrarily
large amount of computation, and since the magnitude of the change needed to fix the problem may be relatively small, it may be useful to find out exactly how the unsuitable candidate does not satisfy the relevant requirements. The Critique of an unsuitable candidate attempts to determine where the problem lies.

The input to the Critique task is a statement of which requirements have been violated by the candidate design. The subtask is responsible for identifying what portions of the candidate are responsible for the unacceptable features noted in the verification task, and indicating how these unacceptable features cause conflicts with the input requirements.

Modifying a Design The Modification task can be broken into two parts. The first suggests possible changes to the proposed design candidate that might fix the problem identified in the Critique Task. The second part implements one or more of the changes and, in effect, proposes a new candidate based on those changes.

The input to the Modification task is the partial design state along with a description of portions of the candidate that are in conflict with the input requirements. These are essentially the output of the Critique task.

A general method for design can be composed from these four tasks. To wit:

- **Propose:** Propose a candidate design. If successful, proceed to next step, otherwise terminate the design process with failure.

- **Verify:** Check whether or not the proposed candidate satisfies the input requirements. If so, then design completes with success. Otherwise, proceed to
the next step.

- **Critique:** If the verification finds a problem, then find out exactly what is the cause of the problem. If the nature of the problem is found, proceed to the next step, otherwise terminate with failure.

- **Modify:** Attempt to make an appropriate change to the candidate and if successful, return to the verification step. If no change can be implemented, then terminate with failure.

This is the most basic form of the Propose/Verify method for design. Each of these tasks may be broken into more detailed subtasks to provide more complicated variations on this basic theme. One of these variations is described in section 2.6.3.

A design plan may compile variations of these subtasks into a single monolithic structure. For example, a simple constraint embedded within a design plan may play the role of design verification. A general-purpose method for design like constraint satisfaction blurs many of these tasks into a single mechanism. Here again, the constraints fill the role of verification, and the underlying mechanism (implicitly) performs a limited form of the *Critique* and *Modification* tasks.

Additional task breakdowns may be useful. When a subtask is generated for the design of a particular subassemblies, for example, how are the specifications for the new subproblem derived from the parent problem? This is often a sizable problem in and of itself, so mapping specifications from one level to another is distinguishable as a subtask itself.
2.6.2 Methods for Proposing Design Solutions

The Propose task is responsible for generating potential design candidates. Methods for proposing design solutions map input requirements into these design solution.

Design Plans. A plan is simply a sequence of actions, the results of which purport to achieve some goal. A design plan in the context of the Propose subtask is a sequence of actions whose execution results in the generation of a candidate design.

Design plans, even in the narrow context of proposing candidate designs, cover a wide range of detail. Plans provide the means to implement a task decomposition as a set of concrete actions to perform. A high-level plan may implement a task decomposition for a particular scheme for generating a design candidate. In the AIR-CYL system, for example, the top level plan for designing an air cylinder calls for performing a rough design task before proceeding with the detailed design of each of the major components of the air cylinder. Such a plan may not directly affect the design state, but is still important for the organization it provides for the lower levels of the design process that do make changes to the partial design.

Eventually, design plans must bottom out in actions that change the state of the partial design. Either the value of an attribute of a component is proposed, or some component is selected.

All of the methods mentioned in this section are generally indexed by the goal that they are intended to achieve. Plans are also indexed by the results that they are expected to produce. The Propose task typically requires plans indexed by a specific
component or device (i.e., for designing <part>, use <plan>), or some function as described in the input requirements (i.e., for achieving <function>, use <plan>). Plans are additionally associated with knowledge that aids in determining the usefulness of the plan in the particular design context.

Note that plans are also a useful mechanism for representing methods addressing any subtask of design. The task of verifying the acceptability of a device, for example, might call for the analysis of a simulation of the device. This could be handled by a plan that first sets up an appropriate simulation of the device, and then analyzes the results of the simulation. The plan implements a task decomposition specific to the verification of the device being designed.

Basic Design Step. In the terminology of Brown and Chandrasekaran [Brown and Chandrasekaran, 1986], a step is the primitive knowledge type that computes and sets the value of a single attribute in the partial design state. This is one form of knowledge for representing the detailed actions that eventually comprise plan sequences.

The form of the knowledge in the step is often a mathematical calculation, but need not be. It is almost certainly a closed, deterministic computation.

Plan Generator Methods. Many methods exist that are capable of transforming various kinds of domain knowledge into a design plan that subsequently produces a candidate design.

For example, an assembly may be statically described as the list of parts that make up the assembly, that is, its structural decomposition. A method employing
a divide and conquer strategy could use this knowledge to directly generate at run time a list of subproblems to work on corresponding to the parts that comprise the assembly.

A *functional decomposition* of a device is a declarative description of how a function may be implemented by a set of components or the interactions of functions of subcomponents and assemblies. Such a structure can be used to determine what parts need to be included in the finished design, how they interact, and give an indication of where constraints on the finished device are derived. These subtasks of the *Propose* task can be systematically generated from a functional decomposition of a device by using the syntax of the functional representation to trigger the generation of the appropriate subtask types. Generating a specific method at run time by examining the functional decomposition is an example of a *Meta-method*, that is, a method whose job is to produce a method for, in this case, producing a design candidate. The subproblems generated may be one of the tasks discussed in the previous section, or may be new design tasks.

Once we break a problem apart, we need to synthesize the parts into a complete solution. However it is not true that the decomposition of a problem into several subproblem necessitates significant problem solving effort to recombine the results back into a complete solution. The decomposition task may be organized such that the individual subproblem solutions only require minimal compatibility checks to insure that the complete solution is workable.
Case Retrieval and Adaptation. Another broad class of methods for producing candidate designs, called case-based design, involves selecting and modifying an example of a previous attempt to solve a similar problem. Experience plays an important role in this approach, that is, previous designs and attempts at designs are used to guide current attempts.

The process of case-based design must handle several broad issues, including the indexing of existing cases, case selection and case modification. Existing cases must first be organized so that relevant cases can be retrieved when a new design problem is presented. Case selection then uses the requirements to the design problem as an index for retrieving cases relevant to the context. Once an appropriate case is selected, it must be modified until it satisfies the current requirements. Once the design is complete, the new case is filed away in the growing case history.

Knowledge-Weak Methods. Many mechanisms exist for producing design candidates which, while not satisfactory for design in the large because of complexity problems, are nonetheless suitable for tackling smaller, well-defined pieces of the problem.

Included in this category are traditional constraint-based problem solving mechanisms, linear programming, and other optimization techniques.

Note that any of these methods suggested for proposing design candidates can lead to task decompositions in their own right. See, for example, KRITIK, for case-based task analysis.
2.6.3 The Trade Study Method For Design

To date, most expert systems that purport to perform design within some domain use only a single method to generate candidate solutions. The requirements to a particular design problem given to such a system will typically map into a single representation, i.e., a single problem statement, and although a potentially large space of designs or partial designs may be considered, only a single final “solution” is offered as satisfying the initial design requirements. Methods may exist other than the one used that produce equally acceptable or even superior but significantly different solutions to the same initial problem. As we consider more complicated design tasks, and include a wider range of methods to perform design, the methods for design will begin to overlap in terms of what the output of the method provides. Several methods may be available to solve the same design problem, or some portion of the design problem. The resulting candidates may all satisfy the requirements, to varying degrees, and with different emphasis.

To further complicate matters, there may be no a priori means of deciding which method will produce the best design.

A similar situation occurs when an engineer performs a sensitivity analysis. In this case, an engineer applies slightly varying initial conditions to the same method and observes the effect on the final design. In this way the engineer “feels out” the design space, noting how the inputs effect some aspect of the design and finding where an optimal solution may be found.

In either case, the quality of the final design is improved by trading off the ad-
vantages and disadvantages of the candidates and choosing one that best satisfies or exceeds the initial requirements. Instead of simply proposing a single candidate design and verifying its suitability, as suggested in the previous section, several candidates obtained by potentially diverse means are compared. This method for design is a study in design tradeoffs, or the Trade Study\textsuperscript{2} method. The Trade Study method is an alternative method to the Propose/Verify method for design. It may be considered to be an extended version of Propose/Verify.

Propose Candidate Design. Instead of generating a single candidate design, several equally viable designs are proposed. The designs may be produced through different methods for design (case, plan, etc), or through similar methods with different parameters. At one level, the strategy of producing multiple designs can be justified when several methods for design are available but no clue exists as to which method will produce the best (ie, nearest to optimal) results. Producing only a single design that marginally meets the design specifications (as the Propose/Verify method might) may miss important points in the design space.

Verification. Each of the design candidates may be individually verified as before to insure minimal conformance to the design specifications. Additionally, the candidate designs may be compared to each other to resolve less well defined design specifications, such as "low cost" or "high reliability." This may be necessary when, for example, reliability metrics are available for design candidates, but explicit "design

\textsuperscript{2}This scheme was suggested to us by our Boeing contacts on the RAMCAD project, and is outlined in [Boeing Computer Services, 1988].
for reliability" knowledge is not available. The only recourse is to generate several candidates and compare the results.

The Trade Study method performs a similar role in the design process as the Propose/Verify method, using similar forms for requirements and producing similar outputs, but works to control a larger search space to achieve its results. Even when the exact form of verification knowledge is unknown, we can nonetheless force the domain expert to think explicitly in those terms and enter knowledge accordingly. Likewise, the form of the knowledge for comparing alternative design candidates may also be unknown, but we can force the domain expert to explicitly consider that issue.

Note that comparing the results of design has a task slot in the analysis, but that this type of knowledge is stubbed out. A theory of comparing partial designs based on the design requirements is no small task (sic) and is outside the scope of this dissertation.

An alternative to the "Propose/Verify" method for design, but at the same level in the design analysis, is the "Trade Study" method. Like all methods, the method has a sponsor to determine, at run time, the method's applicability to the run time context. The applicability of the Trade Study will be high when some sort of optimization is called for and recognized by the sponsor, for example when "Minimize cost" is a design specification. The Trade Study method is a strategy appropriate to handle this kind of situation.

Any method for design may be selected at the user's discretion at run time when multiple strategies are available for proposing candidate designs. The user may either
select a particular method for proposing a candidate design, or elect to set up a trade study of several methods. If no knowledge is available to compare the multiple results of design, DSPL++ comes to the user for help, just as it does when any hole in the knowledge base is discovered.

The Trade Study method does not merely supplant the Propose/Verify method, although it is a generalization of it. Although the two strategies interact at certain points, they are two clearly distinguishable strategies for design, applying to distinguishable situations in the design task.

In general all design should involve convincing oneself that alternative designs have explicitly or implicitly been considered.

2.6.4 The Design of Primitive Components

Eventually we need to address the design of an object that cannot be decomposed, ie, a primitive component. Some of the issues in design at the higher levels still hold at this lower level, for example, we still need to worry about providing specifications to the design process, deciding on some aspect of the design, and incorporating the results into the design context.

Several methods suggest themselves for completing the specification of a primitive component that has been included in the design state. An instance of an off-the-shelf part, that is a completely specified part, may be selected by the design process, in which case the attributes of the component are already completely determined. An instance of a previously design component may be used as a starting point and modified to suit the current design situation, just as at higher levels in the design
process a previously completed design case may be adapted as a proposed solution to a complete design.

Knowledge may be available as formulas for setting individual attributes of the component. In Brown and Chandrasekaran's routine design ([Brown and Chandrasekaran, 1986]), a primitive component is parameterized by applying a design plan composed of steps, each of which sets a value for one of the component's attributes. The attributes are designed in a fixed order, depending on the ordering of the plan, although suggestions embedded in the plan at various points controlled a redesign phase provided a flexible means of modifying the order of design depending on the needs of the design situation.

The initial ordering of attributes for design does not need to be as rigid as suggested by the design plans in DSPL. Much information about the relationship between attributes is already embedded in DSPL steps, and the ordering of the steps in the design plans is implied by input requirements of the steps themselves. That is, most steps must be run in a certain order in order to satisfy the input requirements of the steps. In fact, it is possible to almost completely order the steps in AIR-CYL simply by looking at the input dependencies of each of the steps. The steps need to be ordered so that its input is always available from previous computations. The ordering in the plans can be viewed as simply a compiled form of this information.

In fact, almost of all the DSPL plan behavior both during design and rough design phases can be reproduced by taking advantage of the dependency information in the steps.
Weaker forms of specifying attributes values for a primitive component are possible when less is known about how to select attribute values. Relations between attributes may only be available as constraints on the attribute values, instead of closed form calculations for determining attribute values. In this case, initial attribute values may be generated by inspecting the form of a constraint, and a complete solution generated via some form of constraint propagation. Such weak forms of problem solving are acceptable for the smaller search spaces associated with the design of a single component, but work poorly as a large scale scheme for design where the large number of interactions and large search spaces are unmanageable.

An "intelligent" engineer will not likely be locked into any single mode of value generation. Instead an engineer proceeds with whatever knowledge is available, applicable and expedient in the local design context.

2.6.5 Handling Problems During the Design Process

It is unlikely that any interesting and sufficiently complex design domain will be able to be understood and explored to the point that the design process proceeds monotonically from beginning to end. In any complex task the process will move forward, falter and fall back, before finally proceed to a conclusion. Inevitably, decisions are made during design that lead to a state that is found to be unacceptable or otherwise incompatible with the initial design requirements in some form. Mechanisms must exist that are capable of dealing with the failures and impasses that arise during design.

The ability to handle impasses during the design process consists of several parts.
First, an impasse must be recognized during problem solving. Second, an appropriate problem space must be set up. Third, the conditions for repair of the problem must be known. Fourth, regresses must be avoided by detecting when an impasse is unsolvable. Fifth, when an impasse is corrected, the old problem state must be resumed. Sixth, when an impasse is unable to be corrected, that fact must be handled in an expedient fashion.

Each of these stages may play a more or less important role in resolving a design impasse, and each may be more or less difficult to handle depending on how and where the impasse arises. On one occasion an engineer may easily recognize an inadequate design, such as when the overall length of an air cylinder exceeds the allowable length specified by the input requirements, but be stumped as to how to proceed in reducing that overall length. In this case, the overall length of the assembly is easy to compute and compare with the corresponding input requirement, while the relationships between the components of the assembly and their interactions makes a corrective action less obvious.

In another situation, a design inadequacy may be extremely difficult to detect, while, once detected, relatively easy to correct. In the case of the air cylinder, the overall reliability of the finished product is difficult to ascertain without actually implementing the design and exhaustively testing it. However, a weakness in the air cylinder assembly, such as an under-designed tube wall, can be easy to compensate for by simply increasing the tube wall thickness or changing the composition of the tube.
Intuitively, in their most general form each of these stages of failure handling are open-ended problems. Each stage, in a general context, is a problem domain unto itself, and could be approached by any number of different techniques. General-purpose failure handling strategies are likely to provide little direction in coping with impasses.

Fortunately, the design task analysis gives considerable clues as to the nature of failures that may arise during the design process as well as methods for handling them. Rather than use a global mechanism, such as dependency-directed backtracking, we can employ situation-specific methods within each different subtask of design. Each different subtask within design may have its own mechanisms for handling impasses, and even its own semantics for failure and success. Failure methods are not globally applied, but rather attached to the methods where they are appropriate.

Many examples of situation-specific methods for handling impasses during the design task are suggested by the design task analysis. The Critique and Modify subtasks of the design task may be seen as parts of situation-specific methods to handle certain design impasses that occur at the top level of design. Only when a candidate design has been shown to be lacking during the Verification subtask are the Critique and Modify subtasks engaged, and only then when knowledge about how to criticize and modify the candidate design is available.

Another example of situation-specific handling of impasses occurs when a method for proposing a design candidate completely fails to generate a candidate within the Propose subtask. The Verification subtask is irrelevant since there is as yet no candi-
date to verify. Instead, within the Propose subtask, the selection of a design proposer is retried, in order to determine if an alternative method of creating a candidate is available. The design context is essentially identical to the situation when the Propose subtask was started, except for the additional knowledge that previously selected design proposer has failed to generate a candidate. The reasonable course is to reconsider each method for proposing a candidate and to again select the most appropriate method for generating a design candidate.

Note that within a particular subtask, a uniform mechanism such as dependency-directed backtracking may be entirely appropriate. The argument is not against uniform mechanisms per se, but rather against the use of such mechanisms between different levels of the design object hierarchy and across multiple tasks, especially when different methods of problem solving are used within each task. The force of the design task analysis is its illumination of the diverse faces of the design task. Methods of problem solving that encourage or require uniform representation schemes tend to dilute and blur the task analysis' useful distinctions.

### 2.7 Classes of Design Problems

The preceding sections describing the nature of the input, output, and structures available to the design task suggest that design problems may be cataloged along several dimensions, based on how the design problem is set up, what is requested of the design process, and what knowledge is available to the design process prior to the start of design.

The specification of the device resulting from design necessarily consists of de-
criptions of at several levels. The output of the design task includes the complete specification of the desired device and all of the device's components, the relationships between the device's components, and the details of the components' structure down to the level of primitive components. The primitive components are particularized by completely specifying the values of all of its relevant attributes. In some cases, an account of how the functionality of the device satisfies the required functionality and constraints is also included in the output. Each of these parts must be available in some form as the output of the design task.

The antecedents of each of these forms of output describe a different class of design problem. In simple design tasks, they may be pre enumerated. For example, in certain forms of "design", the primitive components, their attributes, and the possible values of the attributes are known in advance, and the design process consists of selecting the appropriate combination of primitive components in the correct relationships to arrive at the final design solution. Problems of this type are often known as configuration tasks.

The next step along this scale, the primitive components and their attributes may be known, but values may need to be generated to "parameterize" the primitive components before they can be incorporated into the actual design. In this situation, objects are not simply being combined in novel ways, as in the previous task of configuration. Instead, new members are instantiated from existing classes of components. As a simple example of the configuration case, an engineer may select a bolt from a set of standard sizes. This case corresponds to an engineer "parameterizing" the
class "bolt" by deciding that bolt of length of 3.6 inches is called for, a length that may not correspond to any standard size known by the engineer.

Still further along this scale, an abstract description of the primitive components may be available, which must be elaborated to fit the design. The elaboration of a primitive component might include determining what attributes of the component are appropriate to the current problem, in addition to selecting values for those attributes.

Finally, the classes of the primitive components themselves may not be known, or only partially known, in advance of the design process. In this case the primitives must be generated as design proceeds. Note that this is likely, but not necessarily, to be a very open-ended form of design. It seems reasonable that this process, the creation of completely new primitive components, does not happen often in the context of design per se. Consider the a simple transistor as such a primitive device. Certainly one might postulate the ability to generate such a primitive device while attempting to design a radio, but more likely such primitives are handed to the design context, rather fully developed, while generated in the context not of design but of, for example, scientific exploration. Still, the process of generating primitive components during design must be taken into account.

All of the design problems described here either implicitly or explicitly involve the production of a device that meets some functionality, within some specified constraints. The degree to which reasoning about the functionality of the desired device plays a role in the design process suggests another dimension for categorization of design problems.
2.7.1 Search and Problem Spaces in Design

Note that each of these "dimensions" of design problems suggest a space in which to search for possible solutions to the design problem. In the case of design by configuration, where the design begins with a pre-enumerated set of primitive instances, the abstract search space is one where combinations of primitives are selected as solutions, or partial solutions, to the design problem. In the case of discovering new primitives, on the other hand, the space may not be so well defined or easily traversed.

No mention has been made of how the problem space, corresponding to each of the major categories of design problems, should be traversed. This suggests a second level of categorization, one that describes how the space is traversed, what knowledge is used, and how the knowledge is applied.

Note that any of the subtasks of the Propose/Verify process may resort to more or less compiled strategies for achieving their goals. In DSPL, for example, the Critique task is handled by a pre-compiled Failure Handler hierarchy, a sort of decision tree that takes as input the design state represented by a string of nested failure descriptions, and produces as output a decision as to whether or not some form of redesign (a Modification task) should be attempted. Modification in DSPL is handled by a combination of compiled and search techniques. A list of suggestions is compiled into each agent. This list is retrieved from an agent when the agent fails. At this point, a variety of search techniques, such as LEAST-BACKUP, are employed to determine exactly which suggestion should be applied to the partial design state.

Other work by Brown [Brown and Breau, 1986] investigated the possibility of
creating suggestions on the fly by analyzing the construction of the constraint that produced a design failure. Here the compiled list of suggestions in DSPL is replaced by a mini-problem space tailored to the Modification proposal task.

2.7.2 Experience and Problem Solving

Certainly, an engineer's skill at performing design depends in part on his or her ability to draw on previous experiences, both successful and unsuccessful, to guide the design process. Successful designs are modified to suit new situations, strategies for generating designs are reused, and unsuccessful designs are avoided. Each new success builds the engineer's repertoire of designs.

We know that experience alone is insufficient. New situations are encountered during design that don't quite match the designer's experience, forcing him to figure out on the fly what to do next. Thus problem solving abilities are a necessary part of the solution to the design puzzle.

Any single method for design has its shortcomings. The exclusive use of problem solving to do all design is not just ambitious but intractable. New situations will routinely arise that thwart even the largest pools of experience. Neither ability alone, problem solving or recall of previous experience, are sufficient. A successful designer must be able to take advantage of both strategies, shifting back and forth between them as dictated by the situation.

Most of types of design knowledge are the result of previous design or problem solving experience. A design plan, for example, is a record of the actions taken to solve a previous design problem. We are not particularly concerned at this point with
the origin of such records, but primarily with the types of knowledge available during design and its use during subsequent problem solving. It is entirely reasonable to postulate that certain forms of knowledge used in design themselves arise out of some problem solving activity. However the majority of design must be experienced-based, and the synthesis of new design knowledge relatively rare.

From a system building point of view, the highest leverage is gained from an architecture’s ability to represent and apply a wide range of knowledge forms, rather than focusing on interesting but uncommon behavior. If the generation of such forms indeed becomes an important issue in a particular domain, than an open-ended, goal-pursuing architecture should easily accommodate the necessary extensions.

2.8 Summary

This chapter describes an information processing view of design problem solving. From this description it becomes clear that, even while carefully limiting the scope of “design”, design is still a large and complicated task. No single “method” can possibly account for the diverse behaviors found in the execution of the design process. The process of design is made of a range of recursively invoked subtasks, each with its own individual purpose and knowledge structures and methods. By analyzing each of these subtasks of design, their interactions and relationship to the goal of design, we can begin to piece together the mosaic of activities that make up design.
CHAPTER III

The Kernel Architecture

This chapter describes the kernel architecture of the DSPL++ system in detail, including the descriptions of the underlying goal-processing mechanism, the structure of the design goals and methods upon which the design-specific shell is constructed, and the abstract devices used to describe the primitives of a design domain. This chapter also discusses the effects of design failure and how failures are handled in various contexts.

3.1 Overview

At the top level, the DSPL++ architecture consists of four parts: a knowledge base of design knowledge, a goal tree, a goal interpreter, and a design data base that records the evolving design. The knowledge base contains the static design knowledge of the particular design domain as well as all default knowledge about the design task itself. Both the goal tree and the description of the evolving design in the design database contain dynamic data structures generated during design. The goal tree reflects the state of the control process during design problem solving, information that does not directly describe the artifact being designed. The artifact itself is described the design database. This level of the DSPL++ architecture is illustrated in Figure 1.
The knowledge base is a library of methods for design.¹ These methods define the design process. There are a number of different types of methods, each type representing a different form of design knowledge, and each having a unique strategy for using its particular form of knowledge. During design, methods are retrieved and applied to the design state, changing the design state.

The design process commences when a goal representing a design problem is presented to the goal interpreter, and attempts to satisfy the goal. Goals are satisfied by methods retrieved from the knowledge base. The application of methods during the design process is organized by the subtasks of the design process. Each subtask of the design process is represented by a different type of goal, and each goal is pursued by the application of methods appropriate to that type of goal.

¹The term “method” does not refer to an abstract strategy. Throughout this work, the term “method” is used to denote a particular domain-specific piece of procedural knowledge used to satisfy a goal.
Methods use a design database to store the results of the design process. This description is organized as object descriptions that are instantiated during design. Object representations in the design database possess attributes whose values are set and examined during design. The database model used in DSPL++ is an extension of the design database found in DSPL [Brown, 1984]. The DSPL++ design database includes both the descriptions of objects being designed as well as the input specification for those objects.

At the core of the architecture is a goal processor that processes design goals as they arise during the course of design. Active goals are maintained by the goal processor on a goal tree. During problem solving, the goal processor cycles through each of the goals in turn, examining each goal, finds any relevant elements in the design knowledge base to help in satisfying the goal, applies that knowledge to the current design state, then checks to see whether or not the current goal has been satisfied. During each cycle goals may be added to the tree to deal with subproblems that arise, or removed from the tree as goals are satisfied.

The goal processor itself is a general purpose mechanism and does not represent design problem solving behavior in any particular fashion. Instead, what gives the overall system its design behavior is the nature of the goals that are processed and the methods used to satisfy them. The types of goals, their processing methods, and the interactions among the goals during design are an important focus of this research.

The design cycle is recursive. During the design process the goal processor retrieves methods in the knowledge base to satisfy the design goals. These methods
may in turn generate new goals to be handled by the goal processor. Control of the
design process is shared between the goal processor and the methods retrieved from
the design knowledge base. The goal processor manages the goal tree and applies
the relevant methods from the design knowledge base, while the methods determine
what actions to take to satisfy the design goal for which it was activated, including
deciding when the generation of subgoals is necessary. When a new subproblem is
recognized, the goal processor again enters the loop, instantiating the new subgoal,
retrieving new methods and beginning the design cycle over again.

Design problem solving involves a very broad range of activities. The implementa-
tion of a working design system must incorporate a correspondingly broad range
of representations in order to accurately model the process. Thus both the task and
method representations in DSPL++ cover a wide range of activities and concepts,
from the lowest level required to implement the system primitives on which the sys-
tem is built, up to the broadest description of the design task and the top level
strategies for achieving design.

The reader will note similarities to aspects of SOAR [Laird et al., 1987], BB1
[Hayes-Roth and Hewett, 1988], TIPS [Punch, 1989] etc., as the DSPL++ architecture
is unfolded. The flexible use of multiple methods is a goal that the knowledge-
based system area as a whole has been moving to over the last several years. The
DSPL++ architecture occupies a niche in a space of possibilities for such architectures.
Chapter VI contains a discussion of the points of comparison and departure with other
architectures.
The following sections describe each of the major pieces of the DSPL++ architecture. The description ranges from the low level architecture, the design independent portions of the architecture on which the design dependent portions are layered, to the portions that are specific to the design task, in order of generality. We begin by describing the underlying goal and goal-processing mechanisms that form the framework within which the control issues of the design task are later discussed.

3.2 The Knowledge Base

The first major component in the DSPL++ architecture is the knowledge base. The knowledge base is a method library, a collection of information about both the domain in which design occurs as well as the design task itself. Methods in the knowledge base are indexed, based on various characteristics of the method. Methods are retrieved from the knowledge base either by providing the name of a specific method, or by listing a set of desired characteristics of a method. In the latter case, the characteristics are translated to a suitable index and any methods matching the index are retrieved.

Indexing. The relevance of a method to a design context is determined by how the method is indexed. Each method in the design knowledge base is indexed so that it can be later retrieved at the appropriate point in problem solving. Portions of the index are determined by the knowledge engineer when a new method is created, while other portions are created automatically based on the new method's characteristics. Automatically generated indexes include indexes for
• the method’s name,
• the method’s type,
• the role that the method plays during the design process, and
• the domain of the method, if the method is domain-specific.

Other indexes created depending on the type of method are described as the various types of methods are introduced. Indexes may also be added at any time by the user.

Referencing Methods Directly. Methods in the knowledge base may also be retrieved by referring to the method’s name. Not all methods are referenced solely by the components they design or the goals they potentially achieve. The use of a particular method may be precompiled at a particular point in problem solving where it is known to be needed. This situation occurs most often in plans where the actions to be taken, and hence the particular pieces of knowledge implementing these actions, are known in advance.

3.3 The Goal Interpreter

At the core of the DSPL++ architecture is a simple, relatively domain independent, goal processing controller. Its function during the normal course of design is to note which goal is next to be processed and to make that goal the focus of problem solving. The goal interpreter is responsible for several functions during problem solving, none of which are especially specific to the design process. They are the spawning and termination of goals, both during normal design and during failures that arise during
design, and the ability to allow user tracing of the design process. Tracing facilities and other features of the user interface are described in Appendix B.

Problem solving begins when the interpreter is handed a single goal to process. Typically (since we are describing a system for design) the top level goal describes a design problem to be solved. As such, this top level goal includes some specification of a device. The semantics of the goal imply that some version of the device is to be synthesized in some form. Problem solving stops when the device is successfully synthesized, or when it is recognized (through various means) that the device cannot be fashioned.

Goals are processed one at a time, beginning with the single goal that initiated problem solving. Processing a goal may generate subgoals, which may or may not be immediately processed. When a subgoal begins processing, the current goal is suspended and the subgoal becomes the new active focus of problem solving. The new current goal is processed by the interpreter until it either terminates with success or failure, suspends while new subgoals are processed, or sleeps while the parent goal continues processing. A goal that has been asleep can be later awakened by its parent goal. Goals that have terminated with failure may be later retried by the parent goal. Further details of valid goal transitions recognized by the DSPL++ architecture are described in Appendix C.

Goals that have terminated, either with success or failure, are marked as such and are removed from further consideration. A record of all goals, regardless of their state, is maintained for later reference by explanation facilities.
The order in which the interpreter processes goals is partially dependent on the order in which the goals are created, and partially on directions from methods activated by the interpreter. The default behavior of the interpreter is to simply process the goals in the order they are created, and to resume processing in the parent goal of a goal that terminates processing.

Very little problem solving knowledge is directly available to the goal interpreter. Instead, the interpreter relies on methods retrieved from the knowledge base to determine its behavior.

3.3.1 Standard Goal Behavior

During normal operation, the interpreter is driven by a list of commands provided to it by the currently active goal. Each goal has a *command stream* that is read by the goal interpreter in order to determine what actions the interpreter must take in processing the goal. The items placed on the command stream define the behavior of the goal as it is being processed by the interpreter.

The commands on a goal's command stream account for all of the actions of the goal. For example, how and when methods are retrieved from the knowledge base is determined by commands placed on the goal's command stream. Methods applied to the currently active goal are processed by putting their own commands into the command stream for processing. The range of possible commands for goals is further described in Section 3.7 below.

The exact behavior that is produced is dependent on the type of the goal being handled by the interpreter. Different design goals have different behaviors associated
with them. Note that these design goals and their behaviors represent the knowledge about the design task within the DSPL++ system. Section 3.8 describes how these behaviors are represented and attached to goals. The descriptions of the different design goals and their behaviors in DSPL++ are saved for Chapter IV, after the descriptions of the generic mechanisms are complete.

3.3.2 Handling Abnormal Goal Behavior

When a goal cannot be satisfied, or when no progress toward a goal can be made, there is no independent or global backtracking scheme or default reasoning scheme that the goal interpreter falls back on for recovery. This is not to say that backtracking is not allowed in the DSPL++ architecture, rather that these control issues are left up to the task and even domain level descriptions rather than by non-specific or global mechanisms.

Failures and impasses during design problem solving are handled by the goal interpreter in the same fashion as any other task that generates subtasks. An appropriate goal is instantiated, in this case, a Resolve Failure goal, and that goal is given the responsibility of retrieving any relevant recovery knowledge from the design knowledge base to recover from the impasse. The Resolve Failure goal does not itself rely on any global recovery mechanisms, but rather looks to the method that was controlling the design process when the failure occurred for direction on how to recover from the failure. The strategy in DSPL++ is to provide task or even domain-dependent "hooks" into the architecture for mechanisms to handle exceptions that arise during the design process. Mechanisms at this level eliminate the need to attempt to devise a global
mechanism that attempts to handle all exceptional situations in a uniform manner. Further descriptions of the particulars of failure handling methods in DSPL++ can be found in Section 3.9.

3.4 The Goal Tree

The goal tree is a collection of goals dynamically maintained by the goal interpreter. The goal tree is a trace of the flow of control during design, and serves to organize the state of the design problem. The normal behavior of the interpreter is to process goals in the order they are added to the goal tree, resulting in a predominately depth-first traversal of the goal tree by the interpreter as the goal tree is created. There are exceptions to this pattern, for example during the handling of failures, when goals that had previously failed to achieve their objectives are revisited in a subsequent attempt to get results. Goals may be placed on the goal tree either by external sources, as at the beginning of the design process, or as the result of subgoaling behavior during design.

The goal tree is an extension of a simple stack. In a simple stack, items are simply pushed onto and popped off the stack as they are processed. In DSPL++, new goals are placed on the stack, either singly or in clusters, by methods retrieved from the knowledge base. Each method operates independently, manipulating its own local goal stack. For example, the goal at the root of the tree defines the problem that initiated problem solving, and that causes all subsequent problem solving to be invoked. This goal is at the top of the problem stack, and when it is successfully terminated, problem solving is complete. The top level goal generates a set of subgoals
Design a Widget

Propose/Verify Method:
- Propose Candidate
- Verify Result

Adapt Widget Method:
- Retrieve some Widgets AND
- Select a Widget AND
- Adapt the Widget

Figure 2: The structure of a DSPL++ goal tree
through the application of a method. These goals have the entire context of the parent goal available to them, but are processed within their own narrower context. In this situation, the parent goal continues its processing only after all of its method's subgoals have been successfully terminated.

Figure 2 provides an illustration of the goal tree concept. At the top of the problem stack is the goal "Design a Widget." When this goal is complete, problem solving is complete. This goal has two subgoals, “Propose Candidate” and “Verify Result”, both generated through application of the “Propose/Verify” method. Both of these goals must be satisfied before the method is successful and the parent “Design a Widget” goal can be satisfied. The “Propose Candidate” goal in turn has generated several goals through another method, all of which must complete before the “Propose Candidate” goal can complete.

In other cases, several methods may be available from the knowledge base, any one of which may be able to produce a suitable solution for the parent goal. This situation is illustrated in Figure 3, where three subgoals, G1, G2, G3, have been generated from goal G through the application of the “Fit Widget” method. In this case the method indicates that any one of the subgoals completing successfully can satisfy the parent goal, and the method within the parent may continue processing. In this case the goal tree can be seen as an and-or graph of different types of problem tasks, with each task utilizing its own form of knowledge and control suitable to its own task within the overall design problem. This analysis has been previously noted by Chandrasekaran in [Chandrasekaran, 1989] and [Chandrasekaran, 1990a].
Propose/Verify Method:
- G: Propose Candidate
- AND
- Verify Result

Fit Widget Method:
- G1: Adapt Widget #27
- OR
- G2: Adapt Widget #345
- OR
- G3: Adapt Widget #52

Figure 3: An “and/or” goal tree
Both of the situations illustrated above imply a need for a mechanism within the architecture to choose and apply appropriate methods for satisfying goals. Both method retrieval and selection are handled by the architecture through the generation of specialized subgoals. The methods for satisfying the selection subgoal are similar to those used by Brown in DSPL [Brown, 1984] and later by Punch in TIPS [Punch, 1989], and are discussed in detail in section 3.7.

The Current Problem Context. A design context is defined as the information in the goal tree on a path from the currently active goal through the top level goal. This represents the design information that is available to the currently active goal.

To illustrate this concept, consider, the design of a complex system such as shown in Figure 4. The automobile has many subsystems, e.g., brakes, an engine, drive train, etc., each with their own attributes, and each considered during design in relative isolation to each other, but always in relation to the system in which they are contained. During the design of the brakes, the details of the engine are of little concern, only the overall characteristics of the automobile are relevant outside of the brake details itself. Thus, the number of cylinders in the engine is outside the context of the brake shoe design problem, but not total weight of the car. The current context serves to illustrate which parts of the design are immediately available during the design process, and which aspects are only remotely available.

During problem-solving, this cross-section of the goal tree behaves much like a simple stack, with items always added and removed from one end. The difficulty with actually implementing the goal tree as a simple stack is that previously completed
goals cannot be restarted, since they are popped of the stack on completion, and no history is available for explanation.

The flow of control allowed by the goal interpreter over the goal tree is determined by the goal tree structure. Control between subproblems moves vertically up and down the paths defined by the goal tree, and laterally among the goals local to a problem space.

The information in the goal tree differs from the information in the design case in that the former does not directly describe the features of the finished artifact, but rather with how those features were determined.\textsuperscript{2}

\textsuperscript{2}Many design methodologies are beginning to recognize the importance of this "design history" in the maintenance and documentation of a design. See, for example [Boeing Computer Services, 1988] or [Richter, 1987] for proposals for representing and using a "design history" in this fashion.
3.5 Goals and Goal Classes

The DSPL++ architecture describes the design process as a set of goal classes. Each goal class, such as the Design Artifact goal class or the Propose Candidate Design goal class, plays a specific role in the design process. During problem solving, the interpreter operates on particular instances of these classes, particularized to a specific design problem.

A new instance of a goal class is created through a process called instantiation. The instantiation process creates a new goal and parameterizes it according to the current problem context. This process is handled by the goal interpreter, usually under the direction of a method being applied to the currently active goal.3

The term "goal" in this work typically indicates an instance of a design goal class, that is, a specific goal such as "design a sieve tray4 to this set of specifications", rather than the abstraction of a goal, such as "the design of sieve trays" or even "the design of physical artifacts". Only the fully instantiated version of a goal can be reasonably addressed in a routine design situation. The more general version correctly describes a goal class, not an instance of a goal.5

Our current work uses the information saved in the goal tree only to manage the design process. At the end of the design process, the engineer is allowed to optionally save the goal tree along with the case as an annotation to the completed design. In [Herman et al., 1986] we use this type of history for a form of design explanation by using it to justify both features of the finished design as well as intermediate steps in the design process.

3 The terms “class” and “instantiation” describe concepts taken directly from Xerox LOOPS (Lisp Object-Oriented Programming System), in which the entire DSPL++ system is implemented.

4 A sieve tray is a component of a distillation column.

5 In discourse however, it is often easier to simply refer to “the Design Artifact goal,” when what is really meant is either an instance of the Design Artifact goal class, or simply the Design Artifact class itself. Once the basic concept is understood, the distinction is often either unimportant, or
A goal can be viewed as problem solving context. Knowledge about how to handle a goal's portion of the design problem may either be attached to the goal class, indicating a generic means of solving a type of design problem, or may be attached to the goal itself, an instance of a goal class, indicating how a particular instance of a problem may be solved.

All DSPL++ goals, except the top, are linked into the design context by pointers to the parent goal, and pointers to any sub-goals that may be generated during design. These pointers are managed by the interpreter when new goals are instantiated.

Goal classes systematize the retrieval and use of design knowledge in the knowledge base by specifying characteristics and types of knowledge that may be immediately useful in satisfying a goal of that type. A goal class may name particular methods for use in design, or describe only characteristics of the types of methods that could be useful.

The different types of design goals characterize the design process in part by the way they index the methods in the knowledge base during the design process. They constitute the task description of the design process that specializes the underlying general purpose goal processing architecture into a special purpose design architecture.

**Expected Context.** The expected context of a goal is the set of features of the current context used as input to the goal. Each of the design task goals expect certain clear from the context. Thus, in order to aid readability the distinction between goals and goal classes is dropped when there is no loss in clarity. Only when it is not immediately clear from context, or when an important distinction is being made, will the distinction between an instance and a class be underscored.
parameters to be available within the context that the goal is instantiated. These input parameters determine various characteristics of the goal, such as which types of methods should be activated in pursuing the goal, how these methods are selected, and the criteria for satisfying the goal.

The input parameters may be supplied to the goal when it is instantiated and stored as an augmentation to the goal, or they may be derived from one of the parent goals and extracted from the goal tree at run time.

At the top levels of the design taxonomy, such as the Design Artifact goal class (one of the task goals described in Chapter IV), the expected context of a goal is a complete design specification. At lower levels of the design taxonomy, the expected input may be as simple as a list of attributes and their associated values.

In the brake shoe example of the last section, the expected context of the "design brake shoe" goal would include, for example, an approximation to the total braking force required by the system, and perhaps an indication of reliability and maintenance requirements. The braking force would likely be derived from other features of the current context, such as the total mass of the auto, constraints on the size of the wheels, etc.

If a parameter of a goal is marked as being required input but is not available when the goal is instantiated during problem solving, then the architecture sets up a subgoal to acquire the missing data. Several different types of goals are included in the architecture to handle a number of such situations. Some subgoals result in an action as simple as asking the client for a missing value needed for the computation
of a value in the design. A more common example causes the missing value to be derived from existing values in the design case. Other goals can be added to DSPL++ to match the needs of a target domain. This behavior is the foundation for a seamless interface to an intelligent database manager, such as the PATREC system described by Mittal in [Mittal, 1980] and [Mittal et al., 1984], or the IDABLE system suggested by Sticklen in [Sticklen, 1987].

Resulting Context. The resulting context of a goal is the set of changes made to the current context as a result of processing and satisfying the goal. This is the output of the design task. The application of methods to the design context modifies it in a predictable fashion. Each goal must be able to determine, either implicitly or explicitly, the characteristics of the context that indicate its completion, either with success or failure.

In the brake shoe example, the resulting context is the completed brake shoe design, including its dimensions, materials for fabrication, etc.

The architecture performs limited validity checking of both the expected context for the design task goals and the result generated by the methods that satisfy the goal. Wherever possible, the architecture insures at run time that the design context includes the appropriate features for each different design task goal instantiated during design. When an inconsistency in the input or output context of a goal is detected the system halts and the user is informed.
3.6 Methods for Satisfying Goals

A method defines a strategy for satisfying a goal. For the top level design goals, this translates to a broad problem solving strategy that is capable of producing a design, or partial design, perhaps with some constraints, which satisfies the design requirements. Eventually these sweeping strategies must bottom out in detailed actions, and so some strategies in the system will be more algorithmic in nature. The most general methods represented in the DSPL++ system are those at the design task level. Other methods are domain-specific.

A method has several parts:

1. an index, which describes the method's domain of applicability. This index determines the circumstances under which the method is used during problem solving.

2. an optional sponsor, which can used to determine how well this method matches the problem solving situation,

3. an optional set of preconditions to insure that any input requirements for the method have been met,

4. a method and method-type specific set of actions that interact in various ways with the design database and the current problem state in order to satisfy the current goal, and

---

6Concepts such as a "plan", of course, have much broader application than the task of design. For this work, however, we limit the scope of usage of such concepts to the somewhat narrower roles they play in more routine forms of design.
5. an optional set of post-conditions to determine whether or not the method has produced a locally consistent result.

These parts are depicted pictorially in Figure 5.

The DSPL++ architecture defines a number of different types of methods, or method classes. Each method class fills a different role in the design task analysis. Each different type of method has a characteristic representation and usage of design knowledge. The method types are domain independent abstractions for design knowledge.

The distinction between methods and method types is similar to the previously mentioned distinction between goals and goal classes. Method types in DSPL++ are abstractions, while particular instances of methods contain the details of the domain knowledge. The DSPL++ knowledge base is actually a library of instances of several different method classes.

For example, a particular instance of a decomposition for an air cylinder, AIR-CYL-DECOMPOSITION-27, is a method. This method is a domain-specific knowledge entity, and would likely be useful when an attempt is being made to produce an artifact that satisfies a certain range of specifications that describe a corresponding class of air cylinders. It is an instance of the method class decomposition. The method class, decomposition, corresponds to an abstract, non-domain specific type of design knowledge. That is, the decomposition knowledge type applies more broadly than to the air cylinder domain. However the AIR-CYL-DECOMPOSITION-27 method is a particular instance of the decomposition type, and is domain specific.
Figure 5: The structure of a method
3.7 The Basic Behavior of Task-Level Goals

This section describes the basic mechanisms used to pursue the task-level goals. The actual representations for the mechanisms described below is described in detail in the next section, Section 3.8.

Goals in the DSPL++ architecture fall into two basic categories:

1. task-level goals that represent the design level analysis, and

2. primitive goals that help implement the basic functionality of the architecture.

The members of this latter group are the system primitives upon which the design level functionality is built. These goals have nothing to do with the design task per se, but are part of the underlying mechanism of goal processing. The primitive goals have fixed behaviors, and generally do not produce any subgoals during their operation.

Task-level goals in the DSPL++ architecture are satisfied by methods retrieved from the knowledge base. Several phases of processing occur while pursuing a goal. First, when a goal is selected for processing, any and all methods potentially helpful in satisfying the goal are retrieved from the knowledge base. The second phase begins by determining the suitability of each of the methods retrieved. The suitability of a method for the current context is used to help select which of the methods will be used to satisfy the goal. Finally, the chosen method is applied in the current context. The following sections describe each of these phases in detail. This process is shown diagrammatically in Figure 6.
Figure 6: The basic task-level retrieval cycle
3.7.1 Retrieving Relevant Methods

The retrieve phase of the goal processing activates design methods in the knowledge base. The retrieve phase is implemented as a primitive operation of the architecture that acts in service to the task-level design goals. Its actions are not directly controllable by the user, but rather are controlled by the context in which the goal arises during problem solving. No subgoals can be generated by the Retrieve operation, other than the Resolve Failure subgoal, that is generated automatically by the architecture when problems arise during the normal processing of any goal.

The Retrieve Operation. Retrieving methods from the knowledge base is a primitive operation of the DSPL++ architecture. This operation encapsulates the interface to the knowledge base.

The expected input to this operation is a list of indexes, or characteristics, of methods to guide the retrieval of methods in the knowledge base. Every method in the knowledge base is indexed to facilitate retrieval during this phase of problem solving. These characteristics are passed to an index unpacker that translates the desired characteristics into the actual indexes used to store methods in the knowledge base, and performs the actual retrieval from the knowledge base. The method characteristics passed by the Retrieve operation to the index unpacker are determined by examining the parent context. The Retrieve operation provides a list of methods retrieved by the index unpacker from the knowledge base that have the desired characteristics.
The Retrieve operation uses the index unpacker to determine which methods to retrieved from the knowledge base. This behavior is part of the goal’s definition. Thus this goal does not need to search the knowledge base for methods with which to satisfy itself, and an infinite regress is avoided. The arguments to the index unpacker for retrieving methods are a set of keys extracted from the current design context.

Generating an Index. Several strategies are used to identify potentially useful methods that can be used to pursue a goal. Which strategy is used depends on the type of goal that generated the retrieval subgoal.

The Propose Candidate Design goal, for example, retrieves any methods from the knowledge base capable of proposing designs for the class of artifact being designed. Thus, a Propose Candidate Design goal that was instantiated as a subgoal of a it Design Air Cylinder goal would retrieve all air cylinder cases, decompositions and plans available in the knowledge base, since it is in the context of designing an air cylinder, and since cases, decompositions and plans are the method types that are capable of proposing design candidates.

3.7.2 Method Selection

Methods retrieved from the knowledge base are not simply applied to the design context until the desired state is discovered. Instead, the architecture enters a phase of problem solving during which additional knowledge is used to decide which method will most likely succeed in extending the design in the desired direction.

The reader should note that the selection mechanism presented here is a gener-
alization of the DSPL selection mechanism from plans to methods of any type. In [Punch, 1989], Punch used a similar mechanism to select strategies for diagnosis in TIPS. The selection mechanisms of both TIPS and DSPL++ are directly derived from the DSPL approach.

The retrieval phase in DSPL++ casts a broad net, retrieving methods from the knowledge base based on stored characteristics of the methods, such as the domain of the method or the types of actions the method performs. A small set of potentially relevant methods are produced for further consideration. The selection phase chooses from this smaller set of methods, examining in turn the suitability of each within the current problem context. Some of the retrieved methods may turn out to be unsuitable. Other methods will fit the situation to varying degrees.

A primitive goal type, the Select goal, and two knowledge types, the sponsor and the selector, are used to support this phase of problem solving.

The Select Goal. The Select goal is a primitive goal of the DSPL++ architecture whose purpose is to choose which method to apply in the parent context. The context expected by this goal is a list of methods, pulled from the knowledge base by the successful completion of method retrieval, and the parent problem solving context where the selected method is to be applied. The output of Select goal is a single method selected from the set of input methods.

The selection of a method within the Select goal is based on a two-tiered mechanism:

1. The suitability of each of the relevant methods is determined relative to the
current context.

2. Based on the rating of each of the methods, a method is selected for use, and the goal terminates with success.

This two-level approach allows two distinct perspectives to be considered during the selection process. The first step considers the suitability of each method in only in relation to the current context. The second step introduces comparisons between methods, as well as more globally to the current design context.

If no method is found to be suitable, then no method is selected and the goal suspends, pending the application of any relevant failure knowledge that might be available or attached to the active goal. If no failure knowledge is available, then the goal terminates with failure. If a method is selected, then the method is applied in the context of parent of the Select goal.

Sponsors. A sponsor is a knowledge type used to support the method selection during problem solving. The function of a sponsor is to rate the suitability of a method when that method is being considered for selection. Using knowledge about the current problem context, the method's sponsor categorizes the method into one of five, system-defined, discrete classes, describing how well the candidate method fits the problem context. The five classes and their interpretations are listed in Table 1.

Every method in the knowledge base has a sponsor either implicitly or explicitly associated with it. If no sponsor is explicitly specified for a method then one of several default methods, depending on the method type, are implicitly associated with the
Table 1: Suitability classes for methods

<table>
<thead>
<tr>
<th>Suitability</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERFECT</td>
<td>The method is intended to handle exactly this design context; the likelihood of satisfying the current goal is high.</td>
</tr>
<tr>
<td>SUITABLE</td>
<td>Given the current context, the method will likely help in making progress toward satisfying the current goal.</td>
</tr>
<tr>
<td>INDIFFERENT</td>
<td>It is unclear whether or not the method will contribute toward satisfying the current goal in this context.</td>
</tr>
<tr>
<td>UNSUITABLE</td>
<td>In this context it is clear that the method will probably fail to satisfy the goal.</td>
</tr>
<tr>
<td>RULE-OUT</td>
<td>The method is not suitable for this situation.</td>
</tr>
</tbody>
</table>

method. Two such sponsors are described below.

The knowledge encoded in a sponsor may be relatively simple, depending on a single feature of the current context, or may be arbitrarily complex, consisting of many references to the design state, requesting aid from the user to determine the applicability of the method, or even causing one or more subgoals to be spawned to help determine the single method's suitability.

The scope of the knowledge encoded in a sponsor only extends to the applicability of its own method to the current design context, completely independent of other candidate methods that may be available. Interactions between potentially competing methods is judged during the second phase of selection. This scheme considerably simplifies the organization and representation of knowledge for method selection.

Any feature of the design context is potentially available for inspection within a method’s sponsor. All of the design requirements and the complete design state...
is available to the sponsor. The sponsor may inspect which methods have been attempted, as well as the state of any active goals. The sponsor may take input from the client or from an externally defined user function. Appendix B includes a complete list of the functions available in DSPL++ for accessing the design context, their descriptions and syntax, as well as instructions for creating user-defined functions.

If no domain knowledge is available for individually rating methods in the fashion required by the sponsor, then a default rating is assigned to the method and the burden of discriminating and finally selecting between competing methods falls on the second stage of the selection process. The default rating for a method, as shown in the sponsor illustrated in Figure 7, rules a method out if it has been previously attempted, and rates the method indifferently otherwise.⁷

One useful form of representing the knowledge used to determine the suitability of a method is a tabular representation known as a knowledge group. This representation is found in CSRL [Bylander and Mittal, 1986], a language for building hierarchical classification systems, as well as in the original DSPL [Brown, 1984], and the HYPER system [Johnson et al., 1989]. The TABLE construct in DSPL++ organizes domain knowledge in the same way that a knowledge group does in CSRL. A TABLE is orga-
nized as a number of rows and columns, with each column representing a different feature of the current context that influences the decision to be made. Each of the rows represent a kind of "rule" knowledge, each consisting of a different pattern of features, and ending with an action to be taken if that row's pattern is matched with the values of the features in the current context. The patterns are made up of either specific values which are matched to the corresponding feature, or expressions that are evaluated before matching is attempted. The patterns are examined in order, from the first row to the last row listed in the table construct. The action is taken that corresponds to the first pattern that completely matches the context. If no pattern matches, the default action specified in the OTHERWISE row is taken. If no pattern in a row is matched and no default action is specified, then no action is taken.

A default sponsor for methods that propose candidate designs written in the DSPL++) language is shown in Figure 8. A TABLE construct is used to determine
(SPONSOR
  (NAME DefaultProposer)
  (COMMENT Default knowledge for determining whether
          or not a proposer is suitable for the
          current design state.)
  (BODY
    (DECISIONS
      (REPLY
        (TABLE
          (DEPENDING-ON
            (ALREADY-TRIED? METHOD)
            (ATTRIBUTES-SET? METHOD)
            (INPUT-AVAILABLE? METHOD)
          (MATCH
            (IF (T     ?     ?) THEN RULE-OUT)
            (IF (? (IS ALL)  ?) THEN RULE-OUT)
            (IF (? (IS NA)  ?) THEN UNSUITABLE)
            (IF (?     ? (IS-NOT ALL)) THEN INDIFFERENT))
          (OTHERWISE SUITABLE)))

Figure 8: The default sponsor for proposers
the suitability of the method. Three features of the design context are considered, and four patterns of those features are examined in categorizing the suitability. The interpretations of the rows of the table are as follows:

1. If the method has previously been tried, and the goal was not satisfied, then the method will probably not work again, regardless of how the other features are matched. Thus, the method is ruled out of further consideration. Any further patterns considered can assume that the method has not yet been applied.

2. If values for all of the attributes addressed by this method have already been chosen, then there is no use in applying this method: its work has already been completed. Thus, the method is ruled out from further consideration. Any further patterns considered can assume that there is some useful work that can be achieved by this method.

3. If the attributes addressed by this method are not required in this design context, then this method is unsuitable for this situation. Any further patterns considered can assume that the work done by this method is relevant to this design context.

4. If all of the input required for this method is not available, then whether or not this method will be useful depends on whether or not the additional input parameters can be obtained (either through further problem solving or interaction with the user). Since this information is not immediately available, the method is indifferent. Any further patterns considered can assume that all input
necessary to the method is available.

5. Finally, when none of the previous patterns have matched, the method is assumed to be suitable for application. In this case (from the previous patterns), it can be assumed that

- the method has not yet been tried,
- there are design attributes addressed by this method which have not yet been addressed,
- this method does address the attributes required in this design state, and
- all of the input required of this method is available.

Thus, the method is rated as a suitable candidate to try in this context.

The conservative rating of SUITABLE rather than PERFECT is selected since this particular sponsor represents a default recommendation for a method. A method that is particularly well-suited to certain design contexts will include a sponsor that reflects suitability, and rate the method accordingly, i.e., as a perfect match, when that situation arises.

Note that the suitability ratings listed in Table 1 are not absolute ratings, but rather are interpreted relative to a particular problem context, as indicated by the features listed in the row in which the rating appears.

A each method is rated, it is placed on a list of like-rated methods. The Select goal maintains five lists, one corresponding to each suitability rating, where methods are stored as they are rated. These lists are subsequently passed on to the selection
phase. Every sponsor must categorize its method into one of the five suitability classes listed in Table 1. If any other value for a method's rating is returned by the sponsor, then the method is rated as RULE-OUT, a diagnostic is printed, and problem solving continues.

The results of several TABLE constructs may be chained together if more complicated reasoning is involved for determining a suitability. The results of the intermediate tables may be any group of symbolic values, not just the suitability classes listed above.

Sponsors may be unique to a method, or may be shared among a group of methods that are applicable in similar contexts. A default sponsor is used for methods that have no explicit sponsor attached to it. The default sponsor bases its rating only on whether or not the method has been previously attempted in the current context, and if so, eliminates the method from further consideration.

Selectors. A selector is another knowledge type that supports method selection during problem solving. The function of a selector is to choose a method for application to the current context, from the collection of methods relevant to the current context that are provided to it. A method is selected based three sets of criteria.

1. the results of the sponsors associated with each of the candidate methods, rating those methods more highly that have the highest suitability ratings,

2. any situation-specific domain knowledge that may be available to indicate which methods are relatively more likely than others to succeed in extending the de-
(SELECTOR
(NAME Default)
(COMMENT Default knowledge for choosing a method.)
(BODY
(REPLY
(IF (ANY? (METHODS-RATED-AS PERFECT))
   THEN (PICK ANY (METHODS-RATED-AS PERFECT))
   ELSE (IF (ANY? (METHODS-RATED-AS SUITABLE))
     THEN (PICK ANY (METHODS-RATED-AS SUITABLE))
     ELSE (IF (ANY? (METHODS-RATED-AS INDIFFERENT))
      THEN NO-METHOD-CHOSEN
      ELSE NO-METHODS-APPLICABLE)))))

Figure 9: The default selector

sign, and

3. the current design context.

During this stage of the selection process comparative knowledge is introduced, if available, that measures the merits of each method relative to each other. If no particular domain knowledge is available to aid in the comparison of methods, then a decision is made based solely on the suitability ratings of the methods generated by the sponsors.

Figure 9 shows the selector used if no domain knowledge is available for choosing between methods. The selection is based only on the suitability ratings of the methods, and gives first preference to any method rated as perfect for the current design situation, and secondary preference to any suitable methods. If no method is rated as being either perfect or suitable, then no selection is made and the selector fails.
Note that in design there may be several equally reasonable strategies for approaching a problem, and that it is not often necessary or even desirable to find the single, best strategy before choosing one and proceeding with the design. This kind of optimization may be unnecessarily expensive, especially when any reasonable strategy will suffice. Accordingly, the default DSPL++ selection knowledge allows methods to be chosen from a small set of suitability classes, letting specific domain knowledge override the default to make fine discrimination between individual methods in a class when necessary.

If the selected method eventually fails, the selection process may be called upon to choose an alternative method. A history is maintained in each task of the methods that have been attempted in order to complete the task, thus there is no danger of runaway iteration from repeatedly re-invoking the Select goal. Typically, from a small set of potentially relevant methods, only a vanishingly few methods will be rated as suitable, and likely only one or two. Thus, under the most difficult design situations, the selection process will rarely iterate more than once before it fails to produce a reasonable candidate method to apply. In the worst case, when the method sponsors have no domain knowledge available to reevaluate methods downward in a difficult context, each potentially relevant method is tried, in order from the most likely to succeed to the least likely, until the list is exhausted and the selection process fails.

The two part sponsor/selector mechanism may fail to choose a method for a variety of reasons. In one case, all the potentially relevant methods may be rated as being unsuitable to the context. In another case, a number of methods may be rated
indifferently, but no selection knowledge is available to make the final discrimination. In either case the selection fails and an impasse has been reached. In lieu of simply terminating the current task with failure, it may be advantageous to allow human intervention.

Either under global control of the user, or under program control within the selector, control may be interrupted and influenced interactively during the selection process. Several options are available once the selection process has been interrupted.

- Override the selection procedure and force the selection of one of the available, relevant methods different from that chosen by the system.
- Choose a method from the knowledge base not previously deemed relevant. Any method in the knowledge base may be explicitly activated by the user at any point in the design process.
- Unwind to some previous point in the problem context. Any parent task in the current context may be activated. All design decisions made by a discarded goal in the context are retracted, and the user is then allowed to choose from among this set of control options within the new task context.
- Terminate the task with failure.
- Add new knowledge directed towards overcoming the impasse and resume problem solving. Depending on the nature of the impasse, a number of different types of knowledge can be entered, e.g., a new task definition to handle the existing impasse, a new method better suited to this situation, a new sponsor to replace
the function of a default sponsor, a more discriminating selector, additional indexing, etc.

- Modify (or delete) an existing knowledge structure and resume problem solving.

The first four actions listed are also available under program control during failure handling. The last two actions consist of exactly the same modes of operation open to the user in the creation of a new knowledge based expert system. The ability to extend a system interactively during problem solving adds a powerful dimension to knowledge acquisition and system building.

Selection knowledge may also be encoded using the TABLE construct. Figure 10 shows the selector for methods that are design proposers, that uses two successive tables in determining which method to recommend. This selector is an extension of the default selection knowledge of Figure 9, that selects a method solely on the basis of the method's suitability rating. This selector additionally gives preference to methods that are cases over those that are plans, and plans over decompositions.

3.7.3 Application of the Method

The final phase in processing a goal is the application or execution of the selected method in the current problem solving context. This phase may result either in the goal being terminated with success because the method completes successfully and the goal is satisfied, or the goal being terminated with failure when a failure within the method is recognized.

Subgoals generated by the method cause the currently active goal to be temporar-
(SELECTOR
  (NAME ProposerSelector)
  (COMMENT Chooses the best design proposer for the current design state.)
  (BODY
    REPLY
      (IF (ANY? (METHODS-RATED-AS PERFECT)) THEN
        (TABLE (DEPENDING-ON
          (ANY? (METHODS-RATED-AS PERFECT) Case)
          (ANY? (METHODS-RATED-AS PERFECT) Plan)
          (ANY? (METHODS-RATED-AS PERFECT) Decomposition))
          (MATCH
            (IF (T ? ?) THEN (PICK Case (METHODS-RATED-AS PERFECT)))
            (IF (? T ?) THEN (PICK Plan (METHODS-RATED-AS PERFECT)))
            (IF (? ? T) THEN
              (PICK Decomposition (METHODS-RATED-AS PERFECT)))
            (OTHERWISE (PICK ANY (METHODS-RATED-AS PERFECT))))
        REPLY
          (IF (ANY? (METHODS-RATED-AS SUITABLE)) THEN
            (TABLE (DEPENDING-ON
              (ANY? (METHODS-RATED-AS SUITABLE) Case)
              (ANY? (METHODS-RATED-AS SUITABLE) Plan)
              (ANY? (METHODS-RATED-AS SUITABLE) Decomposition))
              (MATCH
                (IF (T ? ?) THEN (PICK Case (METHODS-RATED-AS SUITABLE)))
                (IF (? T ?) THEN (PICK Plan (METHODS-RATED-AS SUITABLE)))
                (IF (? ? T) THEN
                  (PICK Decomposition (METHODS-RATED-AS SUITABLE)))
                (OTHERWISE (PICK ANY (METHODS-RATED-AS SUITABLE))))
          REPLY
            (IF (ANY? (METHODS-RATED-AS INDIFFERENT))
              THEN (PICK ANY (METHODS-RATED-AS INDIFFERENT))
              ELSE NO-METHODS-APPLICABLE)))

Figure 10: A selector
ily suspended from further processing. When the subgoals are resolved the goal is
restarted and the processing of the method continues.

3.8 Defining Goal and Method Classes

One limitation of the DSPL architecture was that the control behavior of the system
was encoded as part of a single, monolithic interpreter. In order to facilitate extensions
and modifications to the DSPL++ architecture, the behaviors that define the design
process have been encoded as part of the default knowledge contained in the system.
Both methods and goals have been designed to facilitate extensibility.

3.8.1 Specifying Goal Behaviors

This section describes how the primitive actions described in the previous section are
formulated into the behavior specification of a task-level goal's behavior in DSPL++.
Most of the specification of a task-level goal's behavior are generic actions, such as the
retrieval of methods from the knowledge base, however, different goals combine the
basic actions differently depending on the needs of the particular function the goal
serves within the design process. The complete description of the task-level goals in
DSPL++ is reserved for Chapter IV.

In this section, the low-level behavior of the Propose Candidate Design goal class
is used as an example of how the basic behavior of a task-level goal is specified. The
Propose Candidate Design goal is a task-level goal whose function is to select a certain
class of methods, called proposers, and apply them to the design context. The role of
the Propose Candidate Design goal is described in further detail in the Chapter IV.
Goal Behavior Definitions. A goal's behavior definition describes the runtime behavior of the goal. Every goal in the DSPL++ architecture, whether system or user-defined, has a corresponding behavior attached to it that defines how the goal is handled by the DSPL++ interpreter. There is only a single possible behavior associated with each goal type, and it is automatically run whenever that goal is selected to be pursued by the goal interpreter.

A goal behavior is a list of actions that are used to direct the goal interpreter during a goal's processing. When a goal is instantiated during problem solving, the goal behavior is taken from the goal's defining class and placed on the goal's command stream. The actions on the command stream are processed sequentially by the interpreter, until either the list is exhausted or the interpreter is directed to begin processing a new goal. The goal behaviors are written in GGBL, a task-specific, high-level language for specifying goal behaviors in DSPL++. The details of the language syntax and primitives are provided in Appendix B. Chapter V shows how this language is used to incorporate new tasks into DSPL++.

Figures 11 through 13 show the goal behaviors for the Propose Candidate Design goal. This definition encodes the Retrieve/Select/Apply cycle described in the previous sections, but particularized for the Propose Candidate Design goal.

The behavior definition shown in Figure 11 is executed by the goal interpreter whenever a Propose Candidate Design goal appears on the goal stack. The BODY of the definition is a list of imperatives to the goal interpreter, describing the actions to take in pursuing this goal. The definition of the Propose Candidate Design goal has
three parts, as described previously in this section. They are:

Retrieve relevant methods from the knowledge base,

Select the most appropriate method, and

Apply the selected method to satisfy the goal.

These actions are sequentially read and carried out by the goal interpreter as it pursues the Propose Candidate Design goal.

The RETRIEVE imperative indicates that methods are to be retrieved, and lists the characteristics of relevant methods. The device type, such as an "air cylinder" or "distillation column" is not known until runtime, and so must be taken from the design context at runtime. The SELECT imperative causes the Select goal to be spawned, where method selection takes place. The Default Proposer selector, described in the
last section, is used to implement the actual selection. The APPLY imperative causes the selected method to be applied to the current design context.

**Failure Methods.** The goal behavior described above only accounts for the standard behavior of the system when processing a *Propose Candidate Design* goal. This behavior must be augmented when failures are encountered during goal processing.

A *failure method* is a method type that encodes a design strategy in DSPL++ for coping with impasses that arise during problem solving. The task-level design strategies in DSPL++ are encoded using this method type. The **BODY** of a design method consists of a list of the actions taken when the method is applied, described in a simple language similar to that found in the behavior definition.

Situation-specific failure handlers are design methods that may be established or removed within a goal's behavior definition in order to control the goal's behavior when a failure occurs. Figures 12 and 13 show two design methods that are used to handle certain impasses that can arise during the processing of the *Propose Candidate Design* goal. These methods modify the basic behavior described by the behavior definition of Figure 11.

The first method, the **AcquireProposerAndRetry** method, is set up to handle impasses that occur in the retrieval phase of the goal processing. This handler initiates knowledge acquisition of a design proposer in the event that no method for proposing design candidates are found in the knowledge base that match the characteristics described in the retrieval phase. When the *Propose Candidate Design* goal is threatened with failure because no relevant methods are available, this method is invoked.
(FAILURE-METHOD
  (NAME AcquireProposerAndRetry)
  (SPONSOR AcquireProposer)
  (BODY
    (APPLY-IN-FAILING-CONTEXT
      (FAILURE-METHODS OFF)
      (SUB-GOAL AcquireProposer)
      (RESTART Retrieve))))

Figure 12: A method for initiating knowledge acquisition

All actions listed in this methods are performed in the Propose Candidate Design goal context, that is the context threatened with failure and that caused the AcquireProposerAndRetry method to be invoked. The context for the actions is chosen by the APPLY-IN-FAILING-CONTEXT imperative. The FAILURE-METHODS imperative prevents any further failures from being recognized. The SUB-GOAL imperative generates a new subgoal to interact with the user to acquire a new proposer suitable for this situation. If a new proposer is successfully acquired, then the RESTART imperative causes method retrieval to be restarted and the method completes with success. The handler is removed after the retrieval phase completes.

Another method, the SelectAgain method, is used to select an alternative design proposer in the event that the initial selection fails to produce a viable candidate. SelectAgain method causes the Select goal to be restarted in the Propose Candidate Design goal and a new proposer to be selected and applied. If at any point the Select goal fails to select a new method, the entire Propose Candidate Design goal terminates with failure.
(FAILURE-METHOD
  (NAME SelectAgain)
  (SPONSOR SelectAgain)
  (BODY
    (APPLY-IN-FAILING-CONTEXT
      (FAILURE-METHODS OFF)
      (SELECT)
      (FAILURE-METHODS SelectAgain)
      (APPLY))))

Figure 13: A method that restarts the selection process

Referring to Figure 13, the APPLY-IN-FAILING-CONTEXT imperative is used again to determine the context in which the rest of the method is executed. The FAILURE-METHODS imperative prevents further failures from being handled. The SELECT imperative causes the Select goal to select a new proposer from the list of methods first retrieved in the Propose Candidate Design goal context. Since no failure methods are active, a failure at this juncture causes the entire Propose Candidate Design goal to fail. If a new method is selected, then the SelectAgain method is again activated by the FAILURE-METHOD imperative to handle further failures, and the newly selected method is applied to the current context by the APPLY imperative. If the new method completes with success, the entire Propose Candidate Design goal terminates with success. If the new method fails, then the SelectAgain method once again attempts to select another method. This process continues until either the APPLY imperative succeeds or complete failure occurs in the Select goal.
3.8.2 Method Behaviors

Because of the complexity and variety of the design task and its domains, no taxonomy of knowledge types could ever hope to be complete. The list of method types found in the DSPL++ architecture is no exception.

Some subtasks useful for design cannot be considered part of the design task, for example, qualitative reasoning in support of simulation for the purpose of verifying the acceptability of a design. No closed set of primitives can cover all possibilities without compromising either efficiency or directness of representation.

Method Behavior Definitions. A method behavior defines the runtime behavior of a method in DSPL++. Every method type in the DSPL++ architecture, whether system or user-defined, has a corresponding method behavior attached to it that defines how the method is run by the DSPL++ interpreter.

A method type definition performs several functions in the DSPL++ architecture:

- defines a class of methods and the range of actions allowed by the class,
- defines the form of the allowable method-specific actions, and
- defines where and under what general circumstances methods of this type will be retrieved during problem solving.

A method behavior is a list of system primitives that are placed on the goal’s command stream during method application. A method is executed by the interpreter when its behavior list is placed on the command stream of the goal to which it is
(METHOD-BEHAVIOR
  (NAME Plan)
  (BODY
    (FAILURE-METHOD FailWithSuggestions)
    (EVALUATE Preconditions)
    (FAILURE-METHOD PlanRedesign FailWithSuggestions)
    (PUSH-ONTO-AGENDA PlanBody)
    (EVALUATE PostConditions)))

Figure 14: A method behavior definition

being applied. The commands come from the method's behavior definition listed in the method's method class. The behavior is written in GMBL, a simple language for specifying control behaviors in DSPl++. The details of the syntax and primitives of this language are provided in Appendix B. Examples of how behavior definitions are used to extend the architecture by creating new method types is discussed in Chapter V.

Method behaviors define how methods are processed during design. No indexing is required, since they are always referenced directly in the method class whose behavior they define. No sponsor is required since there is only a single possible behavior associated with each knowledge type, and it is automatically retrieved and run whenever a method of that type is applied.

Figure 14 shows the behavior definition for plans. The failure handler named FailWithSuggestions is established to trap any impasses arising from the evaluation of the preconditions. A second failure handler, PlanRedesign is established before the body of the plan is pushed onto the interpreter's internal agenda for execution.
If all of the plan items successfully complete, the postconditions are evaluated and plan execution completes. Note that the PlanRedesign handler is left active during the evaluation of the postconditions, allowing for the possibility of a postcondition initiating plan redesign.

3.9 Failure Handling

This section provides an overview of how failures are handled in various design contexts. The circumstances under which failure handling is initiated are listed, the application of failures methods is described, and a description of how complete design failure is also provided.

A failure in DSPL++ is an interruption in the normal flow of processing of either a method or a goal's standard behavior. A failure handler is a method that addresses the conditions of a specific class of interruptions. Intuitively, a failing context is any problem solving context in which no further action can be made by the architecture without the application of exceptional strategies.

Failures that arise during the design process are handled by applying the same ideas used for controlling the overall design process itself. As soon as a method or a goal is threatened with failure, it generates a new subtask to handle the problem. The new goal is handled exactly as any other goal in the architecture. Relevant methods are retrieved, one is selected and applied to the current context, and if successful, the threatened goal resumes processing.

A goal may be threatened with failure in any of several ways.
• No relevant methods are available to move the goal towards completion,

• all relevant methods are found to be unsuitable for the purpose of moving the goal closer to completion,

• relevant methods suitable to the current context are available, but no decision is made on which method to use,

• a method applied to the goal state failed to complete,

• a method applied to the goal completes, but fails to satisfy the goal’s completion criteria, or

• the behavior definition of a goal explicitly terminates the goal with failure.

A method may also fail for a number of reasons.

• A subgoal generated by the method terminates with failure, or

• the method definition explicitly terminates the method with failure.

Any of the above conditions may initiate failure processing.

Failures may arise in the architecture in one of two situations: those in which pre-compiled knowledge is available to handle the failure and those without pre-compiled knowledge. When no pre-compiled knowledge is available, the failure processing mechanism to backup through the goals until some knowledge about how to handle the failure. There are no default methods for handling failures in DSPL++. 
The *Resolve Failure* Goal. The function of the *Resolve Failure* goal is to select an available failure strategy and apply it in the failing context. This goal requires that the parent goal provide a list of failure strategies. If none are available, then the goal terminates with failure, and control is returned to the parent goal. The expected input and resulting context of the *Resolve Failure* goal are inherited from the parent context.

This goal is typically instantiated whenever an application failure occurs. There are only two conditions of failure that are not handled by the automatic instantiation of the *Resolve Failure* goal.

1. the processing of the *Resolve Failure* goal itself is interrupted, because, for example, no methods are relevant or no relevant method could be selected, and

2. any goal fails because of the immediate, unsuccessful application of the *Resolve Failure* goal.

These two cases are "hard-coded" into the goal interpreter, and constitute the only instances where the goal interpreter’s behavior is dependent on the type of goal being processed. In these cases the goal interpreter simply passes the impending failure to the immediate parent. These exceptions prevent an infinite regress from failures within failures.\(^8\)

The *Resolve Failure* goal only retrieves failure strategies by name, not by type. That is, no search is done for a potential strategy, but rather only specific strategies, known to be suitable in the failing context, are attempted. Failure strategies are

\(^8\)Note that failure processing may proceed normally within a subtask generated by a failure method.
selected based on the characteristics of the strategy, just as is done for other task-level goals in the architecture. This mechanism, while not totally flexible, allows each method type to define its own failure strategies and behaviors.

The behavior of the *Resolve Failure* goal, shown in the behavior definition of Figure 15, uses a variation on the Retrieve/Select/Apply cycle to find and apply an appropriate method to attempt to recover from the impasse. The RETRIEVE imperative only retrieves the methods listed by name in the current context. The SELECT imperative shows that the Default selector is used to pick a suitable strategy. If one failure strategy fails, another is attempted until either all possible strategies are exhausted, or are found to be unsuitable.

### 3.10 The Design Database

The last major component of the DSPL++ architecture is the design database. The design database is the repository for the device representations that are elaborated by the design process. Note that all device descriptions exist in the design database.
before the beginning of design. No novel devices are created during design, although novel combinations and configurations of devices may be explored in pursuing the requirements of a particular design problem.

3.10.1 The Structure of the Database

The design database is a collection of device definitions created by the user. The device definitions are implemented as LOOPS objects, created under the control of the DSPL++ interface, and have the essential characteristics and behavior of that system. The DSPL++ system constraints the creation of the LOOPS objects by insuring certain features required of the device description are provided by the user.

3.10.2 Accessing the Database

Several primitives have been provided in DSPL++ for manipulating the elements of the design database during problem solving.

**KB.FETCH** retrieves the current value of a single attribute of a device currently instantiated in the database. The function fails if the specified device does not exist in the database, or the specified attribute does not have a value.

**KB.STORE** sets a new value for an attribute of a device. The function fails if the specified device or attribute do not exist in the database.

**INITIALIZE** creates a new instance of a device in the database, and returns the pointer to the instance.

**DISCARD** removes an instance of a device from the database.
REPORT-ON allows the values and attributes of a device to be inspected interactively by the user.

All of these functions may appear in any DSPL++ expression.

3.10.3 Device Representations

A device is characterized by the set of attributes that describe the device. Each attribute may take on a range of values, either through the execution of a primitive method or provided externally by the user. These attributes perform several roles in the DSPL++ system. First, they are used to define the form of the primitive objects that are being designed. These objects form the vocabulary of primitive objects over which the design process occurs. Second, they are used to represent the requirements that must be specified before problem solving can begin. Third, they describe structural relationships among components through component hierarchies, similar to decompositions.

The design objects used during the design process are instances of device classes in the DSPL++ design database. These device classes are LOOPS programming objects that are defined by the user during a knowledge acquisition phase and is guided by the DSPL++ interface. An example of the acquisition of a device class is shown in the next chapter, Section 5.1.3.

Device definitions are implemented in DSPL++ as LOOPS classes. All of the device classes in the design database are subclasses of the DSPL++ system LOOPS class $DesignObject$. This class is the "generic" design object class in DSPL++ and contains all the generic system behaviors of the device representations defined by the
DSPL++ architecture. The behaviors in the $DesignObject$ class are inherited by each of its subclasses.

Currently, the result of design in the DSPL++ architecture is a fully parameterized instance of a pre-defined device class found in the design database. Devices are instantiated during design, attributes of the device are given values during the design process, and are finally included as part of a design case that records the results of problem solving. Only objects whose structure is previously known and recorded in the design database can be instantiated and designed.

In theory, it is possible that new device descriptions could be added to the design database as a result of design problem solving. There is no practical reason why the actions of a method selected from the knowledge base during design could not include the creation of a new device class. However, we have not currently attempted this class of design, and no methods have been developed that affect the design database in this way. Additionally, only device instances can be associated with a design case that records the results of design problem solving.

**Device Requirements.** Attached to a device definition may be a set of requirements that describe aspects of the device that must be made available before design of the device can commence. These requirements may be used during problem solving to elaborate incomplete problem specifications.

The structure of the device requirements includes:

- **Constraints** on the device describing limits to aspects of its functionality or behavior,
Attributes that parameterize the device, and

Functions that describe the intended use of the device.

3.11 Discussion

Controlling subgoal generation. Any architecture that allows subgoals to be generated to resolve impasses that arise during problem solving must eventually "bottom out" the subgoaling behavior in actions that do not themselves produce any subgoals. If no such terminal actions are taken, the subgoaling behavior becomes an infinite regress. Since the kernel of the DSPL++ architecture has a rather general subgoaling behavior, it is reasonable to ask whether or not it is possible for a system built using DSPL++ to engage in runaway subgoaling?

Subgoal generation is implicitly controlled in DSPL++ by the limited number of places subgoaling can occur and the limited roles subgoals are allowed to address. The task analysis shows where subgoaling can occur, and unless a method, such as a decomposition, is self-referential, subgoaling will always eventually bottom out. The language acquisition module could check for obviously self-referential methods syntactically, and refuse or at least warn of such situations.

A group of methods could be encoded that are mutually recursive, and in general detecting such recursion would be prohibitively expensive. However, the structure of the domain will likely work to prevent such situations. Methods will typically begin with descriptions at one level of abstraction in a domain, (eg, an automobile) and move to progressively more and more detailed levels (eg, brakes, engine, transmission,
etc.) within the domain. It seems unlikely that the design of an automobile would result a subgoal that requires the design of an automobile.

Handling failures during problem solving is another critical area where subgoaling must be carefully controlled. For example, how many iterations of Critique Design/Modify Design will the system engage in before it gives up? If no domain knowledge is available to terminate the succession, then perhaps the architecture could terminate the search after a finite number of iterations, with the potential hazard of terminating a useful (but long) chain of reasoning.
CHAPTER IV

Tasks and Methods for Design

Up to this point, the description of DSPL++ has involved only its task-independent features. This chapter describes the features of the architecture, goals and methods that play specific, functional roles in the design process and that identify it as a shell specifically addressing the design task. The tasks described here organize the selection and application of design knowledge and form the framework for control of the design process.

4.1 The Design Artifact Goal

The Design Artifact goal is a request to the system to initiate the design of a particular type of artifact. This goal defines the top level of the design task, and provides the means by which design requests are couched to the system. It may be generated either internally during design or external to the system. The goal is generated internally whenever a subproblem is recognized, e.g., by a decomposition or as an item in a design plan. The top level goal in a design run is a Design Artifact goal handed to the system by some external source, typically at the user’s request.

The expected input to the Design Artifact goal is a specification of the object to be designed. If the Design Artifact goal is instantiated to handle a design subproblem,
then the type of artifact and its specifications can be extracted from the current context. If the *Design Artifact* goal is instantiated for the beginning of a new design problem, then the artifact type and specifications must be provided by the user.

The structure of an object to be designed must be recorded in the design database before the design proper can begin. This allows the input specifications to be collected and verified before they are used in design proposal. All of the requirements for the design of a device are obtained either from the user or the description of the device itself.

Initially, only the type of the object to be designed is required for the instantiation of a *Design Artifact* goal. The objects defined in the design database are indexed by object type. Thus, the availability of the object type allows the rest of the structure of the desired object to be accessed. In particular, the form of the specifications required for the design of an object are associated with the object in the design database. These data are used to either verify the availability of the necessary requirements in the current design context, or to organize the collection of missing or additional items interactively from the user.

If the parameter of the *Design Artifact* goal is not known to the design database, then a new object must be described. The only method currently available is to enter a knowledge acquisition phase (see Chapter V for details), but methods could be devised that produce new primitives. Regardless of how the new primitive object is described, once the description is complete, the input requirements can be collected and problem solving can proceed.
To summarize, the *Design Artifact* goal performs a number of functions that organize the upper levels of the design task. They are:

- instantiates objects from the design database,
- insures that objects definitions exist,
- initiates the acquisition of the definition of unknown objects,
- collects requirements if necessary,
- retrieves and selects among the possible design strategies for producing and verifying an object of the desired type, and
- pursues the selected design strategy until an object is produced that meets the design specifications.

The goal behavior definition that produces the behaviors of the *Design Artifact* goal is shown in Figure 16. All object initialization and specification collection is handled by the *Initialize Artifact* goal. The methods for satisfying the *Design Artifact* goal are the subject of the next section.

The result of satisfying this goal is a complete specification of the desired object that satisfies the input requirements.

### 4.2 The *PVCM* Method for the *Design Artifact* Goal

The *PVCM* method is a design strategy based on the task analysis found in Chapter II. This method combines the preceding architectural features into a strategy for
Proposing and Verifying a Design. One important class of methods for design involve proposing a design, and subsequently attempting to verify that the proposed design indeed satisfies the design requirements. This is the motivation behind the PVCM method for design. The method succeeds if a candidate design can be generated, and can be verified to meet the design specifications. If the PVCM method succeeds, then the goal that the method was applied to, namely an instance of the Design Artifact goal, also succeeds.

In the simplest case, then, the PVCM method causes two subgoals to be generated and placed on the goal tree for consideration, one that proposes a candidate design and one that checks to see if that design meets the input specifications.
(DESIGN-METHOD
(NAME PVCM)
(COMMENT Finds one candidate design that satisfies
the specifications.)
(SPONSOR Suitable)
(DESIGN-TASK DesignArtifact)
(ARTIFACT Any)
(BODY
(Failure-METHODS OFF)
(SUB-GOAL ProposeCandidateDesign)
(Failure-METHODS ModifyCandidate FindNewCandidate)
(SUB-GOAL VerifyDesign)))

Figure 17: The PVCM method

Figure 17 shows PVCM method as it is encoded in DSPL++. The method generates the two subgoals, Propose Candidate Design and Verify Design, as described above. Initially the FAILURE-METHODS imperative removes all active failure methods. If the subsequent SUB-GOAL imperative for the Propose Candidate Design goal fails to complete, then the entire method fails and control is returned to the parent Design Artifact goal. When the Propose Candidate Design goal completes with success, the FAILURE-METHODS imperative establishes two potential strategies for handling any problems within the Verify Design goal.

Ignoring the failure strategies for the moment, the method of Figure 17 reflects the most direct path through the design process:

1. a design is proposed,

2. the proposed design fits the specifications of the problem, and
3. the design problem is solved.

Of course, design is not always such an easy proposition. Complications invariably arise, and alternate behaviors must be available to compensate for them. These alternatives will be discussed in due course. However, one simple complication to garden path design can be discussed and dispensed with without reference to any additional design behaviors. The problem is, what happens if the Propose Candidate Design goal immediately fails to produce even a single candidate design?

If no candidate design is available, then we have very little with which to work, and the PVCM method simply fails. The Propose Candidate Design goal is the only source of design solutions, and if it fails to produce a candidate solution, there is simply no recourse. Unless an alternative to the PVCM method is available to the Design Artifact goal, then the Design Artifact goal fails also. This is reflected in the illustration in Figure 17, which shows that all failure handlers are disabled before the Propose Candidate Design goal is generated. Thus, if an impasse arises during the processing of the Propose Candidate Design goal, no failure strategies are available and the method fails. Control is passed on to the Design Artifact goal.¹

We now turn to the description of the goals that implement the two subtasks of proposing and verifying candidate designs.

¹Of course, there may be considerable activity to handle failures that arise due to difficulties within the Propose Candidate Design goal itself, but this is of no concern to the design process at the Design Artifact level of analysis. Failures within the proposal process are completely localized within the Propose Candidate Design task. Unless and until the Propose Candidate Design task fails completely to generate a candidate design, no failure reported to the Design Artifact goal.
The Propose Candidate Design Goal. The purpose of the Propose Candidate Design Goal is to generate a candidate design. The input to the goal is the specification of the object to be designed in the current context, i.e., the device specification listed in the parent Design Artifact goal. When the goal is instantiated, the device specification is retrieved directly from the context stack.

The behavior of the Propose Candidate Design goal follows the cycle of retrieving, selecting, and applying methods described in Chapter III. The index to the knowledge base is generated partly through the device specification and partly through knowledge embedded in the goal definition that describes the types of methods that are capable of proposing designs.

The methods that help satisfy the Propose Candidate Design goal are those that propose design alternatives of one sort or another. We call such methods design proposers. The Propose Candidate Design goal retrieves for consideration any instances of methods that are capable of proposing a candidate design and that match the object's requirements. Several types of methods are defined in the DSPL++ system that are used to propose candidate designs.

Plans: A highly compiled method which, when applied to the current context, results in the desired object.

Task Decompositions: A method that specifies a set of smaller design problems that need to be solved.

Cases: This method uses an instance of the desired object that was designed previously to an earlier set of specifications and stored in the knowledge base. Any
case in the knowledge base whose specifications are relatively near the current specifications is a potential candidate for the current design.

The list of methods for proposing design in DSPL++ does not represent an exhaustive list of all possible techniques for proposing design. The method types listed meant to address some of the most common forms of proposing designs over a wide range of domains. It is expected that these method types will serve a range of applications, but that other applications will admit variation. Other proposer types may be used, as required by the domain and defined by the user.

These methods are described in Section 4.5. At this point is it sufficient to recognize that each of these methods are capable of producing a design to a given set of specifications.

All of the methods for proposing designs are stored in the knowledge base and are indexed by the specifications of the object that they generate. Only those methods relevant to the current design problem are considered for use. For example, a goal to propose a piston design activates any piston plans, but not any other plans.

Note that successfully generating a candidate design satisfies the Propose Candidate Design goal, but that does not imply that the parent Design Artifact Goal is necessarily satisfied, only that a candidate design is available. Successful completion of the parent goal occurs when the parent’s method completes successfully. This involves an independent verification that the design satisfies the design specifications, as indicated by the PVCM method.

The retrieval phase of the Propose Candidate Design fails if no design proposers
are found in the knowledge base for the requested device. The goal fails immediately, and control is returned to the parent method.

If design proposers are available from the knowledge base, but the selection phase determines that none are appropriate, then the goal also fails.

If a design proposer is selected, but fails to complete, then a new proposer is selected and applied, until either a candidate design is successfully generated, or all proposers have been exhausted. These behaviors are specified as part of the *Propose Candidate Design* goal's behavior definition.

**The Verify Design Goal.** The function of the *Verify Design* goal is to take a completed design and check that it satisfies a given set of design specifications. The input to the *Verify Design* goal is the candidate design produced by the *Propose Candidate Design* goal, and the design specifications provided to the parent *Design Artifact* goal. The goal terminates with success when the candidate design has been shown to meet the design specifications by every method found to be applicable to the design state. If the candidate design can be shown to violate the specifications, the *Verify Design* goal terminates with failure. If no methods are available that can either verify or reject the candidate design, then the *Verify Design* goal suspends processing, and control is passed to the parent *Design Artifact* goal.

A design can be inspected from many different points of view. The methods that verify a design will typically only check the candidate design from a single perspective. Thus, verification is accomplished in a number of passes over the design, rather than by a single strategy from a single method. This is reflected in the *Verify Design* goal’s
behavior.

The *Verify Design* goal, like the preceding task level goals, uses a variation of the Retrieve/Select/Apply cycle. However, because the verification of a candidate design can never be absolutely confirmed by a method (only more evidence accumulated that it meets the specifications), the cycle is modified slightly. Instead of terminating after the successful application of the first method selected, the behavior of the *Verify Design* goal applies *every* suitable method for verifying the design until either one method succeeds in rejecting the candidate design, or all methods find the design acceptable. If all verification methods find the design acceptable, then the *Verify Design* goal completes with success.

Figure 18 shows the behavior definition for the *Verify Design* goal. The *RETRIEVE* imperative retrieves any verifier methods that pertain to the current design’s domain. Each method retrieved that is found to be suitable to the context is then applied, until the *SELECT* imperative fails to produce a method. When the *SELECT* fails, the current failure handler causes the goal to terminate with success.

The methods that satisfy the *Verify Design* goal may potentially take many different forms. One default method and two method types are provided in the *DSPL++* architecture:

**Verification Plans:** A verification plan is a compiled method for verification, corresponding roughly in form to the design plan for proposing candidate designs. The verification plan is the only method type in the *DSPL++* architecture for verifying candidate designs. The elements of a verification plan do not make
changes to the design state as do the elements of a design plan. Instead, a verification plan consists of a collection of constraint checks on the candidate design that insure that it meets its design specifications.

Constraints: A constraint is a test on the value of one or more attributes in the candidate design. If the test succeeds, the method succeeds.

The “Ask User” Method: In the event that there are no methods present in the knowledge base relevant to verifying the correctness of a design, a single, default verification method has been provided in the DSPL++ architecture that interactively interrogates the user about the acceptability of the candidate design.

The verification plan provides a simple mechanism for representing a highly compiled
form of verification knowledge in any domain. Depending on the domain require-
ments, the user may add more complicated methods to perform some form of spatial
reasoning or qualitative simulation, or even more traditional methods of an analytic
nature, such as a finite element analysis.

The AskUser verification method, shown in Figure 19, is a default verification
method that simply suspends problem solving to allow the user to inspect the can-
didate design to determine whether or not it meets the design specifications, and
subsequently “accept” or “reject” the design. If the user accepts the candidate de-
sign, then the method completes with success. If the user rejects it, then the method
fails. Both the inspection of the candidate design and the selection of the design’s
suitability is made interactively though the user’s intervention. The AskUser method
is always given an "indifferent" rating by its sponsor, and is never considered for use unless no other method is found to be suitable for the verification task. Only in this case does the Verify Design selector choose this method.

The Verify Design selector is illustrated in Figure 20. This selector first chooses among the methods that are rated as perfect for the verification task at hand, and then among the suitably rated method. If no perfect or suitable methods are available, but at least one method has been tried, then the selector terminates with no selection. Finally, if there are no perfect or suitable methods, and no method has yet verified the candidate design, then the Ask User method is selected.

Modifying a Faulty Design. If a candidate design is created, and subsequently verified, then the Design Artifact is satisfied and design is complete. But what if the proposed design does not meet the specification? If considerable effort has gone into
generating the candidate design, then rather than discarding it and looking for a new candidate solution, perhaps it can be modified in some fashion to make it work. This is the motivation behind the \textit{Modify Candidate} method that acts to reduce the cost of searching for a suitable design by modifying an existing design rather than generating a new one.

This method is activated to handle failures generated by the \textit{Verify Design} goal. When verification of a candidate design fails, this method continues the design process. Two new subgoals are generated and placed on the goal tree for consideration, one, the \textit{Critique Design} goal, to attempt to figure out how the proposed design fails to meet the input specifications, and another, the \textit{Modify Design} goal, that attempts to take this information and use it to change the faulty candidate design. When a suggestion to modify the faulty candidate design is successfully implemented, a new \textit{Verify Design} subgoal is generated to check the results of the modified candidate design.

If the \textit{Critique Design} goal fails to produce any suggestions for modifications to the faulty candidate design, this strategy immediately fails, and control is returned to the parent \textit{Design Artifact} goal. Other problems may arise in attempting to modify the candidate design which will be addressed shortly.

The method that encodes the strategy to modify a faulty candidate design is shown in Figure 21. The SUB-GOAL imperative is used to place each of the goals to critique, modify and re-verify the candidate design on the goal tree. A new failure handler is established to handle problems that may arise in either the implementation
of a suggestion or the verification of the resulting design. The failure handler and the goals that implement the two new subtasks are described below.

The Critique Design Goal. The function of the Critique Design goal is to determine which aspects of a candidate design are responsible for the candidate's failure to meet the design specifications, and produce a suggested change to the design to correct the design. The Critique Design goal is instantiated only after a completed design, or portion of a completed design, has been found by the Verify Design goal to be deficient in some fashion.

The expected context of this goal is a (faulty) candidate solution to the current design problem and the details of how that design has failed to meet the design specifications. The result of this task is a suggested change to the candidate design intended to correct the candidate's deficiencies.
The behavior of the Critique Design goal, listed in Figure 22, is again a variation of the Retrieve/Select/Apply cycle, this time very similar in form to the Propose Candidate Design goal's behavior. The Critique Design goal applies methods relevant to the current candidate and its specifications until one successfully generates a suggestion for modifying the faulty design candidate. If no relevant criticizers are found in the knowledge base through the RETRIEVE imperative, a failure handler established by the previous FAILURE-METHODS imperative provides for the option of acquiring the appropriate domain knowledge. Otherwise, the SELECT imperative is used to select a method, and the APPLY imperative applies it to the context. The SelectAgain failure handler catches failures generated from applying the selected method, and causes a new method to be selected and applied to the context until one finally succeeds, or the selection process fails and the entire method fails.

In general, criticizing a faulty design is a very broad problem, and has been ad-
dressed in detail elsewhere, for example, in [Goel, 1989] and [Goel and Chandrasekaran, 1989]. The task properly includes an analysis of the reasoning behind the rejection of the design, examining the relationship of both the candidate design and the specifications to the verification knowledge that was used to reject the design, as well as many other practical considerations, such as computational expense required to make those relations apparent and the models and analytical tools available. General methods to attack the general form of this problem are not yet available. Strategies that depend upon more compiled forms of knowledge tend to be domain dependent, but have the advantage of being much more approachable in the context of building practical expert systems.

Several routine forms of criticism are included in the DSPL++ architecture. The simplest methods use pre-compiled suggestions either attached to the verification knowledge or indexed according to the design state. A slightly less compiled strategy derives a suggestion for redesign based on the structure of the verification method that rejected the candidate design. If the verification method is a constraint that failed in the verification of the design, then a suggestion can be generated based on the structure of the constraint and the way the constraint was violated.

The methods for criticizing candidate designs and producing suggestions for modifying them are available in the DSPL++ architecture are

**Constraint Analysis Method:** This method is only suitable when the cause of failure of the *Verify Design* goal is a simple constraint. This method examines the constraint that caused the failure of the *Verify Design* goal, and attempts
to extract a suggestion from the structure of the constraint.

**Direct Suggestion Method:** This method examines the method that caused the failure of the *Verify Design* goal for any pre-compiled suggestions that may be listed there to direct the modification process.

**Suggestion Maps:** A suggestion map uses the current context to index pre-compiled suggestions for directing the modification process.

In the DSPL system, compiled suggestions are attached to various agents to indicate how to direct redesign when the design proper failed to produce a suitable design solution. This behavior is hard-coded into the DSPL interpreter. The DSPL++ interpreter also supports redesign by suggestion, but implements this behavior using a framework in which extensions to the suggestion language as well as novel forms for both suggestions and modifiers can be integrated by the user into the basic architecture.

Another potentially useful form of knowledge that has not been provided in DSPL++ is the classification hierarchy of CSRL [Bylander and Mittal, 1986]. Quantitative simulators, written in a language such as MATHEMATICA, or qualitative simulations using, for example, the consolidation techniques suggested by Bylander in [Bylander, 1986] would also be appropriate mechanisms for developing methods for both for the verification and critique tasks. Goal in [Goel, 1989] describes a form of case based design. While practicality considerations preclude every possible mechanism from being included in a single language system, it should nonetheless be possible to easily integrate different modes of reasoning into the design process. The
methods listed here are intended as a starting point for representing some basic types of methods.

Even without extending the DSPL++ language to incorporate additional modes of problem solving, the task analysis promotes an architecture that facilitates interaction with other problem solving systems. This property arises from the clear functional description of the design task, and its direct implementation in the DSPL++ architecture.

The Modify Design Goal. The Modify Design goal is responsible for modifying designs based on the results of design critiques. The expected context of the Modify Design goal consists of the faulty candidate design and a suggestion indicating a potential direction for modifications on the candidate design to make it conform to the design specifications. The Modify Design goal attempts to implement a single suggestion by applying one or more modifications to the faulty candidate design.

The behavior of the Modify Design goal, listed in Figure 23, again uses the Retrieve/Select/Apply cycle similar to the Critique Design goal. Again, a list of relevant methods are retrieved, and applied until a single method succeeds in implementing the suggestion provided to the goal. A method is relevant if it is a modifier capable of implementing one of the supplied suggestions. If no relevant method are found in the knowledge base through the RETRIEVE imperative, a failure handler provides for the option of acquiring modification knowledge from the user. The SELECT imperative selects a method, and the APPLY imperative applies it to the context. The SelectAgain method handles failures in method application by selecting a new
method for implementing the design suggestion. The goal succeeds when a method implements the suggestion, or fails when selection process fails to find any suitable method for attempting the modification.

The types of methods that are relevant to this task are those capable of implementing suggestions. The methods and method types defined in DSPL++ for this purpose are:

**Modification Plans:** A modification plan is a compiled method for modifying a candidate design. The structure and behavior of a modification plan is similar to the design plans for proposing candidate designs, except that the elements of the plan typically change the existing values of attributes of the existing candidate rather than proposing new ones.

**Redesign Steps:** A redesign step contains the knowledge to modify the value of
a single attribute in the candidate design given a suggestion involving that attribute. Methods of this type are used in conjunction with the design plan in which the attribute value was initially set in order to consistently revise all attribute values on which the step redesigner's attribute value depends. The design plan is selectively re-executed from the point at which the attribute was set, updating all portions of the design that are dependent on the changed attribute. This method type is only applicable when the suggestion provided addresses an attribute whose value was set during design proposal by a plan proposer.

**Independent Redesign Method:** This is a domain-independent strategy for implementing modification suggestions for candidate designs that were not proposed by a design plan. When a candidate design is not generated using a plan structure, redesign steps may be independently selected and applied to the candidate design to implement a modification suggestion. Consistency in the design candidate is maintained by using the input and output dependencies of individual design steps to determine which attributes need to be modified.

**Pursuing a Faulty Design.** In the above discussion, an effort was made to modify a faulty design rather than immediately discard it and generate a new candidate design. The faulty candidate design was analyzed in the *Critique Design* task, a modification was made, and the new variation of the candidate was verified to meet the design specifications. But what happens if a problem arises in the attempt to
modify the design?

By the same reasoning that led us to the strategy of retaining a faulty candidate and modifying it to meet the design specifications, we should continue to seek modifications to the current candidate, even though at least one proposed modification has failed to make the proper adjustments.

Figure 24 shows the method that restarts the Critique Design goal in order to generate another attempt at modifying a faulty candidate design. This method is only activated from the ModifyCandidate method, and only when either a previous modification to the faulty candidate design could not be successfully implemented, or a modified candidate design failed to meet the design specifications. This method uses the RESTART imperative to resume the existing Critique Design goal, rather than creating a new goal, because of the history maintained of the modifications that have already been generated. When a new modification is successfully imple-
mented, then the method uses the SUB-GOAL imperative to generate a new Verify Design goal to examine the new design. If the Verify Design goal fails, then the TryAnotherModification method is again used to restart the Critique Design goal for another round of modifications to the candidate design. This cycle continues until the Critique Design goal fails to find a potential modification to the candidate design. The method then fails, and control is returned to the Design Artifact goal.

Discarding a Faulty Design. If an initially faulty design candidate is created, but we can identify the features that cause the specifications not to be met, and then correct those features, then the design process is again completed successfully. But what if we can’t figure out what is wrong with the design, or can’t figure out how to fix it?

If the Critique Design goal fails, then we have no idea how the design fails to meet the specifications, and the design process must terminate with failure.

Or do we? It is possible that, although this candidate design has failed miserably, there may be other candidates waiting to be discovered. Instead of giving up on the design totally, it seems reasonable to at least be sure that no other design candidates are in the wings.

Since the Propose Candidate Design goal is responsible for generating candidate designs, one strategy would be to instantiate a new copy of that goal. Unfortunately, that would simply cause the system to rehash the same candidate designs that brought it to this impasse to begin with. So, instead of spawning a new Propose Candidate Design goal, the previous Propose Candidate Design goal is restarted from its current
dormant state, a state which previous to this point was thought to be successfully completed. The newly activated goal resumes selecting from the set of relevant design proposers that it retrieved on its initial activation. The sponsors for each of the design proposers are reevaluated, this time with additional knowledge that one of their number has failed to produce a satisfactory design. A new design proposer is selected, and, hopefully, produces a new candidate design. The design process resumes as described above, acting on the new candidate design.

But what if no candidate design can be found that satisfies the specifications? Does the process continue indefinitely?

Eventually, perhaps after considerable design activity, the finite set of design proposers available to the Propose Candidate Design goal will be exhausted, and the Propose Candidate Design goal will fail during its selection phase. At this point, the system has exhausted all possible design candidates to the problem, and all reasonable variations of those candidates, and the PVCM method fails. Control is returned to the initial Design Artifact goal that initiated the design process.

This part of the PVCM strategy is illustrated in Figure 25. Note that this method is only activated after the failure of the Verify Design goal, and possibly after the failure of either the Critique Design or Modify Design goals. The APPLY-IN-FAILING-CONTEXT imperative indicates that all subsequent actions are applied to the goal in which the current failure processing is taking place, i.e., the Design Artifact goal. The first action in the context of the Design Artifact goal is the RESTART imperative, which restarts the Propose Candidate Design goal. Restarting
(FAILURE-METHOD
  (NAME FindNewCandidate)
  (COMMENT Gives up on a candidate design for a new proposal.)
  (BODY
    (APPLY-IN-FAILING-CONTEXT
      (FAILURE-METHODS OFF)
      (RESTART ProposeCandidateDesign)
      (FAILURE-METHOD ModifyCandidate FindNewCandidate)
      (SUB-GOAL VerifyDesign)))))

Figure 25: A strategy for discarding a faulty design

rather than spawning a new goal prevents the same design candidates from being continually revisited. If the Propose Candidate Design goal successfully produces a new candidate design, then the same failure strategies are reinstated for it as were for the initial (or previous) design candidate. The SUB-GOAL imperative generates the Verify Design goal, which either succeeds in verifying the design, or fails, causing the candidate to be either modified or rejected as before. The entire PVCM method fails when restarting the Propose Candidate Design goal fails to produce a new candidate design, and causing the Design Artifact goal to resume control without having satisfied its design specifications.

Once the parent Design Artifact goal has control, it may itself attempt to find another method to satisfy the initial design requirements. Since the PVCM method has supposedly covered the entire design space known to the system and failed to find any satisfactory design solution, then any remaining methods will likely fall outside the realm of design.
One method that still might reasonably be supported involves interacting with the design client to loosen the design specifications. Inspection of the failed designs will certainly provide useful guidance along these lines. Once the specifications have been relaxed, the PVCM method could be employed again on the new design problem.

Strategies even more remote from the design task are possible, such as attempting to get out of the customer contract that presented the specification (and with considerable data as to why this would be a reasonable alternative course of action), or simply pushing the problem off on another designer. Obviously, such strategies fall considerably outside the focus of this research.

4.3 The Trade Study Method for the Design Artifact Goal

While the PVCM method cycle is certainly a very basic and general purpose method for design, it is by no means unique. While the PVCM method is capable of finding a candidate design that satisfies the design specifications, if one exists, there are still potential problems.

One problem arises because the PVCM method halts after the first satisfactory design candidate is found. Thus a marginally satisfactory candidate is selected when an obviously superior design may lurk just ahead. The PVCM method encodes no intuition about the relative goodness of a candidate design, or expectations about how good a candidate design can be. Of course, it is possible that many novice human designers lack this intuition as well.

A second problem occurs in the context of very large search spaces. When confronted with the task of designing an entire system, a design engineer faces an even
broader range of design spaces to search than when designing a single device. The engineer may be faced with a range of technologies with which to implement the design. Different decompositions may produce radically different characteristics in the final design. Even when the engineer is given a single set of functions to satisfy, there may be several different schemes for producing a useful artifact. Each of these decisions still leaves a broad design spaces for the engineer to explore.

Another potential difficulty arises for our intrepid engineer when the design specifications call for one or more optimizations, i.e., produce a device with the lowest cost, or highest reliability. If the design knowledge is not already organized or indexed to address these issues, the search process can be formidable indeed.

A common thread runs through all of these examples.

1. the design space is exceptionally large

2. many candidate designs in the space at least marginally satisfy the design specifications

3. some candidate designs that satisfy the design specifications may be deemed in some sense superior to others

The Trade Study method attempts to address these issues. The basic strategy of this method is to propose several partial candidate designs, critique each design, to the extent possible, with respect to the design specifications, and proceed with the most promising candidate. In search space terminology, this method is a kind of iterative deepening of the design space.
In terms of the PVCM method described above, the selection of a design proposer within the Propose Candidate Design goal is extended to compare the results of extending several candidates, each of which start with a different choice to a key design decision.

Several new design tasks arise to handle this method.

**The Select Critical Specification Goal.** The function of this goal is to decide which feature of the specifications, or of the candidate design space, is most critical to the selection of a design approach. The input to this goal is the original set of design specifications, and possibly the set of features of the current design state that are yet undecided.

Many possible methods exist for creating and navigating the space of candidate designs. One possibility is to perform a sensitivity analysis on one or more variables in the design specification and proceed with the design based on which configuration of those variables is most promising.

**The Select Candidate Characteristics Goal.** The function of this task is to set up the alternatives that will be explored in the trade study. The input to this goal is the current candidate design and an indication of the critical feature of the design, as well as possible values to consider.

The goal must select several values to explore. The output of this task is a list of trade study candidates to be explored by the system.
The *Extend Candidate Design Goal*. The function of this goal is to extend each of the trade study candidates, not to completion, but until some indication of each design’s worth relative to the critical features of the design can be established.

The *Evaluate Partial Design Goal*. Each of the trade study candidates are examined for their merit relative to the critical specification. The input to this task is one of the partially completed trade study candidates, and the output is a measure of the worth of the candidate relative to the critical feature of the trade study.

The function of this task parallels that performed by the *Sponsor* in the selector mechanism of the basic control architecture.

The *Select Trade Study Candidate Goal*. This task parallels that of the *Select* goal in the basic architecture. Its inputs are a list of suitabilities of each of the trade study candidates, similar to the basic selection phase.

The tasks handled by this goal could also be subsumed within the operation of the standard selection mechanism.

The methods needed to explore these spaces have not been investigated. Certain pieces of the trade study additions will certainly map into existing portions of the DSPL++ architecture. The extending of a trade study candidate, for example, will likely use the same design proposers found used in the *Propose Candidate Design* goal.

A useful tool can be obtained by simply interacting with a design engineer for any operations unknown to the architecture. The analysis of the trade study method
above allows the system to present the relevant issues to the engineer in a meaningful fashion, and can direct the engineer through the design process. When methods are available to the system, it can resume its automatic control.

The design proper, that is, the development of a design candidate once a likely direction is established, proceeds in the same manner as described in the PVCM method. The basics of design will not likely change regardless of how the design space is navigated. The same knowledge types are used in the same fashion as before.

4.4 Other Methods for the Design Artifact Goal

The methods described above certainly do not exhaust the possible combinations of strategies for design. Domains may require specific subtasks not mentioned here, and more comprehensive methods will surely be developed. Rather, these methods are meant to represent a foundation upon which other strategies can expand and develop.

Chapter V describes how additional design strategies can be adapted to the DSPL++ system.

4.5 Methods That Propose Designs

One important category of methods in DSPL++ are those that suggest or propose a solution to a portion of a design problem. Methods of this type are called proposers. Given a set of requirements to a design problem or subproblem, a design proposer produces a single, internally consistent, specific design that purports to satisfy those requirements. Since the production of new designs is the activity most often associated with design as an activity, it is not surprising that design proposers organize
most of the obviously design related, domain-specific knowledge in the architecture.

Each proposer has domain knowledge of a certain type. Each type of design proposer is defined by the form of its domain knowledge and the control mechanism used to manipulate that knowledge. The control scheme for each proposer is embedded in the proposer, i.e., decomposition behavior is embedded only in decomposition proposers, plan behavior is embedded only in plan proposers, etc. This simplifies the interpreter and makes it modular and extendible. Design proposers are activated by design goals, and may in part handle their design task by generating other goals to activate other design proposers.

Design proposers are indexed by the name of the artifact that is being proposed and the type of the method.

Many different strategies can be used to generate design candidates, so it is not surprising that design proposers may be represented in a number of different forms. The rest of this section describes several different types of methods that can propose designs: task decompositions, plans, primitive plans, and cases.

**Task Decompositions.** One method of developing a design is to break the design problem into a number of smaller problems and solve each of those in turn. This is the purpose of the task decomposition in DSPL++. These methods are not same as a device decomposition of a complex device, which describes the subcomponents that make up a device. Task decompositions and device decompositions are often related, however, since one strategy for the design of a complex object is to design each of its pieces.
The types of knowledge specific to a decomposition method are:

1. knowledge about how to map the requirements of the current problem into those of the subproblems,

2. a list of subproblems whose solutions can be composed to satisfy the current design problem, and

3. knowledge about how to map the solutions of the subproblems into the context of the current problem, that is, how to compose a solution of the current problem from those of the subproblems.

These parts are depicted pictorially in Figure 26.

The decomposition lists a set of design problems. When the decomposition is applied, goals for each are created and appended to the current context.

Figure 27 shows a decomposition of a sieve tray distillation column, taken from the STILL design system. The example shows that there are three subtasks to the design of a distillation column:

1. the Section design task,

2. the Reboiler design task, and

3. the Condenser design task.

There are no preconditions specified in the decomposition. One postcondition, CheckReboilerCapacity, insures that the subtasks have produced a workable design solution.
Figure 26: The structure of a decomposition
(DECOMPOSITION
  (NAME SimpleDistillationColumn)
  (COMMENT Lists the major subsystems of
        a simple sieve tray distillation column. )
  (SPONSOR SieveTrayColumnProposer)
  (PRECONDITIONS None)
  (SPECIFICATION-MAPPERS SectionTemperature ... )
  (BODY DistillationColumn --> Section Reboiler Condenser)
  (SOLUTION-MAPPERS MaximumThouhgput TotalEnergyConsumption ... )
  (POST-CONDITIONS CheckReboilerCapacity ))

Figure 27: A decomposition

(SPECIFICATION-MAPPER
  (NAME SectionTemperature)
  (BODY
    (KNOWN
      Tf (KB-FETCH 'DistillationColumn 'FeedTemperature))
    (DECISIONS
      REPLY (KB-STORE 'Section 'FeedTemperature Tf))))

Figure 28: A specification mapper

Specification Mappers. When a decomposition is used, the state of the cur-
rent design must be mapped into the requirements of each of the subproblems de-
scribed in the decomposition. The specification mapper is the method that maps the
requirements of a design problem into the requirements of a sub-problem.

Figure 28 shows an example of a specification-mapper that translates one of distil-
ation column requirements into a needed requirement of one of its sub-components.
The temperature of the feed flow, indicated by the FeedTemperature attribute of the
distillation column, is an input requirement to the design problem. This attribute is used in the design of each of the sections of the distillation column. The specification mapper listed fetches the value of this attribute from the distillation column description, and stores it in the corresponding attribute of the column’s section. In this case, the value is mapped into the sub-context of the distillation column without change.

**Solution Mappers.** When each of the subproblems described in a decomposition have completed, the resulting designs are pulled together into a solution for the parent problem. The solution mapper method represents the knowledge necessary to map the solution of a sub-problem into the parent context.

An example of a solution mapper is shown in Figure 29. The interpretation of this piece of knowledge is that the maximum throughput of the distillation column is
limited by the smallest maximum throughput of each of its components. It is executed after each of the components listed in the decomposition have successfully completed, and demonstrates how attributes of a completed component of a device are used to determine the attribute values of the device itself.

**Plans.** The most obvious method for proposing a design is to follow a plan whose actions can result in the creation of the desired object, or at least make some progress toward its creation. A design plan is a sequence of design whose execution results in a fully elaborated instance of the desired device.

The actions that compose a plan include:

- Design steps, the primitive method for setting the value of a single attribute in a design,

- Primitive plans, which organize collections of related design steps,

- Invocations to design a subcomponent of the desired device. This action causes an new *Design Artifact* goal to be spawned, and

- Constraints, to check the intermediate progress of the design process.

The plan assumes that the input parameters expected by each subproblem are available and no checks are made during problem solving to verify that assumption. It is the knowledge engineer's responsibility to insure that the requirements for the subproblems dovetail correctly. Support for building correct plans is available to the knowledge engineer by "symbolically simulating" the plan when it is entered into the
Figure 30: A plan

system. This simulation checks that design attributes needed by one plan component are made available by previous plan components. The "symbolic simulation" features and other aspects of knowledge acquisition in DSPL++ are discussed in Chapter V.

Figure 30 shows an example of a simple plan for the design of the section of distillation column, taken from the STILL system. A sponsor for this plan is explicitly listed for this plan. The plan body lists three subproblems in the design of a distillation column section, each of which will cause the DSPL++ architecture to set up a new problem space in which to work. Within each subproblem, the knowledge base is searched for appropriate methods to apply to the new context. Those methods are applied to the subproblem until the requirements of the subproblem are satisfied, at which point control is returned to the plan context.

Primitive Plans. A Primitive Plan is a related but distinct form of a plan in which the plan elements are restricted to be elemental design decisions, such as the deter-
(PRIMITIVE-PLAN
(NAME FinalTrayDesign)
(BODY
TraySpacing
DowncomerArea
TotalTrayArea
TrayDiameter
(TEST-CONSTRAINT Spacing)
(PRIMITIVE-PLAN DetailedTrayDesign)
(TEST-CONSTRAINT ConvergedActiveArea))

Figure 31: A primitive plan

mination of single attribute values and possibly constraint checks on those values, and direct reference to other primitive plans.\(^2\) This type of method defines an important subset of methods because a range of problem solving techniques are available to handle impasses that are computationally infeasible for plans in general. These techniques parallel the failure handling methods for tasks in DSPL.

From the task analysis point of view, a primitive plan typically represents the development of a collection of tightly related attributes in one component or section of the design. The primitive plan represents a monolithic, although well-structured, collection of design knowledge. The primitive plan is applied as a single, complex operator to the design state, and no mechanism is available to effect the flow of control during problem solving.

\(^2\)The equivalent method in DSPL is the task. The motivation of the terminology shift in this work is twofold. First, the term primitive plan suggests fact that these methods are related, both in form and function, to plans. Second, the term “task” is already somewhat belabored, and the unfortunate naming conflict in DSPL has always been a source of confusion.
Figure 31 shows an example of a primitive plan from the STILL system. This method coordinates the design of the attributes of a single tray in a distillation column.

The structure of a primitive plan is similar to that of a plan, except that the references in the primitive plan body are limited only to primitive design methods or other primitive plans. Thus, no mechanisms are available to invoke any explicit control mechanisms to perform any explicit search. The tasks in DSPL differ from primitive plans in DSPL++ in that the former does not allow selection knowledge to be associated with them.

A primitive plan is indexed both by the type artifact that it modifies and the task or feature set that it develops. In the AIR-CYL system all primitive plans are referenced directly through plans.

Cases. A case is a method that proposes a previously designed example of a device as a candidate design. A case contains all of the specifications and all of the fully elaborated components of a device that resulted from the design specifications. Cases are indexed by both the device type represented by the case, as well as the specifications used in the creation of the device.

A case may be entered and indexed interactively by the user, but are also generated from previous design runs. At the end of a successful design run, the design specifications and completed, verified device may be entered into the knowledge base as case.

Figure 32 demonstrates the structure of a case. Note that structure is automati-
(CASE
(NAME SampleSieveTrayColumn)
(COMMENT Lists all of the components of a previous design run.)
(SPONSOR SampleSieveTrayColumn)
(PRECONDITIONS None)
(REQUIREMENTS-LIST
  (FeedTemperature 180)
  (FeedFlowRate 350)
  etc...)
(CONTENTS
  (COMPONENT DistillationColumn
    (NumberOfPlates 12)
    (FoamingTendency moderate)
    etc...)
  (COMPONENT Section
    (FeedTemperature 180)
    etc...)
  (COMPONENT Reboiler
    etc...))
(POST-CONDITIONS None))

Figure 32: A case
cally available whenever the results of a design run are saved in the knowledge base.

4.6 Methods for Primitive Design Actions

The purpose of these methods is to make a commitment about some concrete feature of the progressing design. The selection of a primitive domain object to be included in the finished design, the calculation of an attribute value, or the determination of the relationship between components are typical actions at this level.

All other methods in DSPL++ mentioned to this point are control structures that only indirectly modify the developing design, either by shifting the focus of the design context or by controlling the invocation of subordinate methods. Primitive design methods are distinguished by their lack of control knowledge. These methods make their decisions based only on the current design context, and thus only indirectly rely on the results of other problem solving methods. They cannot directly invoke any other methods, and cannot initiate any subordinate problem solving behavior. These methods represent the "bottoming out" of the design problem solving mechanism.

This does not mean that these methods necessarily implement trivial pieces of computation. These methods may involve, for example, considerable numerical processing for solving a set of simultaneous equations in order to determine some aspect of the design. The important point is that, whatever the size of the computation embodies by a primitive method, the computations are algorithmic in nature. No additional problem solving (ie, search) is called for.

Primitive design methods are indexed by the name of the attribute that they compute as well as the name of the component to which the attribute belongs. Since
these methods have an index, they can be invoked through search using the sponsor/selector method. One scheme using this behavior to produce plan-like behavior is illustrated in Chapter V. Typically, however, these methods are invoked by direct reference from tasks. This mode is analogous to the use of steps in tasks in the original AIR-CYL system.

The possible range actions taken by different members of this class of methods is potentially large and very dependent on the domain of discourse. In routine mechanical design, for example, primitive design actions involve computing the physical dimensions of objects, computing loads and stresses on these objects, and perhaps choosing the material for a component[Brown, 1984] Other domains require other primitives. The primitive actions of circuit design could involve the selection of components such as capacitors, resistors, etc., determining how these components are connected, and perhaps computations concerning current relations and such[Sussman, 1978]. There is no single language that can concisely express actions in every design domain.

The DSPL++ system allows a somewhat broader range of primitive actions than the original DSPL language. The complete spectrum of primitives is presented in Appendix B. However, for illustrative purposes the examples found in this section contain only algorithmic formulas.

**Design Steps.** The design step in DSPL++ is essentially identical to the step agent found in DSPL. The description and example shown here are provided for the sake of completeness and to emphasize the differences between the two constructs.
A design step is a primitive design method in which the knowledge to make a single decision about the value of a single attribute in the evolving design is encoded as a set of mathematical equations or logical alternatives. The design step can retrieve values of attributes and other status about the evolving design through the design context, and make calculations or choices based on those values. A design step always completes with the storing of a value of a single attribute in the design database.

A number of other methods can be used to assist the design step's processing. A set of initial constraints are used to ascertain that the preconditions are met for this piece of knowledge. Final constraints determine if the value computed by the step is locally acceptable. One or more redesigners are used to "tweak" the design step value if it is determined, either locally or externally, to be wanting in some fashion. Finally, a set of suggestions are used to direct redesign in the parent method in the event that no acceptable value can be determined by the design step.

The design step, like any other method in DSPL++, may have an optional sponsor associated with it. However, design steps that are referred to directly from the body of a plan or primitive plan have no need for a sponsor, since they are not retrieved from the knowledge base using the Retrieve/Select/Apply cycle.

The syntax and semantics of the design step are essentially identical to those of the step design knowledge found in DSPL, with the following exceptions:

- Constraints are no longer allowed in the body of a step, in favor of listing any necessary constraints in the postconditions to the step.

- In DSPL++ a design step body always terminates with a value being stored in
the design case.

- Failure handlers are no longer listed with a design step, having been replaced with the failure handling mechanism embedded in the goal interpreter.

- The syntax for identifying the design step’s attribute name and the clients of step have been eliminated since this information is now maintained automatically by the DSPL++ language system.

- An optional sponsor may be associated with a design step.

An example of a DSPL++ design step taken from the STILL design system [Myers et al., 1988] is shown in Figure 33. The BODY of the step retrieves the values of several attributes of the Plate, and uses these values to compute the value of the DowncomerArea of the Plate. The BODY completes with a KB-STORE, which stores the computed value in the design database. A redesign step is also associated with this design step.

### 4.7 Methods That Verify Designs

Verifiers are methods that examine a candidate design and decide whether or not it meets its design specifications. Two method types are provided in DSPL++ for representing knowledge of this type: constraints and verification plans.

**Constraints.** The constraint construct in DSPL++ is taken almost unchanged from the constraint agent found in DSPL. The description and example of the constraint are provided here primarily for the sake of completeness.
(DESIGN-STEP
(NAME DowncomerArea)
(COMMENT Dependent on flow rates, densities, and active area )
(REDESIGNER DowncomerArea)
(BODY
(KNOWN
   Aa  (KB-FETCH 'SieveTray 'ActiveArea)
   LGPM (KB-FETCH 'SieveTray 'LiquidFlowRate)
   Ff  (KB-FETCH 'SieveTray 'FloodFactor)
   Sf  (KB-FETCH 'SieveTray 'DeratingFactor)
   Ts  (KB-FETCH 'SieveTray 'Spacing)
   Pl  (KB-FETCH 'SimulatedSieveTray 'LiquidDensity)
   Pv  (KB-FETCH 'SimulatedSieveTray 'VaporDensity))
(DECISIONS
  Vd  (SMALLER (* 250.0 Sf)
       (SMALLER (* 41.0 (* Sf (SQRT (- Pl Pv)))))))
       (* 7.5 (* Sf (SQRT (* Ts (- Pl Pv))))))
  Adp (/ LGPM (* Vd Ff))
  Ad  (LARGER Adp
       (SMALLER (* Aa 0.11)
          (* Adp 2.0)))
REPLY (KB-STORE 'SieveTray 'DowncomerArea Ad))))

Figure 33: A step
A constraint is a test to see that the value of some attribute of evolving design falls within some criteria. It retrieves attribute values from the design state, performs any necessary calculations or comparisons on those values, and and either accepts or rejects the current design state. Constraints are used to recognize unacceptable design states and allow the course of design to change based on that recognition. A constraint does not otherwise change the state of the design.

The FAILURE-MESSAGE clause is a string that characterizes the design state when the constraint fails. A list of FAILURE-SUGGESTIONS may be provided as a compiled means of immediately criticizing the unacceptable design. The body of a constraint consists of a KNOWN section, where values from the design database and the design context are identified. A TEST indicates a Boolean expression involving values retrieved or computed in the KNOWN section.

When the constraint is applied during problem solving, the values in the KNOWN section are retrieved, and the Boolean expression is evaluated. If the expression evaluates to “true”, then the constraint succeeds. If the expression evaluates to “false”, then the constraint fails. If the constraint succeeds, then problem solving continues normally. When a constraint fails, the message in the FAILURE-MESSAGE clause as well as the suggestions in the FAILURE-SUGGESTIONS are evaluated and attached by the interpreter to the current goal’s history, and control is passed indicating the constraint’s failure to the method or task that invoked the constraint.

Premature failure of the constraint may occur if one of the values in the KNOWN section are unavailable, or unable to be computed. In this case both the
(CONSTRAINT
   (NAME 'Tray spacing and diameter compatible?')
   (COMMENT Checks to see if the tray spacing is appropriate.)
   (FAILURE-MESSAGE The current tray spacing does not match the tray diameter.)
   (FAILURE-SUGGESTION (CHANGE Spacing OF SieveTray))
   (BODY
      (KNOWN
         ts  (KB-FETCH 'SieveTray 'Spacing)
         Dt  (KB-FETCH 'SieveTray 'Diameter)
         tsb (TABLE
            (DEPENDING-ON Dt)
            (MATCH
               (IF ((< 3.0)) THEN 12.0)
               (IF ((< 5.0)) THEN 18.0)
               (IF ((< 6.0)) THEN 24.0)
               (IF ((< 8.0)) THEN 30.0)
               (OTHERWISE (FAIL)))
            (TEST (= ts tsb)))
      )
   )

Figure 34: A constraint

FAILURE-SUGGESTIONS and FAILURE-MESSAGE are discarded, since they are not relevant to the failure of a constraint in the manner.

Figure 34 shows a constraint from the STILL system. This constraint checks that two attributes of a plate in a distillation column section are mutually compatible. An optimal tray spacing, tsb, is determined from the current tray's diameter, using a TABLE construct to match one of four possible situations. This optimal value is then compared to the current value for the tray spacing, ts. Note that if the optimal tray spacing cannot be determined, as indicated by the OTHERWISE clause of
(VERIFICATION-PLAN
  (NAME CheckFinalTrayDesign)
  (TYPE Consistency)
  (DOMAIN DistillationColumn)
  (SPONSOR Default)
  (FAILURE-MESSAGE "The tray design is inconsistent."
  (FAILURE-SUGGESTIONS (CHANGE TraySpacing))
  (BODY
    (TEST-CONSTRAINT Spacing)
    (TEST-CONSTRAINT ConvergedActiveArea))

Figure 35: A verification plan

the TABLE construct, then the constraint fails prematurely and the FAILURE-MESSAGE
and FAILURE-SUGGESTION are discarded. Such a failure does not represent an incom-
patibility of the tray spacing and diameter, but rather indicates an “out of range”
condition for this particular piece of domain knowledge.

Verification Plans. A verification plan is a method that verifies some perspective
of a candidate design with respect to the design specifications.

The structure of a verification plan is essentially the same as a primitive plan,
except that the elements of the plan are further restricted to only include constraints
and other verification plans. A sponsor is used to determine the whether or not the
verification perspective encoded by the plan is suitable in the current context. The
FAILURE-MESSAGE and FAILURE-SUGGESTIONS clauses perform the same function as
in the constraint method. Figure 35 shows an example verification plan.
4.8 Methods That Critique Faulty Designs

Methods that critique faulty designs in DSPL++ generate suggestions for correcting a candidate design’s deficiencies.

A *suggestion* is a data structure in DSPL++ used to communicate information about how a faulty candidate design can be modified to meet specifications. It consists of a type, an attribute, and a set of optional parameters that elaborate the suggestion’s type.

Both the type and attribute of a suggestion are used in indexing methods from the knowledge base that are relevant to implementing the suggestion. Note that *any* token may be specified as a suggestion type or attribute. They are simply used as additional indexes for accessing appropriate methods for implementing suggestions for modifications to candidate designs.

The attribute of a suggestion can refer to either an actual attribute listed in a device definition of the candidate design, such as the "TraySpacing" attribute of the Plate device, or to a "virtual" attribute of the candidate, i.e., one that is not explicitly associated with the design, such as it “Cost”. Thus, the construct

\[
\text{SUGGEST (DECREASE Cost OF Plate)}
\]

is a valid suggestion even if “Cost” is not an attribute in the Plate device definition. This implies, however, that some form of modification knowledge exists that is indexed by “Cost”.

In addition to methods that produce suggestions in support of the critique task, suggestions may also be explicitly listed within a method in a FAILURE-SUGGESTIONS
clause. When the method fails, the FAILURE-SUGGESTIONS are passed along to the Resolve Failure goal in the event that a failure handler exists that can take advantage of the suggestions to direct the processing of the failure.

**Suggestion Maps.** A suggestion map produces one or more suggestions for modifying the candidate design based on the current design state.

The suggestion map consists of a body that retrieves known values from the design database, and performs calculations and matching on those values to determine an appropriate set of suggestions for modifying the design candidate. A set of suggestions are returned by the method to the problem solving context. The method fails if no suggestions are generated.

The map is indexed by the device that the knowledge in the map addresses. A sponsor may be attached to the method to further discriminate the situations in which the method is applied.

Figure 36 shows a suggestion map for reducing the overall envelope of an air cylinder based on the current envelope. A sponsor is used to determine if the current design state will benefit from the knowledge encoded in this method. The current envelope parameters are retrieved from the design database in the KNOWN section, and suggestions for new values are organized by the TABLE construct in the DECISION section. The first pattern in the TABLE indicates that the method fails if the current length of the air cylinder is less than a minimum value.
(SUGGESTION-MAP
(NAME ReduceEnvelope)
(DOMAIN AirCylinder)
(TYPE Fictional)
(COMMENT Finds a suggestion for modifying the size of a device based on its current parameters.)
(SPONSOR ExcessiveEnvelopeCheck)
(BODY
(KNOWN
  Length (KB-FETCH 'AirCylinder 'Length)
  Width (KB-FETCH 'AirCylinder 'Width)
  Height (KB-FETCH 'AirCylinder 'Height))
(DECISIONS
  REPLY
  (TABLE
    (DEPENDING-ON
      (Length) (Width) (Height))
    (MATCH
      (IF ( <10 ? ? ) THEN (FAIL))
      (IF ( <15 >5 ? ) THEN
        (SUGGEST (DECREASE Width BY 2)))
      (IF ( <15 ? >10 ) THEN
        (SUGGEST (DECREASE Height BY 1)))
      (IF ( <15 ? >10 ) THEN
        (SUGGEST (DECREASE Length BY 3)))
      (OTHERWISE (SUGGEST
        (DECREASE Width BY 2)
        (DECREASE Height BY 1)
        (DECREASE Length BY 3))))))

Figure 36: A suggestion map
4.9 Methods That Modify Designs

A modifier is a method that changes an existing candidate design based on a suggestion generated by a Critique Design goal. Modifiers are indexed according to the type of suggestion that they implement. Thus, within a Modify Design goal, only those methods that address the specific suggestion provided will be considered during the selection process. There are two types of methods and one specific method for implementing modification suggestions. These are described below.

Redesign Steps. The redesign steps found in DSPL++ are also similar to the corresponding step redesigners found in the original DSPL work, however in DSPL these agents were structured as part of a design step. In DSPL++, these constructs are viewed as separate entities from design steps, and the description and examples are provided primarily to illustrate the distinctions between the two works.

A step redesigner is a method that proposes a change to the value of a single attribute in the design, typically based on design suggestion provided to it. The value of the attribute modified by the step redesigner must be previously set by an earlier decision in the design process. The function of the redesign step is to change the value of an attribute sufficiently that design suggestion is satisfied and the candidate design can be successfully verified. A redesign step is indexed according to the attribute that it addresses. Thus, the index for a redesign step must always refer to an existing attribute of an instantiated device in the design database.

The structure and behavior of a redesign step is similar to that of the design
step. Values are retrieved from the design database in the KNOWN section of a step, and calculations are made on those values and stored in the DECISION section. The redesign step always completes with a KB-STORE, which stores the value of an attribute in the design database. However, there are several important distinctions between design steps and redesign steps.

1. Unlike design steps, redesign steps may base their calculations on a previous value of the attribute that they compute. Redesign steps are not used when a new candidate design is being proposed, only to change an existing one.

2. Redesign steps recognize an optional minimum allowable change that would be expected to make a difference during design. If a suggestion is received by a redesign step to make a change smaller than the minimum, then the minimum value is used instead.

3. Redesign steps are directed by suggestions for redesign generated from the task that initiated the redesign process. Redesign steps are keyed only to certain types of suggestions, and only respond with redesign actions when a suggestion of the appropriate content is provided.

There are three situations in DSPL++ in which redesign steps may be invoked:

**Directly referenced by a design step:** The redesign step is used directly by a design step when the design step discovers that the attribute value it computed is unsatisfactory. This mode implements the step redesign of DSPL.
(REDESIGN-STEP
  (NAME Spacing)
  (COMMENT Changes the tray spacing based on the tray diameter.)
  (TYPE CHANGE)
  (BODY
    (KNOWN
      Dt (KB-FETCH 'SieveTray 'Diameter)
    )
    (DECISIONS
      Spacing (TABLE
        (DEPENDING-ON Dt)
        (MATCH
          (IF ((< 3.0)) THEN 12.0)
          (IF ((< 5.0)) THEN 18.0)
          (IF ((< 6.0)) THEN 24.0)
          (IF ((< 8.0)) THEN 30.0)
          (OTHERWISE (FAIL)))
      )
    )
    (REPLY (KB-STORE 'SieveTray 'Spacing Spacing))
  )
)

Figure 37: A step redesigner

Directly referenced by a primitive plan: A suggestion for modifying a candidate design directed at a primitive plan is used to initiate redesign within the plan. The plan structure is used to determine which portions of the design candidate must be modified. This mode implements the task redesign of DSPL.

During Independent Redesign: When attribute design is not organized by a design plan, redesign steps may be invoked individually to modify a design candidate. Changes are propagated through the candidate design by following the input and output dependencies of design and redesign steps.

Figure 37 shows an example of a redesign step written in DSPL++. The TYPE
clause indicates that this redesign step is relevant for implementing CHANGE suggestions. The TABLE construct is used to decide the value of the tray spacing of the plates in a distillation column. The value is dependent on a single attribute value in the design, the tray diameter, which is fetched from the design database in the KNOWN section of the step. Like all steps, the example ends with a KB-STORE that saves the value of the step’s attribute, in this case the tray spacing attribute, in the design database.

**Modification Plans.** A modification plan is a method that implements a suggestion through the actions of a compiled sequence of actions.

Unlike a redesign step, the suggestion type associated with a modification plan does not need to refer to a known attribute of a device in the design database. For example, although "cost" is not an attribute of a "Plate", the construct

(SUGGEST (REDUCE Cost OF Plate))

is a valid suggestion that invokes a correspondingly indexed modification plan.

The structure of a modification plan is again similar to the structure of a design plan, except that redesign steps may also be included as part of the plan, and a DISCARD imperative may be used to remove portions of the candidate design that must be designed from scratch.

A modification plan fails if either an attempt is made to modify attributes of a candidate design that have not been set, or an attempt is made to propose values for portions of the design that already have values.
(MODIFICATION-PLAN
  (NAME PlateCostReduction)
  (IMPLEMENTS (REDUCE Cost OF Plate))
  (COMMENT Reduces the cost of a section by increasing
    the spacing between plates thus reducing the
    number of plates required.)
  (SPONSOR Default)
  (FAILURE-MESSAGE "'The plate cost could not be
    reduced by increasing the tray spacing.'")
  (FAILURE-SUGGESTIONS NONE)
  (BODY
    (REDESIGN-WITH-SUGGESTION (Increase TraySpacing BY 1.0))))

Figure 38: A modification plan

Figure 38 shows an example in DsPL++ of a modification plan. This plan serves
to map an abstract suggestion for modifying the design into a concrete strategy for
changing the value of an attribute of the design to satisfy it.
CHAPTER V

The Extensible Design Shell

The entire DSPL++ architecture is predicated on the assumption that it is not possible to anticipate every aspect of the design task, and that new domains may well require various extensions to the task description provided with the DSPL++ architecture.

Toward this end, the DSPL++ architecture includes a task description that provides the user with a framework for addressing a range of design domains, and also provides high-level tools to extend the architecture along a number of different dimensions. Our goal in this effort is twofold. We wish to provide the flexibility of a general-purpose architecture, but without sacrificing the high-leverage benefits of a task-specific architecture. This chapter describes how the DSPL++ architecture can be extended by the user, and how the often conflicting issues of flexibility and high-leverage, task-specific features are resolved in the architecture.

The issue of extensibility is addressed at several levels in the DSPL++ architecture. These levels are summarized in Figure 39. The baseline level of interaction in DSPL++ provides the user with a description of the design task and a number of method types for encoding knowledge about design. This level involves the ability to add new instances of existing method types that describe knowledge in a target domain. The
Level 0: Required additions to build a knowledge base:
  • Adding new instances of existing method types
  • Adding new device definitions

Level 1: Anticipated extensions for increasing domain and task coverage:
  • Adding new task definitions from existing system behaviors
  • Defining new method types by recombining existing behaviors

Level 2: Unanticipated extensions that fit within the original architecture:
  • Extending the vocabulary of primitive behaviors of the system

Level 3: Unanticipated extensions, that may or may not fit within the original architecture:
  • Unrestricted additions and/or modifications to the system architecture

Figure 39: Levels of extensibility

system uses its knowledge of the structure of a known method type to focus the knowledge acquisition process. Section 5.1.2 describes this level of interaction with the user, describing how additional methods of existing types are captured by the DSPL++ system in a very "user friendly" way, using the system's knowledge of the design task and the known structure of the method. This section includes descriptions of task-specific techniques for assisting the user in knowledge acquisition tasks.

DSPL++'s description of the design task is far from complete. At the next level of detail, DSPL++ provides for ability to extend the architecture by allowing the user to define new task and method types. Extensions to the architecture at this level are supported almost as completely as defining new method instances at the
previous level, but because of the open-ended nature of the problem, it is not possible to provide to the user with much more than an open access to the same primitives used to build the existing task and method types. The second part of this chapter, beginning with Section 5.2, describes how additions are made to the existing DSPL++ repertoire of method types and design tasks.

The next level of extensibility is characterized by an ability to cope with unanticipated extensions to the architecture, but that are still compatible with the architecture. Since these changes are unexpected, it is impossible to provide the user with any guidance beyond that provided by the principles of the architecture itself. Extensions of this nature are discussed in Section 5.4.

The last level of extensibility, which includes any unrestricted modification to the architecture, is provided primarily for experimentation with the architecture. At this level, modifications are implemented by escaping to the underlying Lisp code and implementing directly in Lisp. The task-oriented approach to design presented by Chandrasekaran [Chandrasekaran, 1990a, Chandrasekaran, 1989] and described in Chapter II is sufficiently broad that the application of the DSPL++ architecture to typical design domains should rarely require the user to resort to this level of modifications.

5.1 The Intelligent Design Apprentice

The first problem confronting the builder of a knowledge based system, regardless of the representation mechanisms used for system construction, is the specification of knowledge from the target problem domain sufficient to solve problems in the domain.
This process of mapping concepts from the a problem domain into structures that can be represented within a machine architecture is called Knowledge Acquisition.

The ability to create a knowledge based system with acceptable performance in a problem domain depends primarily on the ability of the problem solving architecture to represent the important concepts in the target problem domain. This ability in turn rests partially on the architectural primitives available in the architecture, and equally importantly on the ability of the user to access and manipulate these structures effectively during knowledge acquisition. Up until this point we have focused on the features available in the DSPL++ architecture for representing design knowledge and the design task. However, without effective access to these architectural features, the effectiveness of the architecture is lost. This section demonstrates how the structures described in the previous chapters are utilized during knowledge acquisition.

5.1.1 Checking Method Consistency

An important aspect of building a knowledge base system is maintaining the consistency of the information within the knowledge base. That is, when used in a problem solving context, is the knowledge within the system capable of producing a satisfactory solution to the problem, or do the various pieces of knowledge incompatible or conflicting in their contribution to the task?

In general the maintenance of virtually any characteristic of a knowledge base such as consistency is at best beyond the reach of current technology, and at worst theoretically intractable. However, by limiting the scope of our demands on the characteristics of a knowledge base, certain inroads into knowledge base maintainability
can be made.

Within the design architecture described thus far, certain guarantees can be made about certain limited subsets of the methods entered into a DSPL++ knowledge base. This is done by taking advantage of the device structure that DSPL++ requires to be available before methods referencing that device structure can be entered into the system, and by indexing methods in the knowledge base by references to the attributes of a device. With this information, the input from the design database required by a method as well as the changes made to the design database by a method can be analyzed to insure that no attributes are used by a method before they are available. The analysis is accomplished by symbolically simulating the execution of a method, or a set of methods, and checking at each access to the design database whether or not the accessed attribute is available.

Figure 40 shows the MaxThroughput Solution Mapper, a piece of knowledge from a decomposition in the STILL knowledge base. This piece of knowledge is used within a distillation column decomposition to determine the throughput of the column based on the throughput of each of its serially connected components. Figure 41 shows the output resulting from symbolically simulating the MaxThroughput Solution Mapper. The simulation lists the input attributes required by this particular solution mapper, and which attribute is set. Obviously, the input corresponds to attributes referenced in the KB-FETCH commands, while the output corresponds to the attribute from the KB-STORE.

This rather trivial example demonstrates a means for the knowledge engineer
(SOLUTION-MAPPER
  (NAME MaxThroughput)
  (KNOWLEDGE-PARTITION STILL)
  (COMMENT Determine the MaxThroughput of the DistillationColumn
       from sub problem solutions)
  (BODY
    (DEPENDS-ON
      CondenserMaxThroughput
        (KB-FETCH (QUOTE Condenser) (QUOTE MaxThroughput)))
     ReboilerMaxThroughp
        (KB-FETCH (QUOTE Reboiler) (QUOTE MaxThroughput)))
     StrippingSectionMaxThroughput
        (KB-FETCH (QUOTE StrippingSection) (QUOTE MaxThroughput)))
     EnrichingSectionMaxThroughput
        (KB-FETCH (QUOTE EnrichingSection) (QUOTE MaxThroughput)))
    (DETERMINED-BY
      DistillationColumnMaxThroughput
        (SMALLER CondenserMaxThroughput
           (SMALLER ReboilerMaxThroughput
              (SMALLER StrippingSectionMaxThroughput
                 EnrichingSectionMaxThroughput)))
    REPLY
        (KB-STORE (QUOTE DistillationColumn) (QUOTE MaxThroughput)
                   DistillationColumnMaxThroughput)))

Figure 40: A solution map from STILL
Simulation results for MaxThroughput SolutionMapper:

Input required:

- (Condenser . MaxThroughput)
- (EnrichingSection . MaxThroughput)
- (Reboiler . MaxThroughput)
- (StrippingSection . MaxThroughput)

Output set:

- (DistillationColumn . MaxThroughput)

Used before set:

None

Unreferenced:

None

Figure 41: The simulation of a solution map

to quickly summarize the input/output behavior of a method without resorting to inspecting the actual source code. However, symbolic simulation is not limited to this level of analysis. Considerably more leverage is provided by symbolic simulation of complex method, such as a plan or decomposition, which may contain a large number of such atomic pieces. An example of this is provided in the next section.

5.1.2 Acquiring Methods

The current paradigm for building knowledge based systems calls for an expert within the target problem domain to interact with an expert in the domain of knowledge representation, that is, the knowledge engineer. However, by providing a sufficiently "user friendly" interface to the problem solving architecture, it should be possible
to eliminate, at least in certain situations, the need for the extra level of expertise between the domain expert and the system architecture. By providing sufficiently understandable tools, the role of the knowledge engineer is essentially shared between the computer and the domain expert himself.

The crucial difficulty in eliminating the knowledge engineer's distinct role in the creation of a knowledge based system, especially in a task-oriented system, is the plethora of concepts, their complexity, and the terminology used to describe these concepts that the knowledge engineer must internalize before a system can be created. The complexity of a task-oriented knowledge based tool is usually reflected in a complicated or extensive language syntax. For tools that employ some sort of uniform representation scheme, a complicated syntax is replaced by the need to attend to many implicit details to control the architecture. Either situation requires considerable expertise before a user has a chance to successfully construct a working knowledge base system.

There are a number of possible approaches to solving these problems. Brown and Chiang [Chiang, 1987, Chiang and Brown, 1987] describe a DSPL acquirer that eliminates the need for the knowledge engineer to work with that language's syntactic forms. Instead, the user is led through an organized series of questions that result in the creation of the forms needed for a working DSPL system.

The approach taken by DSPL++ is similar to the work by Brown and Chiang in that it, too, eliminates the need for the knowledge engineer to work with the DSPL++ syntactic forms (which bear many similarities to those of DSPL), for most situations,
in the creation of a design system. However, rather than direct the user through a set of linear questions and answers to generate these structures, the DSPL++ knowledge acquisition scheme is controlled rather more directly by the user through a series of command menus that allow the knowledge acquisition to proceed along several fronts simultaneously.

The acquisition of new methods in DSPL++ is initiated in two different situations:

1. explicitly by the user, during system creation, and

2. whenever problem solving is halted due to missing knowledge in the system.

In the first case, the user must supply all necessary information to the system about the design knowledge being acquired. In the latter case, the system uses whatever context information is available to supply default values for the new method.

5.1.3 Acquiring a Decomposition

This section provides an example of how the structure of a complex method, in this case a decomposition for a distillation column, is captured by the DSPL++ architecture. The example provided here is explicitly initiated by the user, presumably during a system building phase and outside of the context of actually attempting to design an instance of a distillation column. At the end of this section we discuss the possibility of integrating the knowledge acquisition phase with design problem solving.

The steps in acquiring a decomposition in the DSPL++ architecture are summarized in Figure 42. The rest of this section shows how these steps are implemented in
1. Choose a device in the design database to decompose
   
   (a) If the desired device is not yet defined in the design database, then first acquire the device description and place it in the design database.

2. Describe the subtasks in the decomposition. This is the body of the decomposition. For each subtask described, do the following:

   (a) If the device identified in the subtask has not yet been defined in the design database, then acquire the device description and place it in the design database.

   (b) Describe how the requirements of the new subdevice are extracted from the requirements of the current (parent) device. This is done by creating Specification Maps that are associated with the subdevice.

   (c) Describe how the attributes of the completed subdevice integrates into the current (parent) device. This is done by creating Solution Maps that are associated with the subdevice.

3. Describe any preconditions on the current device’s requirements or the design context in general that limit the decomposition’s applicability

4. Collect any post-conditions for the decomposition to insure that a consistent design solution has been reached.

   Figure 42: Steps in Acquiring a Decomposition
Figure 43: Manually initiating the acquisition of a new method

The acquisition of knowledge in DSPL++ can be initiated manually in a number of different ways. Figure 43 illustrates one method for beginning the acquisition of a new DSPL++ structure, where the user has selected the top level command to “Acquire” a DSPL++ structure. This command, as can be seen from the figure, is one of a number of commands available to the user from the top level DSPL++ interface. A complete description of these commands, as well as the rest of the DSPL++ interface, can be found in Appendix A.

Once the user has requested the system to acquire a new structure, the system responds by presenting the user with a list of all of the architectural types currently known to the system that can be added by the user. The user selects from this list to initiate the creation of one of these elements. The list of options presented to the user, shown in Figure 44, includes most of the structures discussed in Chapter III. The list includes new goal definitions, the definition of a new device, and all of method types currently known to the system. The example of this section focuses on
Figure 44: Selecting the type of entity to acquire

the details of entering new method and device descriptions. Section 5.2 describes the addition of new task descriptions, while Section 5.3 describes adding a new method type.

As an example, suppose the user wishes to add a new design task decomposition to the DSPL++ knowledge base. By selecting the “Decomposition” item from the...
menu listed in Figure 45, the user initiates the acquisition of a decomposition. The user is prompted with a list of all the devices currently known in the design database, and selects one to decompose. The system then provides the user with a blank decomposition template for that device. This template guides the acquisition of the decomposition from the user. An example of a blank template for a distillation column decomposition is shown in Figure 45.

At this point, the user may now begin to describe the details of the decomposition to the DSPL++ system. Several menus are available on the decomposition template to direct the user in describing these details. The most important part of the decomposition is the description of the subparts into which the device is broken. Other information, such as pre- or post-conditions, may be specified by either selecting from constraints that already have been described to the DSPL++ system or creating new constraints.

The entry of the subparts of a decomposition is initiated by buttoning the template. The subparts of the decomposition are selected by the user either from the list of components that are already defined in the design database, or by entering the name of a new component. When a new component is entered, the system records the fact that the component definition must also eventually be entered into the design database.

The template describes the state of the ongoing decomposition acquisition. Figure 46 shows the state of the decomposition just after two components, the Condenser and Reboiler, have been selected as subcomponents of the Distillation Column. Addition-
ally a precondition to this method, the HydroCarbonFeed constraint. The method’s domain, the STILL domain, is listed at the top of the template. The figure shows the options available to the user for proceeding with the specification of the decomposition. At this point, as indicated in the figure, the user has three options:

1. enter a specification map that specifies how one or more of the distillation column requirements are mapped into the condenser requirements,

2. initiate specification of the condenser description, or

3. delete the condenser object from the decomposition.

The description of the decomposition need not follow any particular sequence. Typically, the specification of a device, its attributes, and how those attributes are derived all proceed in parallel. The user may choose to describe several attributes of a component, then describe how those attribute values are derived before completing the
description of the rest of the component attributes. The DSPL++ interface supports unstructured form of knowledge acquisition, and keeps track of the state of each of the entities as they are described to the system by the user.

Figure 47 shows an intermediate state similar to the situation described above. The decomposition template, shown in the upper left corner of the figure, is set aside for the moment while the user attends to the description of the condenser component. After partially describing the condenser, shown in the lower left of the figure, the user has opted to specify how one of the condenser specifications, the FeedFlowRate, is derived from the distillation column specification. The specification map template,
Simulation results for CondenserFlow SpecificationMapper:

Input required:
(\text{DistillationColumn} . \text{FeedFlowRate})

Output set:
(\text{Condenser} . \text{FeedFlowRate})

Used before set:
None

Unreferenced:
None

Figure 48: The simulation of a specification map

shown to the right in the figure, allows the user to select from the currently defined attributes of the device being decomposed, in this case the distillation column attributes, in determining how the subpart specification is computed.

At any point during the knowledge acquisition process, the user may simulate the input/output behavior of the various partially completed methods using the simulation techniques described in Section 5.1.1 above. The output of the simulation of the condenser's feed flow specification map, for example, is shown in Figure 48. The simulation shows that the derivation of the condenser flow depends on the column's feed flow rate, without showing the details of how that calculation is made. This simplified summary of the specification map allows the user to methodically track which aspects of the growing knowledge base are completed and which still need additional work as knowledge acquisition progresses.
Simulation results for SimpleDistillationColumn Decomposition:
Input required:
(DistillationColumn . Components)
(DistillationColumn . FeedFlowRate)
(DistillationColumn . FoamingTendency)
(DistillationColumn . NumberOfFeeds)
(DistillationColumn . NumberOfSideStreams)
(DistillationColumn . VacuumOperation?)
(SimulatedDistillationColumn . FeedPlateNumber)
(SimulatedDistillationColumn . NumberOfTrays)
(SimulatedDistillationColumn . PlateData)

Output set:
(Condenser . FeedFlowRate)
(Condenser . MaxThroughput)
(DistillationColumn . Cost)
(DistillationColumn . MaxThroughput)
(DistillationColumn . NumberOfTrays)
(EnrichingSection . FoamingTendency)
etc...

Used before set:
None

Unreferenced:
(DistillationColumn . Condenser)
(DistillationColumn . CostSimulation)
etc...

Figure 49: The symbolic simulation of the final decomposition
An abbreviated listing of the output of the simulation of the entire decomposition is listed in Figure 49. The simulation lists all attributes set by each of the components of the decomposition, as well as any input required, and any attributes of the distillation column that are not addressed by this particular decomposition. The list of attributes under the heading Unreferenced: represent all attributes of the distillation column device that have been defined in the design database, but for which no value has been computed by the decomposition. This output provides the knowledge engineer with a simple means of identifying potential gaps in a method's specification that need to be elaborated before a design problem solving is attempted. This allows a more rapid verification of the knowledge base since this information is available through static analysis of the method, rather than by resorting to inspection of the result of a test design run.

5.2 User Defined Goal Types

The rest of this chapter describes how the task and method descriptions found in DSPL++ can be extended, at several levels of detail. This section describes how additional goals can be added to the DSPL++ architecture, while subsequent sections deal with extending the method types found in DSPL++.

5.2.1 Goals in DSPL++

The task analysis of Chapter II provides a framework for organizing the design task, describing different types of knowledge used during design, and describing the roles these different forms of knowledge serve during design problem solving. Chapter
IV described how a particular set of goals actually implement the design task. This section describes the requirements for extending the task coverage of DSPL++ beyond the analysis provided in Chapter II, allowing the user to describe new tasks within the framework set out in Chapter III.

A goal description in DSPL++ is characterized primarily by

1. the goal’s requirements for activation, that is, the goal’s input or expected context,

2. the cross-section of methods in the knowledge base that are relevant to the goal and are activated by it,

3. the goal’s criteria for selecting a method, and

4. the goal’s criteria for completion.

All of these items are directly controlled in DSPL++ by the goal’s behavior description. Thus the principal focus in adding a new goal description is describing the above activities as part of the goal’s behavior description. Additionally, the new goal description must be integrated into the existing DSPL++ design strategy, typically by referencing the new task within an existing method or design strategy at some point in the design process.

As mentioned in Chapter III, a goal’s behavior definition describes the runtime behavior of a goal. The behavior is described as a list of actions taken by the interpreter when the goal is made the active goal on the goal stack. Typical actions that can be specified in the goal’s behavior are retrieving a set of relevant method
from the knowledge base or applying a method to the current context. The list of actions defining the goal’s behavior are placed on a command stream when the goal is instantiated, and are processed sequentially by the interpreter until either a subgoal is generated, or the goal terminates.

The set of actions allowed by the DSPL++ architecture are specified by statements written in a task-specific language, the Generic Goal Behavior Language (GGBL). This language is not generally visible to the user who is building a knowledge base in DSPL++ by instantiating existing method types, the activity described in the first level of extensibility listed in Figure 39. The user working at this level deals only with the pre-defined design primitives described by the existing method types found in DSPL++. GGBL is used to create new method and task descriptions to extend the design vocabulary of DSPL++. These activities are associated with the second level of extensibility described in Figure 39.

GGBL is not a design language. That is, most of the primitives of the GGBL language do not directly address the design task. Rather, GGBL’s primitives focus on manipulating the features of the basic DSPL++ architecture: managing and maintaining a goal stack, accessing the design database, retrieving elements from the knowledge base, etc. GGBL is also not intended as a general-purpose programming language, although certain features of the language allow considerable flexibility and generality. For example, in controlling the flow of a method or goal’s behavior execution, issues arise similar to those that arise in the design of any procedural programming language. Thus the constructs for controlling program flow in GGBL
resemble those of languages targeted at more general applications.

The actions of GGBL fall into several categories that describe how they function within the architecture. These categories are:

- creating and maintaining the goal stack,
- accessing and updating context information from the goal stack,
- controlling the flow of execution within the behavior definition,
- retrieving and selecting methods from the knowledge base,
- managing the design database, and
- user interactions.

A complete description of each of the commands in the GGBL can be found in Appendix B.

Many of the functions found at the level of interaction with the DSPL++ architecture described by GGBL overlap those found in the first level of interaction embodied by the existing method and task descriptions in DSPL++. The primary difference between these levels is the degree of flexibility available. For example, both levels of interaction support interactions with the design database. However, within existing methods, the interactions with the design database are highly proscribed and are only allowed in certain forms and contexts.

At the GGBL level, however, the database interactions are much less constrained. The user is free to develop new methods that treat the objects, attributes and values found in the design database in any fashion necessary for the new knowledge type.
5.2.2 Defining New Goals

The creation of a new design task requires the creation of a goal class and its corresponding behavior definition in GGBL. This is done by creating an appropriate LOOPS class and inserting the behavior definition as a LOOPS class variable in the class definition. The existing system has no automatic support for these activities, other than the language and architectural features described above.

The primitives of GGBL are implemented as LOOPS methods in the LOOPS class $Goal$, thus any new goal class must be a subclass of this LOOPS class in order to properly inherit the GGBL and DSPL++ goal class functionality.

The details of this process are further described in Appendix B.

5.3 User Defined Method Types

The creation of a new method type in DSPL++ is somewhat more complicated than the addition of a new task definition, however many of the concepts used in describing task definitions in DSPL++, such as the behavior definition for new tasks, carry over into the definition of methods and only need to slightly extended.

5.3.1 The Structure of a Method

This section describes the structure of a method definition in DSPL++. The user must fill in a method definition for a new method type before instances of that method type may be entered into the DSPL++ knowledge base.

Figure 50 shows an example of a blank method definition used to create a new method type in DSPL++. Each of the question marks in the method definition are
replaced with the appropriate information for a new method type. The function of each of the slots in the method definition template are described below.

The NAME slot defines the name of the new method type to be defined.

The KNOWLEDGE-PARTITION defines which partition in the knowledge base the method definition is associated with. Generally, method definitions are included as part of the base DSPL++ system, and so the value of this slot defaults to the value "System."

The SOURCE-LABEL slot indicates a list of tokens that are used to identify methods of this type in the source code for defining a new method. Most method definitions will define only a single source code label, or perhaps two if the method type name is lengthly. The second label in this case may be an abbreviated version of the full label. These tokens in this list are used by the DSPL++ parser to determine the type of method being parsed. Typically, the same token is used as the method type's name, but this is also not a requirement. In the method definition for
the PrimitivePlan method type, for example, TASK is included in the list of source labels in order to maintain some compatibility with the original DSPL source code conventions.

The META-METHOD-TYPES is a list of tokens that describe the role that method of this type will play in the design process. Most method definitions define only a single role that the methods will play during problem solving. Each of the tokens in this list are used to index instances of methods of the defined type when they are inserted into the knowledge base. In order to be properly retrieved at the appropriate point during problem solving, a corresponding design task must also refer to the tokens listed on this list during its method retrieval phase. More is described about defining new tasks in DSPL++ in Section 5.2.

The INSTANCE-OF-CLASS slot lists the name of an existing LOOPS class. New methods of the defined type will be instances of this LOOPS class. The class name listed as the value of this slot is restricted to be either the DSPL++ system-defined FoundationMethod class, or a subclass of this class. The FoundationMethod class contains all of the default behaviors contained in the DSPL++ system. Any subclass of this class will inherit all of the behaviors defined in this class as well as any additional behaviors defined in the subclass. Note that all of the method-specific extensions to the generic goal behavior language (GGBL) that define the generic method behavior language (GMBL) are found in the FoundationMethod LOOPS class.

The SOURCE-TEMPLATE slot defines the blank source code template that is presented to the user when a new instance of the method type is added to the system.
The TYPE slot defines what syntax is allowed by the DSPL++ compiler. Currently, one of four values are accepted in this slot, corresponding to the syntactic forms accepted by the compiler. They are

**CONSTRAINT-BODY** This syntactic form consists of a KNOWN section where values are retrieved from the design database, followed by a single Boolean expression.

**DECOMP-BODY** This is the syntactic form used in a decomposition. It consists of a device name, followed by a right-pointing arrow, followed by a list of device names.

**NOBODY** This syntactic form does not have a body. This is used for method types that are completely defined by their slot values, such as cases, plan redesigner, or plan selection specialists.

**PLAN-BODY** This form resembles the form found in the DSPL++ plan.

**SCRIPT-BODY** This form accepts a body written in the GMBL, such as the DesignMethod type.

**SPONSOR-BODY** This form does not have a KNOWN section, but has a body consisting of variable/expression pairs like the selector and sponsor knowledge forms in DSPL++.

**STEP-BODY** This form has a KNOWN and a DECISION section that consist of variable/expression pairs.
Other type descriptions could be added by making appropriate extensions to the DSPL++ compiler. Extensions to the compiler itself are discussed in Appendix C.

The BEHAVIOR slot defines the actions taken when a method of this type is executed. The value of the slot is a program written in the generic method behavior language (GMBL) introduced in Chapter III, and described in detail in Appendix B.

5.3.2 An Example Method Definition

Figure 51 shows the method definition for the Plan method type in DSPL++. This is exactly the method type described in Chapter III as one form of knowledge for proposing candidate designs. In the following description, it is important to note that the code listed in Figure 51 does not represent a method that is entered into the DSPL++ knowledge base, but rather represents a definition of a method type.

The NAME of the method type is Plan, that is, the code of this figure defines the Plan method. This particular form is entered automatically into the DSPL++ system when the system is loaded, since the plan method type is one of the pre-defined method types that comes with the DSPL++ system. Until this definition is loaded no methods of type "plan" may be entered into the knowledge base. The fact that this is part of the DSPL++ system architecture is indicated in the KNOWLEDGE-PARTITION slot, which has the token “System” listed as the name of the partition that this source code belongs to.

The SOURCE-LABEL slot listed in Figure 51 indicates that the source code for a plan begins with the “PLAN” token; this is also reflected in the sample source code template listed in the SOURCE-TEMPLATE slot. As described in Chapter III, a plan
(METHOD-DEFINITION
 (NAME Plan)
 (KNOWLEDGE-PARTITION System)
 (SOURCE-LABEL PLAN)
 (META-METHOD-TYPES Proposer)
 (INSTANCE-OF-CLASS Plan)
 (SOURCE-TEMPLATE
  (PLAN
   (NAME PlanName)
   (KNOWLEDGE-PARTITION SystemName)
   (ARTIFACT DeviceName)
   (SPONSOR DefaultProposer)
   (COMMENT What is this plan trying to do here?)
   (INITIAL-CONSTRAINTS None)
   (FINAL-CONSTRAINTS None)
   (REDESIGNER None)
   (FAILURE-MESSAGE "What couldn't be done?"
   (FAILURE-SUGGESTIONS None)
   (BODY
    (INITIALIZE DeviceName)
    (DESIGN Widget)
    (TEST-CONSTRAINT SomeConstraintName)
    (PARALLEL Task Task)
    (EXECUTE-TASK SomeTaskName)
    (EXECUTE-STEP SomeStepName)
    (EXECUTE AnyLispExpression)
    (REPORT-ON DeviceName))))
 (TYPE PLAN-BODY)
 (BEHAVIOR
  (FAILURE-METHODS FailWithSuggestions)
  (EVALUATE InitialConstraints)
  (FAILURE-METHODS PlanRedesign FailWithSuggestions)
  (PUSH-ONTO-AGENDA Body)
  (EVALUATE FinalConstraints)))

Figure 51: The definition of the Plan method type
is a means for proposing candidate designs. The fact that this method type is used in
the design proposal role is indicated in the META-METHOD-TYPES slot, which has
a value of Proposer. Instances of this method type are instances of the LOOPS class
Plan. This class describes the extensions to the generic method behavior language
(GMBL) for the plan body. Any instance of this class will inherit these extensions to
GMBL, and thus, since all plan methods are instances of this class, they will all have
the ability to execute their plan body.

The BEHAVIOR slot of Figure 51 describes the generic actions taken by the
DSPL++ interpreter whenever a plan is selected for execution by the system. The
slot lists a sequence of actions written in GMBL, as described in Chapter III. The
generic behavior of a plan in DSPL++, as described in the figure, are to execute the
initial constraints listed in the plan, using the FailWithSuggestions failure method
in the event that any of the constraints found there fail. Then the plan body is
pushed onto the agenda of the currently active goal. This time, in the event of any
problems arising, either the PlanRedesign or the FailWithSuggestions failure method
is available. Finally, any final constraints listed in the plan are executed.

It is important to recall that the code listed in Figure 51 does not define a method
in DSPL++ that is placed in the DSPL++ knowledge base, but rather describes a
method type, that is, one form that methods entered into the knowledge base may
take.
5.3.3 The Method Behavior Definition

As described in Chapter III, *method behavior* defines the runtime behavior of a method in DSPL++. As described in the previous section, this behavior is part of the method type definition. This procedural specification is used by the interpreter to determine what actions to take when an instance of that method type is run by the interpreter.

The syntax and semantics of the method definition is essentially identical to that described for the goal behavior definitions in GGBL. In fact, the method behavior language is a superset of the GGBL language. The extensions to GGBL that describe GMBL are found in Appendix B.

As with GGBL, there is currently little automatic support for the entry of method behavior definitions, beyond the architectural and language features described above. The behavior definition is described in the METHOD-DEFINITION of the last section is inserted in the method type definition, but no syntax checking is performed.

5.4 Adding New Behaviors

The next level of extensibility (Level 2 in Figure 39) involves the support of unanticipated extensions to the DSPL++ architecture, but that still fit within the general scheme of the architectural assumptions.

Extensions of this type take the form of extension to the GGBL and GMBL primitives. As mentioned previously, these primitives are implemented as LOOPS methods in the LOOPS class $Goal$, and its subclasses. Thus extending the GGBL and GMBL involves adding new LOOPS methods that implement the needed new
functionality.

No support is provided in the current version of DSPL++ for extensions of this sort. However, the design of such methods is considerably constrained by the DSPL++ architecture, and is thus correspondingly simplified.

5.5 Summary

Extensions to the DSPL++ architecture may be categorized into several levels, each of which is supported to a different degree depending on how well the needs of the user have been anticipated by the task analysis and underlying architecture.

The baseline level involves the creation of the knowledge base and design database necessary to build a working design expert system. This level is very tightly coupled with the design task description in DSPL++. Because of this close relationship allows an extensive user interface to be developed in support of the knowledge acquisition phase, as long as the forms of knowledge represented in DSPL++ and the task description matches the needs of the target design domain.

When the demands of the domain exceed the task description provided in DSPL++, the user may need to either add to the repertoire of method types or extend the design task description. These actions fall into a second category of extensions in which existing system behaviors are recombined into new method and task description. These activities are supported in DSPL++ by a pair of task-specific languages, GGBL for new goal descriptions and GMBL for method descriptions, that allow the specification of behaviors in terms of existing architectural primitives. This level of entry provides a considerably less sophisticated user interface, but still pro-
vides a considerable conceptual framework within which the user may specify new design primitives.

The last and most weakly supported level of system extensions allows the user to add new architectural primitives to the existing architectural framework. Although such additions have no supporting user interface, nonetheless such extensions should be relatively easy as long as they fit within the original design considerations of the underlying DSPL++ architecture.
CHAPTER VI

Other Design-Support Architectures

This chapter compares the DSPL++ architecture and its implementation to other strategies constructing design-support architectures. The purpose is not to provide an exhaustive survey of the design architectures, but rather to provide a few alternative strategies that serve to put the DSPL++ architecture in perspective.

Architectures for building knowledge-based systems fall into a range of categories depending on the generality of the architecture’s representation. In the most general case a knowledge-based system can be constructed using a completely general symbol-processing language, such as Lisp, with no initial commitment to any particular representation beyond the use of symbols. Any conceivable computation over an arbitrary set of symbols can be expressed in such a language. Such symbol-processing systems are as close to being computationally universal as can be realized in a physical computing machine.\(^1\) While systems built in this fashion have the advantage of having very few restrictions on the kinds of computations which can be performed, they have the disadvantage of forcing the system builder to explicitly construct any

\(^1\)To be truly computationally universal requires some form of infinite memory capacity. With this caveat we shall state, without proof, that many (if not actually all) of the architectures mentioned here are computationally universal devices. The more interesting question is, for a given architecture, which computations are easily achieved and which are problematic?
useful higher-level descriptions over which computations need to be performed.

Near one end of the spectrum are general-purpose architectures such as blackboard systems and rule-based systems. Such systems were not developed with the intent of solving design problems, but nonetheless can be applied to the design task and share several important characteristics with the DSPL++ architecture. At the other end of the spectrum are special-purpose architectures developed with a particular design application in mind. Such systems are usually developed with a particular target domain in mind. The systems described here will survey a few interesting examples from the design literature.

The location of a particular system or group of systems on our spectrum of generality is not an exact science, for a variety of reasons. An architecture tailored to a very narrow domain may yet choose to include escapes to general-purpose mecha-
nisms for portions of a domain or task that are poorly understood. Thus, although such a system may be classified as being very domain- or task-specific, there may yet be very general-purpose features in it.

The reverse situation may also arise. A system otherwise categorized as a rather general-purpose architecture may resort to highly compiled mechanisms in certain instances. In fact, this is the case with the DSPL++ architecture with regard to the failure handling mechanisms. In this instance, we have taken a very narrow view of handling impasses during design simply because the problem in general is so open-ended. Thus a system with a fairly broad scope of application in the design task, has certain features that are very narrow in scope.

6.1 General-Purpose Architectures

6.1.1 Blackboard Architectures

This section is divided into three parts. The first part describes the characteristics of blackboard architectures in general, and is loosely based on the original HEARSAY-II work ([Erman et al., 1980]). The second part deals with the BB1 architecture ([Hayes-Roth and Hewett, 1988]) and related work. The third part discusses the ACCORD work ([Johnson and Hayes-Roth, 1989]), a level of architecture built on top of the BB1 architecture. Each of the sections also comments on the relationship of the work described to the DSPL++ architecture.

The General Blackboard Architecture. One popular class of architectures for building knowledge-based systems are blackboard systems, deriving primarily from
the HEARSAY-II system of Erman [Erman et al., 1980]. The basic architecture of consists of three components:

- A blackboard, which is a global database of partial solutions, input data, and other data concerning the state of the current problem,

- A collection of knowledge sources, the containers of domain knowledge in a blackboard system, that access and record problem solutions in the blackboard during problem solving, and

- A separate control mechanism that opportunistically schedules the execution of knowledge sources based on the state of the problem described in the blackboard, and the appropriateness of the knowledge source.

A single knowledge source actively operates on the blackboard at a time, selected by the scheduler based on the blackboard state from a list of currently pending knowledge sources. This basic architecture is summarized in Figure 53, the components of which are described in more detail below.

The blackboard is a global collection of data which contents are completely available to both the scheduler and the currently active knowledge source. The blackboard servers to contain the initial problem specification, the final solution, and all working storage required during problem solving. The contents of the blackboard also records all control status, and is used by the scheduler to determine when knowledge sources should be "triggered." The blackboard can be organized in terms of whatever abstractions are suggested by the application.
Each knowledge source is an independent piece of knowledge about how to solve some aspect of a problem. Knowledge sources may communicate with each other through data posted on the blackboard. Each knowledge source describes a set of actions that may contribute to the problem solution, and the conditions under which those actions are relevant. The knowledge within a knowledge source about when to contribute to the problem-solving process is called its *triggering condition*.

No constraints are placed on the form that knowledge sources may take. Knowledge sources may be as simple as a rule in a rule-based system, or a plan or procedure, or even a neural net or an entire sub-expert system. Specialized representations as demanded by the application.

The control mechanism is separate from the domain-specific knowledge sources. Selection is determined based on the state of the blackboard. The model can shift
problem solving focus based on the blackboard state, and can use goal driven, data driven, model driven, or any other mode base knowledge source selection.

The control mechanism responds to events, which are typically changes to the state of the blackboard. These events include such actions as the creation, deletion, modification or access of data objects within the blackboard. The scheduler is notified of each such event that occurs. These events are used by the control mechanism to, for example, trigger the execution of knowledge sources, or rank order a set of previously triggered knowledge sources for execution.

6.1.2 Hayes-Roth and Hewett's BB1

BB1 is a domain-independent architecture based on the blackboard model. However, BB1 uses a separate blackboard for control decisions. The same control mechanisms used to solve domain problems are also applicable to solve the control problem. The domain blackboard is concerned with solutions to the domain problem, while the control blackboard determines dynamically what control decision to implement next. That is, BB1 determines dynamically which knowledge source to invoke at each cycle of the basic architecture.

This level of architecture is similar in function to the kernel architecture of DSPL++ described in Chapter III, and shares many of the same issues in its design.

- BB1 does not provide any guidance on how to approach any particular task, but it does provide the framework for clearly stating task-specific strategies.
Control issues in BB1, while separated from the domain problems, are still not entirely resolved. The control problem is still a potentially unbounded problem, since there is no limit on the number of knowledge sources that may be activated in determining a control action.

These issues are paralleled in the DSPL++ kernel architecture. The goal pursuing kernel does not provide any guidance on what sort of goals should be pursued. This issue is resolved in the task-level structures described in Chapter IV. Also, the kernel architecture has no mechanism for detecting or preventing runaway subgoaling.

6.1.3 Hayes-Roth's ACCORD

The ACCORD work [Johnson and Hayes-Roth, 1989] is an intermediate level built on top of the BB1 system [Hayes-Roth and Hewett, 1988, Hayes-Roth et al., 1988]. The BB1 system is blackboard architecture derived from the general blackboard architecture described above. To this basic architecture Hayes-Roth has added a domain-independent framework for the class of arrangement problems. The ACCORD level of constructs are layered on top of the BB1 architecture.

The ACCORD vocabulary allows the specification of solutions to arrangement problems using an incremental assembly method. ACCORD provides task-specific forms of knowledge which address this particular task:

- a skeletal concept network for domain-specific objects and constraints,
- a vocabulary of arrangement roles (e.g., partial arrangement, anchor, anchoree),
• a type hierarchy of assembly actions, events, and states (e.g., *anchor is a position action*), and

• linguistic templates for instantiating assembly action, event, and state sentences (e.g., *anchor anchoree to anchor in partial arrangement with constraints*).

This level of architecture parallels the task-level structures for design found in Chapter IV. What the ACCORD primitives provide in a domain-independent manner for the arrangement task, the DSPL++ methods and goals provide for the design task.

### 6.1.4 Laird, et al’s SOAR

The SOAR architecture is a goal-processing system implemented on the OPS-5 rule-based architecture. In SOAR [Laird et al., 1987], a goal is an object that consists of a symbol, called its identifier, and a set of augmentations. The augmentations are optional and may contain, for example, information such as a description of the desired goal state.

SOAR’s production working memory contains the complete processing state for problem solving. The working memory consists of a context stack, various objects and their augmentations linked to the context stack, and preferences. The context stack is a hierarchy of active contexts. Each context refers to a single goal and contains slots for the goal’s current problem space, state, and operator. The context stack doubles as SOAR’s goal stack.

Briefly, problem solving is implemented by the replacement of an object in a slot
by another object from working memory. For example, an operator within a context is selected by replacing the current operator (which may be empty, or "undecided") with a new operator. The replacement of a context object is driven by a two part decision cycle.

The first part of the decision cycle is the elaboration phase, during which all knowledge in the production memory relevant anywhere in the context stack is activated and applied. During this phase, new objects, new augmentations of old objects, and preferences are added to working memory. This phase continues until quiescence, ie, until all relevant knowledge has been applied and no further knowledge is relevant. Relevance is determined by production activation.

The elaboration phase is encoded in SOAR as productions, similar to OPS-5 productions. The only conflict resolution principle in SOAR is refractory inhibition, ie, an instantiation of a production is fired at most once during the elaboration phase in which it is activated. Rather than attempting to control problem solving at the level of productions by conflict resolution, control is exerted by an explicit decision procedure.

The second part of the decision cycle is the decision procedure. During this phase preferences generated during the elaboration phase are analyzed by a preference calculus to determine which results will be applied to the context stack. The decision procedure determines which slot in the context stack should have its content replaced, and replaces an existing object in some slot, ie, in one of the roles of a context in the context stack. The decision procedure computes the best choice for a slot based on
the preferences in working memory and the semantics of the preference concepts. If no final choice can be made, then the decision procedure fails and an impasse occurs.

There are four types of impasses that generate subgoals in SOAR. They are:

1. Tie. This impasse arises when there are multiple equally likely, non-conflicting choices for proceeding with problem solving. These are competitors for which insufficient knowledge (expressed as preferences) exists to discriminate among them. (In SOAR, a problem solving choice is implemented as filling a slot in the context stack or related object.)

2. Conflict. This impasse arises when there are multiple conflicting choices for proceeding with problem solving.

3. No-change. This occurs when there is no change in the problem solving state.

4. Rejection. This happens when the current extension to problem solving is rejected and there are no alternative choices.

Typically, the decision cycle affects only the processing of the goal at the top of the context stack, but this is not always the case. All goals are always "active" and may be affected by any decision cycle.

The expert system builder using SOAR is presented with a framework for expressing a problem as a state space search in a problem space. He has at his disposal a vocabulary of preferences for expressing domain heuristics and a built-in calculus to process those heuristics. The implementation language for expressing those heuristics
are production rules, i.e., situation-action pairs. Goal creation, goal selection, goal
termination are all handled automatically by the SOAR architecture.

The SOAR architecture compares to the kernel architecture of DSPL++, with
many of the same considerations and design issues.

- SOAR does not provide any guidance on how to approach any particular task.
- SOAR does require that all problem solving (i.e., search) be described in terms
  of state space search in a problem space.
- Control issues are partially addressed by SOAR's automatic subgoaling feature
  for resolving impasse situations.
- SOAR provides a number of weak methods which implement heuristic search
  knowledge. These methods are available when domain knowledge is unavailable
  or fails to reach a problem solution.

While the DSPL++ kernel architecture roughly corresponds in function to the SOAR
architecture, there are a number of important distinctions. The most significant
distinction is SOAR's explicit formulation of all problem solving methods in terms of
problem spaces. While it may in fact be true that all methods can be cast as problem
spaces, we feel it is an unnecessary restriction to force this view. The space of possibly
useful techniques for problem solving is large, and in attempting to create a useful
technology for system building we wish only to make the commitments needed to
allow for sufficient flexibility in method representation.
6.2 Task-Specific Architectures

6.2.1 Steinberg's VEXED.

The VEXED design system reported by Steinberg ([Steinberg, 1987]) uses a top down refinement model of design, plus constraint propagation, with the user making control decisions that are subsequently carried out by the system. The model ideally attempts to first decompose the structure into a few main pieces and completely defined interfaces between those pieces, so that the design of each piece becomes a totally independent sub-problem. Each of the finished pieces are then simply plugged together to solve the original problem.

The VEXED system (Vlsi EXpert EDitor) operates in the domain of digital circuit design, and has been more recently applied to the mechanical design domain. Both use the same design model, but with different knowledge bases.

The refinement rules describe legal, correct implementations, but not necessarily optimal or even preferred implementations. The refinements are structural decompositions, breaking a module to be designed into pieces.

Control decisions are left to the user, while all other processing is handled by the system. The user chooses which pieces to refine next, and also chooses which way to refine it. The system keeps track of which pieces need to be refined, as well as what alternative refinements are available. The system also performs any necessary constraint propagation. Constraint propagation in VEXED is done by the CRITTER system [Kelly, 1984].

The computational cost of constraint propagation does seem to the authors to be
a major issue. The authors believe that limiting the exhaustive propagation of every constraint everywhere it can as soon as it can perhaps could reduce this cost.

Control issues are left entirely up to the user. There are no internal representation of the goals and plans that go into a strategy for designing a circuit, and thus no support can be offered by VEXED in deciding which module to work on next or which refinement to make.

6.2.2 Tong’s DONTE

The DONTE system [Tong, 1987] is an attempt to study some of the issues of how a system based on top down refinement and constraint propagation might be extended to make the control decisions currently handled by the user.

6.2.3 Mitchell, et al’s LEAP

The LEAP system [Mitchell et al., 1985] is an augmentation of the VEXED system that attempts to infer which of the available refinements for design are locally plausible for a given design by observing the manual selection of refinements by the user during the design process.

The LEAP system provides advice on how to refine a design, while allowing the user to override that advice when necessary. In the cases where the user manually refines the design, LEAP records the action as an example of a missing rule, and generalizes from the example a new rule that summarizes the new design tactic. Since the user only intervenes in problem solving when the system is missing knowledge relevant to the task at hand, the training examples will focus only on missing knowledge.
LEAP (LEarning Apprentice Systems) is an attempt to directly assimilate new knowledge by observing and analyzing problem solving steps of a user during the normal use of a design system. The hope is to continually acquire new knowledge without an explicit "training mode".

The knowledge base of LEAP is partitioned into knowledge that characterizes correct implementations, and knowledge that controls the selection of preferred implementations. The authors point out that elements of the first type of knowledge are independent of each other, and thus no conflicts need be resolved when new knowledge is acquired in this partition. However, knowledge in the control partition may interact, thus adding new control rules may necessitate the adjustment of existing rules as well. To date, however, the authors have only considered the learning of knowledge of the first type.

6.2.4 Punch's TIPS

TIPS performs method selection the same way that DSPL++ does. His use of the term method and the whole sponsor/selector mechanism have the same roots, but for different domains. [Punch, 1989, Punch et al., 1986, Punch et al., 1989, Punch et al., 1990] The issues of task integration addressed by Punch in the TIPS system are similar to those found in DSPL++.

The TIPS system is very similar in its intention to the DSPL++ task-level architecture, but for the diagnostic task rather than the design task. TIPS provides for the runtime selection of problem solving strategies, but still requires that strategies be identified statically, prior to execution. This strategy doesn't allow for existing
problem solvers to automatically take advantage of new strategies as they are entered into the knowledge base.

6.2.5 Brown’s DSPL

In [Brown, 1984] Brown describes the DSPL architecture, from which the DSPL++ architecture described in this work springs. The attempt in that work was to address a subset of design problem identified as Class 3 design, which has since come to be known as routine design.

In routine design it is assumed that all of the devices and all of their parameters are known in advance. No mention is made of the functionality of the device, nor is there any consideration made for representing functions within the DSPL language. In their example of a system that designs air cylinders, the responsibilities of the specialists correspond closely to the structural decomposition of the air cylinder, but no feature of the language enforces or depends on this relationship. In fact, DSPL has no real model for an abstract device at all, other than to say that devices in the design database have attributes. In their model, requirements are not associated with a device, rather, by programmer convention, the input to a design system are a set of parameters labeled as requirements, and also by programmer convention, used and verified as input by the system.

The principal mechanism for affecting the flow of control in DSPL++ is method selection. In DSPL, specialists select plans to affect the flow of control. In DSPL++, the sponsor-selector mechanism is integrated into the basic architecture and not associated with a single agent type. As the basic mechanism for method selection for
goals, its application is broader than just the selection of plans within a specialist, although the roles in each system are similar.

The goals of this architecture are similar to the design messages passed among the agents of the DSPL architecture in that both define a task-level vocabulary for the design process. However the DSPL vocabulary was limited to a fixed set of primitives such as design, rough design, and redesign. This architecture has an extensible vocabulary consisting of a greater variety of primitives.

The fundamental difference between these design goals and the design messages of the existing DSPL architecture lies not so much in their content as in the way they are used. In DSPL, design messages are passed between agents via pre-defined links in the design knowledge base. The DSPL messages are information carriers, but they play no direct role in the flow of control during problem solving. In the DSPL++ architecture different design goals trigger different problem solving modes. Most goals are managed by the top level interpreter and broadcast to the entire design knowledge base, much like a blackboard system. Also, in DSPL the results of design are passed between agents via failure and success messages. The analogous communication in the DSPL++ architecture occurs through changes in the status of goals on the goal stack.
CHAPTER VII
Contributions and Future Research

This chapter reviews our investigation of the design task and the DSPL++ problem solving architecture and outlines potential directions of the research. These ideas are divided among five different topics in the research: design problem solving, features of the kernel architecture, the user interface to the system, system building aids, and the implementation of the DSPL++ architecture itself.

7.1 Design Problem Solving

This research has taken as its starting point the DSPL language for design problem solving [Brown, 1984], and the outline of the design task structure proposed by Chandrasekaran [Chandrasekaran, 1990a, Chandrasekaran, 1989]. This work carries the concept of routine design problem solving beyond that represented by Brown's original version of the DSPL language. There are several areas in which DSPL++ has made concrete contributions.

Multiple Methods and a Rich Task Description. In the terms of the research presented here, the DSPL language provided only a single method type for proposing design candidates, the design plan. In DSPL++, the forms of knowledge for proposing
designs have been broadened to include task decompositions and design cases, as well as providing for new method types to be added by the DSPL++ user. We have also begun to address how these multiple types of design knowledge are integrated into a statement of the design problem solving task.

In addition to expanding the knowledge types for design, DSPL++ also represents an expansion of the structure of the design task itself. The original DSPL architecture admitted four phases of the design task: a requirements phase, rough design and design phases, and a redesign phase. The requirements phase was not explicitly supported by the DSPL language. The rough design and design phases, while different in content, were identical in form. Verification and critique were not addressed.

In the DSPL++ system, the rough design and design phases have been subsumed into an expanded proposal phase of design, as described above. The DSPL++ system also allows for the verification of a completed candidate design. The specification of requirements for design is more formally supported both during the acquisition of design knowledge where the form of requirements are collected, and during the design process itself where the values of the design requirements are collected by the system. The design process in DSPL++ has also been expanded to include post-design critique and modification of a candidate design that fails to meet the design specification.

**The PVCM Design Strategy.** The PVCM method for design serves as one example of a strategy for achieving a satisfactory design within the above-mentioned task description. The implementation of this method also serves as an initial programming example in the DSPL++ architecture that may be used as guide to the exploration
and development of other methods for design as well.

Domain Examples. Two significant design examples are currently implemented in the DSPL++ architecture. The AIR-CYL system, derived from the earlier work of Brown [Brown, 1984], provided the initial test of the basic architectural features. The STILL system for distillation column design has developed into the prototype for most of the features of DSPL++ beyond those found in the original work.

7.1.1 Future Work

Additional Method Types and Design Tasks. While the DSPL++ work has extended the design task description beyond that of the original DSPL work, the analysis of the design task is not yet complete. Part of the original objectives of this work is to provide an open-endedness that allows a knowledge engineer the (hopefully simplified) opportunity to explore additions to the design task. We intend to pursue this work along several lines, described below.

One of the central contributions of DSPL++ is the concretization of the ideas, both conceptually and through implementation in a working system, of the decomposition of the design task into the subtasks of Propose, Verify, Critique and Modify, as described by Chandrasekaran in [Chandrasekaran, 1990a, Chandrasekaran, 1989]. The DSPL++ architecture is a task-specific realization of the task structure proposed by Chandrasekaran.

Each of the subtasks of design can be addressed relatively independently of the others. The DSPL++ architecture provides a relatively rich description of the Propose
subtask, and a somewhat weaker description of the other subtasks. The needs of each of these subtasks must be more completely addressed in order to build useful design systems. The work of Goel [Goel, 1989], for example, provides a more thorough investigation of the critique task than that presented in DSPL++.

The critique and modify tasks in the existing DSPL++ are currently structured as little more than post-design versions of the redesign by suggestion concept found in the original DSPL architecture. It is not clear that this is an optimal or even recommended strategy for organizing these tasks. Considerable work needs to be done in this area.

The critique of a faulty design could also be expanded in many directions. For example, this task would be a prime target for some form of fault diagnosis. The application of the considerable work on diagnosis to the problem of detecting the nature of mismatches between design candidates and design specifications could extend the application of knowledge based design systems considerably.

An example of another area for exploration is the incorporation of design simulations into the design process. The expandibility of the DSPL++ architecture provides a convenient springboard for investigations of this nature. The complete STILL system for designing distillation columns demonstrates how various types of simulations can be integrated into problem solving, even though the actual representation of simulations in DSPL++ is rather weak. The rigorous simulation of state used in the preliminary design of distillation columns in the STILL expert system is typical of a more traditional numerical simulation. The representation of other types of sim-
ulations, such as spatial simulations, and their role in the design task has yet to be investigated.

The Trade Study Design Strategy. The exposition of this method for design presented in Chapter IV is far from complete. We have yet to investigate how competing studies are set up and what types of knowledge are necessary to achieve this. One limitation to our exploration of this area has been the lack of a suitable domain example from which to gather the necessary data.

7.2 The Kernel Architecture

The remaining sections in this chapter address the DSPL++ architecture in terms of a practical technology for building knowledge-based systems for design.

Combining Task-Specific and General-Purpose Architectures. Perhaps the most significant feature of the DSPL++ architecture is the way in which a design-specific architecture is built over a fairly general-purpose base. At the task level, the user benefits from the high-leverage, task-specific description of design that is used to focus the flexible general-purpose architecture on which it is implemented.

Goal and Method Extensibility. An important feature of DSPL++ is its open-ended architecture. Although a useful subset of the design task has been implemented in the DSPL++ system, it is not complete. Thus, the ability extend the task and method behaviors of the architecture is a critical aspect of the continued exploration of the design task. As described in Chapter V, DSPL++ includes high-level tools to
extend the architecture along a number of different dimensions, including the ability to add user-defined task definitions and method types.

7.2.1 Future Work

Explanation Facilities. In [Herman et al., 1986], we described an extension to the original DSPL language that allowed explanation of design decisions, based on the DSPL agents used to make design decisions, and the function of those agents as defined by the DSPL syntax and semantics. In [Tanner, 1989], Tanner describes similar task-oriented explanation facilities in the context of diagnostic tasks. Similar techniques could readily be applied to the DSPL++ architecture. Several features of the DSPL++ architecture combine to provide the potential for a simple yet very powerful explanatory capability. They are:

- The rich design task description,
- Multiple method types, each with its own role in the design task, and
- The Goal stack, which organizes the system’s "working memory."

The basic idea for explanation in [Herman et al., 1986] is similar to many of the attempts at explaining the behavior of knowledge-based system: simply annotate each decision with the knowledge used to make the decision and the context that the decision was made in, and refer to those annotations when an explanation is called for. The advantage to this scheme in a rich task environment is that the knowledge used to make a decision has considerable explanatory power even when taken out of the immediate problem solving context and examined in light of its overall role in the
task description. The advantage of multiple method types with individual roles in the
design task stands in contrast to architectures with uniform representation schemes,
where function of each atom of knowledge must be deliberately attached to the knowl-
edge. The function must be ascribed by the knowledge-base programmer, and is not
immediately available from the knowledge form, as in the task-based approach.

An additional bonus in the DSPL++ architecture is the organization of the prob-
lem context by the goal stack. Each goal in the goal stack corresponds to a distinct
task in the problem context. Thus, the goal stack partitions the problem context
according to the role that each piece of the memory plays during design. This serves
to also organize explanation of items in the working memory in terms of the task it
is associated with during design.

**Formalization of Design Requirements.** In the current system, design require-
ments are represented as a list of attributes of a device whose values must be available
before the design of the device may commence. Although this scheme is probably ex-
pressive enough to represent the requirements of most devices, it is still only a modest
improvement over the original DSPL scheme for representing requirements, and does
not capture the internal structure of requirements or their relationships to each other
or the specification of the device. Considerable work remains to be done in this area.

Once a sufficiently rich taxonomy of design requirements is available, a number of
interesting research avenues become available. Operations on the requirements could
allow the progressive loosening of the design requirements when a design cannot be
synthesized to completely satisfy the original problem statement.
Another possibility is to allow the system to examine the requirements to determine which requirements are "soft" and open to slight modification, and which are "hard" and unable to be changed. This could be used to assist setting up initial conditions for several different trade-off studies at the beginning of the design process, rather than waiting for redesign to attempt to patch up problems at the end of the design cycle.

Another interesting area of research involves exploring the relationship between a set of design requirements and design candidate that either implements or fails to implement them. In the current system, the client has very few cues as to how a candidate design will be modified by adjusting the input requirements. Thus, when presented with a set of design requirements that cannot be satisfied by the automated design, he has little clue as to modify the problem to meet with success. The design client simply doesn't know the right questions to ask. Knowing how to adapt requirements to make a simpler design problem only comes with engineering experience, which the design client presumably does not possess. Such problems cannot begin to be addressed unless and until we can represent design requirements with more structure than just attribute/value pairs.

7.3 The User Interface

Many of the improvements to the interface of DSPL suggested by Brown in [Brown, 1984] have been incorporated in DsPL++. A few of the more important ones are listed below.
Automatic Tracking of System Structure. Many of the details of the domain knowledge that were managed by the programmer in the DSPL system are now handled automatically by the system. Thus, problems dealing with missing methods or incomplete pieces of domain knowledge are greatly reduced.

The symbolic simulation discussed in Section 5.1.1, for example, provides direct assistance to the knowledge engineer in identifying missing or inconsistent methods in the method library. Using this feature, it is fairly easy for the knowledge engineer to add or delete pieces of knowledge when a method is found to be in error.

Graphical Display of System Structure. The interface between the user and the knowledge base is accomplished in DSPL++ through graphical displays (see Appendix A). The user may now graphically see and access the methods as they are being constructed in the knowledge base. This improves the efficiency of the knowledge acquisition phase considerably.

Tracing and Controlling the Design Process. The quality of the system’s interaction with the end-user during design has improved to the point that a system can now be constructed that is available to users other than the system’s original implementor. This is due primarily to the judicious use of the underlying LOOPS and Interlisp-D programming environments.

The user of a knowledge-based system written in DSPL++ can exert considerable control over the progress of the design process. The user may not just observe and control any primitive action of the DSPL++ system, but may choose exactly the area
and level of detail desired. This control is made possible by both the features of
the structure of the kernel architecture, such as the goal stack and knowledge base
of design methods, as well as the design expertise captured in the DSPL++ task
analysis.

7.3.1 Future Work

Graphical Output of Designs. One area where the DSPL++ system could be
improved is in the display of the design results. The existing system only provides
a list of the attributes and their values for each of the components in the candidate
design. The visibility of the design process would be improved considerably if, at
least for mechanical domains, the structure of the candidate design could be displayed
graphically as the design progressed.

7.4 System Building Aids

A number of features of the DSPL++ architecture aid the knowledge engineer in the
system building task.

The Method Editor. The syntax-checking compiler for methods detects many er-
rors that the knowledge engineer would otherwise have to track manually. By keeping
track of devices and their attributes, the structure of methods, and noting unresolved
references in new method definitions, the efficiency of traditional knowledge acquisi-
tion techniques are greatly improved.
**Task-driven Knowledge Acquisition.** The traditional strategy for creating knowledge-based problem solvers is to first enter each bit of knowledge, and then execute the system to determine what behavior is lacking or buggy. When problems with the knowledge are discovered, more bits of knowledge are entered and the cycle is repeated.

A task-specific architecture, such as the DSPL++ architecture, allows an improvement on this approach. The knowledge of the design task encoded in DSPL++ includes the forms of knowledge that are acceptable to the system as well as the roles that those forms can take. Using this information, the system can be made to prompt the knowledge engineer for each type of knowledge needed as it progresses through the design of a novel object. Each time the system fails to pursue a goal for lack of knowledge, the same retrieval keys used to query the knowledge base may be used to initiate the acquisition of additional knowledge from the knowledge engineer. In this fashion, a new application can be "debugged" into existence. This feature of the architecture also minimizes the amount of expertise the user must have about the way in which the system performs the design task, since he is led on an interactive session through all of the tasks performed.

**Symbolic Simulation of Methods.** This feature, described in Section 5.1.1, aids the knowledge engineer considerably in controlling inconsistencies and missing knowledge during the knowledge acquisition process.
7.4.1 Future Work

The Goal Behavior Language. The existing version of DSPL++ does no syntax checking on the behavior specifications entered by the user. Instead, somewhat like the original DSPL, problems with the behavior definitions are only “discovered” at run time. This poses no real practical problem since defining new behaviors should be a relatively infrequent task.

However, it would be a relatively simple matter to develop a grammar based on the structure of the behavior definition language presented in Chapter III, similar to those for the DSPL++ methods. This would easily eliminate a small but bothersome class of errors encountered when defining new method types and tasks. Additionally, the proposed parser could perform analysis functions similar to those already performed by the existing method compilers, with similar benefits. It would be relatively simple to support the ability to browse the structure of the task knowledge, similar to the way that method structures can be browsed in the current system. Currently, there is no easy way to examine the structure of the task knowledge.

An Integrated Acquisition/Maintenance Editor. Currently the simplified forms presented in Chapter V for knowledge acquisition are only available for creating new methods, and not for updating existing methods. These forms should be extended to include adding goal descriptions and method type definitions.

A More Accessible Port. The Xerox LOOPS and Interlisp-D programming environment that contributes so greatly to the current implementation of DSPL++ is
unfortunately its Achilles' heel. The DSPL++ architecture is currently implemented in the Koto-Beta Product release of LOOPS, that runs exclusively on the Xerox 1100 series hardware. This will severely limit the potential for propagating the ideas contained in DSPL++ in the research community. A later release of the Xerox environment is available that is compatible with a much wider range of hardware platforms. Upgrading the DSPL++ software to this later version of Lisp would help reduce this problem. A version of the DSPL++ software compatible with a more common software environment, perhaps Common Lisp or C, would make the software much more widely accessible, but unfortunately would involve a much larger development effort.

Improved Method Indexing. Currently, methods are retrieved from the knowledge base by mapping through a list of LOOPS instances, checking each one for the desired characteristics. While this may be satisfactory for systems containing a few hundreds of methods, a large system containing perhaps tens of thousands of methods would be bogged down. A more efficient indexing method would be essential.
Appendix A

The User Interface

This chapter describes how to load and interact with DSPL++ on a Xerox 1108 Lisp machine (a.k.a Dandelion), running the Beta Product LOOPS software on at least the Koto release of Interlisp-D.\(^1\)

It is assumed that the reader is familiar with both LOOPS and Interlisp-D on a Xerox Lisp machine, as well as the theoretical motivations underlying the DSPL++ language. Throughout this document the standard LOOPS functions are omitted, and only those functions relating to the DSPL++ operation are described. For a description of the standard browser functions found in LOOPS (as well as a complete complete description of the LOOPS software system), the reader is directed to the LOOPS primer ([Xerox, 1987a]) and the LOOPS reference manual ([Xerox, 1987b]). The prudent user will also have access to the Interlisp-D reference manuals ([Xerox, 1986b]), and be familiar with the material in the Interlisp-D primer ([Xerox, 1986a]).

\(^1\)This chapter discusses the interface to the DSPL++ architecture as it currently exists. As of this writing, a project is underway to port the Beta Product LOOPS version of DSPL++ into the released Product LOOPS on the Medley version of Xerox lisp. Variations between the two ports will inevitably occur, however the bulk of material in this chapter should remain unaffected.
A.1 Loading DSPL++

The DSPL++ system is loaded into an existing lisp environment. The Xerox Beta Product LOOPS software must already be loaded. Considering the size of the DSPL++ software and the potential size of the DSPL++ knowledge base, a fresh system sysout is recommended.

To load DSPL++, insert the diskette with the DSPL++ files on it and type into the console window:

LOAD(FLOPPYLOADME.DCOM)

When the files have been successfully loaded, an icon will appear on the screen that allows access to the DSPL++ system. Figure 54 shows the DSPL++ icon and its top level menu.

Figure 54: The DSPL++ icon
A.2 The DSPL++ Icon

The DSPL++ icon facilitates access to the top level DSPL++ functions through the use of the window and mouse facilities of LOOPS. The icon allows for the creation and inspection of the basic components of the DSPL++ architecture, the knowledge base, the design database, and design cases, as well as the initiation of knowledge acquisition and the execution of design problems. The commands available from the icon are described below.

Middle Button Commands.

**Acquire** Acquire design knowledge from the user.

**Browse** Brings up a specialized browser for one of the components of the DSPL++ system currently defined. This item has three subitems, one for each of the components that has a specialized browser.

**Case** Brings up a case browser for an existing design case. The user is prompted to select from a menu containing a list of the existing cases in the knowledge base.

**Knowledge Base** Brings up a browser on the existing knowledge base.

**Design Database** Brings up a browser on the design database.

**Create** Creates a new instance of one of the major components of the DSPL++ architecture. This item has three subitems:
Case  Creates an empty instance of a design case.

Knowledge Base  Creates a new knowledge base, with no methods in it.

Design Database  Creates a new design database, with no design objects in it.

Design  Causes the design process to begin. The user is prompted for an object to be designed from a menu of all objects currently defined in the design database. Design requirements can be entered interactively, or taken from an existing design case.

Load  This is the facility to load files into the DSPL++ knowledge base. There is a single subitem.

Source Code  Loads DSPL++ source into the currently defined knowledge base. The user is prompted for the name of an ASCII file, presumed to contain DSPL++ forms that are read and compiled into the current knowledge base. The progress of the load is noted in the console window. If any compilation errors occur during the load, the source is automatically saved by the system on its internal list of forms known to have syntax errors, and the load continues.

Copyright notice  Displays the OSU LAIR's copyright notice for the DSPL++ system and software.
A.2.1 Running DSPL++

The design of an object in DSPL++ is initiated by selecting the "Design" command from the DSPL++ icon’s menu. The user is presented with a list of the devices and objects known to the design database. By selecting one of these items, the user begins a design session that hopefully will result in the complete design of the selected device.

The system uses different strategies to collect the device input requirements, depending on the currently selected input mode. The form and necessary content of the requirements are determined by the device’s definition in the design database. The values for the requirements are determined by the input mode as follows:

- User interaction: In this mode the user is prompted for the input requirements as described in the device’s database definition.

- Previous Case: In this mode, all values for input are taken from an existing case. The user is prompted from a list of existing cases.

- Defaults: Values are selected from "typical" values as described in the device definition.

If the design of a device is desired that is not currently defined in the design database, an option is available to introduce a new definition into the design database at the start of design. The user is then prompted for a device description before the design process begins.
Depending on the initial execution modes set in the DSPL++ knowledge base, several additional browsers and tracing aids are displayed at the beginning of problem solving. A Goal Stack Browser (see Section A.6 below) displays the goals spawned by the system as the design proceeds. The Case Browser (Section A.5) displays the components instantiated during design. The interpreter’s behavior may be interrupted or modified at any time during design via the interpreter control menu (Section A.6.1).

A.3 The Knowledge Base Browser

The knowledge base browser displays a lattice representing the methods contained in the DSPL++ knowledge base. The knowledge base in the DSPL++ system manages all method source code. Through the knowledge base browser in DSPL++ the user can access, inspect, and modify methods contained in the knowledge base.

The knowledge base browser is implemented in LOOPS as a specialization of the LOOPS InstanceBrowser class. Thus, each of the nodes in the lattice displayed in the browser are instances of classes. As previously described in Chapter III, all methods in DSPL++ are implemented as instances of method Classes.

The user may either add methods through the knowledge base browser or through the knowledge acquisition features available in the DSPL++ icon and as triggered by the design process. When adding methods through the knowledge base, the system creates a source template for the desired method type, and invokes the Dedit Interlisp-D editor on the new template. The user edits the form and exits the editor. When Dedit is exited, the DSPL++ system parses the edited source and compiles a new method instance that is subsequently installed into the knowledge base. If errors
Figure 55: The DSPL++ knowledge base browser

are encountered by the system during the compilation process, the source may be re-edited, or optionally saved for later consideration.

Figure 55 shows a knowledge base browser as it appears in the LOOPS environment. Only a subset of all the methods in the knowledge base are being displayed in the figure. The title of the browser, shown in the heavy black bar at the top of the figure, shows the filter being used to determine which methods are currently displayed. In this figure only Proposer and Verifier methods from the STILL and DesignTask domains are being displayed. There are apparently six such methods in this particular knowledge base. In the lower right of the figure, next to the arrow-shaped cursor, is
the configuration menu used to create the filter described above. The two features described above are listed in the menu, as well as a menu command to add new features to the filter, and a menu command to exit the menu. When the menu is exited the current configuration of the filter is applied to the browser. To the lower left in the figure is a menu displaying all of the currently defined functional roles that methods take in the design process. This menu is used as one means of accessing and adding new methods to the knowledge base.

Another mode of method display is exemplified by the browser shown in Figure 56. This figure shows a subbrowser of a single method in the knowledge base, but this time with its complete internal structure. Each of the nodes in this lattice are discrete
entities in the knowledge base that may be independently inspected and modified.

The rest of this section describes the commands available to the user from the knowledge base browser for manipulating DSPL++ methods.

**Title Menu Commands.** The title menu commands are displayed in a pop-up menu when either the left or middle mouse button is pressed while the cursor is pointing to the title bar within the browser. The following commands are available in addition to the standard **LOOPS** commands:

- **SaveSource** Saves the DSPL++ source code for all methods currently defined to an ASCII file. A list of all knowledge base partitions are presented to the user to determine which methods are to be saved. The user may optionally select one or more partitions to save to a single file, save all partitions to a single file, or save each partition to its own individual file. The name of the file and the number of methods saved is displayed in the browser's prompt window.

- **Edit** Edits a method in the knowledge base. The user is prompted with a list of all method types in the system, and then a list of all methods of that type to determine which method to edit. There are a number of subitems for this menu item.

- **Edit** Same as above. Edits an item in the knowledge base.

- **Add** Adds a new method to the knowledge base. The user is prompted with a list of all currently defined method types.
Delete Removes and destroys a method from the knowledge base. The user is prompted with a list of all currently defined method types, and then with a list of all of the currently defined methods of the selected type in order to determine which method to delete. The user is prompted for confirmation before any action is taken.

Inspect Bring up an inspector window on a method. The determination of which method to inspect is the same as noted above.

Dump Sends a listing of methods to the printer.

Reparse Reparse some methods.

Edit unreferenced method Edit a method that is not referenced by any other method. The user is prompted from a list of such methods derived from the system symbol table of all methods defined in the knowledge base.

Add referenced method Adds a new method to the knowledge base. The user is prompted from a list of all the methods referenced by other methods in the knowledge base but not yet defined. This list is produced by examining the system symbol table of all methods currently defined in the knowledge base.

Edit bad source Edit a piece of DSPL++ source known to have syntax errors. The user is prompted from a list of methods maintained by the system whenever translation errors occur.

Function Edit Menu Produces a permanent menu of all of the MetaMethod-Types currently known to the system. This feature provides a short cut
to editing methods.

**Form Edit Menu** Produces a permanent menu of all of the method types currently known to the system. This feature provides a short cut to editing methods.

**Show all** Shows every method in the knowledge base. This item and its subitems control what subset of methods are displayed in the knowledge base browser.

**Show all** Same as above. Shows all methods in the knowledge base.

**Default show** Sets the default display mode for the knowledge base. The design proposers for all domains are selected for display.

**Set starting types** This command allows the user to determine the characteristics of the methods to be displayed in the browser. The user is presented with a menu in which he can create a set of retrieval keys that determine which methods are displayed. See figure 55 for an example.

**Set ending types** Allows the user to determine the amount of structure to be displayed in the methods selected for browsing. A menu identical to the menu described for setting the starting types is presented that allows the user to configure the method characteristics that determine the display cutoff point.

**DestroyKnowledgeBase** Destroys the knowledge base and all methods contained in it. The user is prompted for confirmation before any action is taken.
Set initial execution mode Allows the user to control the default behavior of the interpreter before execution begins. The user is presented with a list of control and display options similar to the menu used to control the interpreter's behavior during design. See section A.6.1 for more details on controlling the interpreter.

Show initial execution mode Displays the initial execution modes of the interpreter in the prompt window.

Middle Button Commands. The middle button commands are displayed in a pop-up menu when the middle mouse button is pressed while the cursor is pointing to a node, representing a method in the knowledge base, in the browser lattice. The command selected will act on the method that the cursor was pointing at when the mouse button was pressed. The following commands are available to operate on the individual methods displayed in the browser:

Delete Deletes the selected method from the knowledge base. The user is prompted for confirmation before any action is taken.

SubBrowser Spawns a new browser with the selected method as the root and displays the method's complete internal structure.

Symbolic Simulation Runs a simulation of the selected method's and displays a report on the method's input/output behavior. The output may be directed to either the default print window or a file.
A.4 The Design Database Browser

The Design Database Browser displays and allows access to the device definitions in the DSPL++ design database. The devices definitions in DSPL++ are implemented as LOOPS classes, so the browser is a variation of the LOOPS ClassBrowser object, and as such inherits most of its behavior from that Class. This section describes the additions to the basic ClassBrowser behaviors.

One important addition to the ClassBrowser is the Attribute manipulation features. Attributes of DSPL++ devices are implemented as LOOPS IVs, and could be manipulated using the standard LOOPS editing functions for IVs. However, the attribute editing functions included in the database browser also maintain several internal lists of attributes not known to the standard LOOPS functions. It is advised that only the DSPL++ functions are used.

Additionally, an assortment of IV facets are defined by the DSPL++ system that allow various characteristics of device attributes to be defined. The DSPL++ system recognizes several access functions for input of attribute values, the definition of units for attributes, default values, minimum and maximum values, attribute type-checking, all implemented as IV facets recognized by the system. These features are all accessible through the design database browser, and are described below.

Figure 57 shows a design database browser as it appears in the LOOPS environment. The database shown contains the device definitions for both the STILL and AIR-CYL knowledge based systems. Note that since devices are implemented in LOOPS as Classes, the names of the devices defined in the system must be a unique
Figure 57: A DSPL++ database browser
class name.

The rest of this section describes the commands available in the design database browser.

**Title Menu Commands.** The following commands are accessed through the title menu and operate on the entire DSPL++ database:

- **Add Design Object** Causes a new design object to be added to the design database. The system enters a knowledge acquisition mode allowing the new device to be defined, including attributes, attribute definitions, input requirements, input constraints, and display attributes of the device.

- **Save** Saves the database LOOPS classes, method and instances associated with the current database to disk. The file definition is built automatically before the save takes places.

- **Build File COMS** Causes the necessary file package commands to be executed to add all existing database objects in the current database to the file definition. This allows the user to inspect and modify the file definition before the file is saved, if necessary. This command is automatically executed by the Save command.

- **Inspect the browser** Brings up an Interlisp-D inspector window on the browser instance.
Destroy the database Allows the user to remove the current instance of the database object and all of its components from the LOOPS class structure. The user is prompted for confirmation before any action is taken.

Middle Button Commands. The following commands operate on the individual device definitions displayed in the database browser lattice. They operate only on the selected device definition.

Add Attribute Add a new attribute to the selected design object.

Add Attribute Same as above.

Edit Attribute Allows the user to edit an existing attribute definition attached to the selected design object. The user is prompted with a list of the currently defined attributes.

Delete Attribute Removes an attribute from the current design object.

Rename Attribute Allows attributes to be renamed.

Move Attribute Allows an attribute to be moved from the currently selected design object to the currently boxed node in the browser. A node is boxed using the standard LOOPS BoxNode command in the Left Button Commands.

Remove Design Object Removes an object from the database. The user is prompted for confirmation before any action occurs.
A.5 The Case Browser

The case browser displays the device instances created for a particular design run of the DSPL++ system. The device instances created during the design process are instances of the classes defined in the DSPL++ database. The case browser is implemented as a specialization of the LOOPS InstanceBrowser class, and inherits most of its behavior from that class. This section describes the additions to the InstanceBrowser behavior for creating, inspecting and maintaining DSPL++ cases.

Figure 58 shows an example of a case browser displaying the results of a run of the STILL knowledge-based system. Each of the nodes in the lattice are instances of the design objects listed in Figure 57. The result of the design run was the creation of a DistillationColumn instance, shown on the left of the figure as the root of the lattice, with all of its components and input requirements displayed to the right.

Title Menu Commands.

Inspect case Brings up an inspector window for the case instance.

Add instance Adds a new instance of an object from the design database to this case.

Left Button Commands.

Inspect Brings up an inspector window for the selected instance.

Display This brings up a slightly more “user friendly” inspector window for the selected instance. This window displays the instance’s attributes and current
Figure 58: The DSPL++ case browser
values, allows attribute values to be entered and updated, and allows any access
routines defined in the device definition to be run when editing requested.

**Middle Button Commands.**

**Destroy** Removes the selected instance from the case and destroys it. The user is
prompted for a confirmation before any action is taken.

**Initialize** Allows all of the selected instance’s attributes to be set en mass. There
are three options for initializing an instance’s attributes:

- **All** Clears out all values in the component.
- **Defaults** Sets values of all attributes to their default values, if any are defined
  in the device definition.
- **Prompt** Runs the access routines for all attributes in the device.

**A.5.1 Attribute Displays**

In order to facilitate access to the individual attributes and attribute values of the
device instances within a case, DSPL++ provides a menu for quickly accessing the
current values within a device instance. This display lists all of the device attributes
with their currently defined values.

The attribute display may be generated either manually by the user by selecting
the “Display” Left Button Command in the case browser, or under program control
during problem solving via the REPORT primitive. Under program control, this
display allows the user to easily track the progress of the design process.
This display is convenient for inputting design requirements prior to the design process. Rather than allowing the design system prompt for the necessary design requirements as they are used during design, the user may explicitly set the input values in any order.

Figure 59 shows an example of an attribute display from the STILL system. The figure lists all of the attributes and their corresponding values for a sieve tray. The left side of the display lists the attributes, and the right side lists the corresponding values. Buttoning on an attribute value allows a new value to be entered. The system uses any access functions found in the attribute definition to drive the input of the new value.
A.6 The Goal Stack Browser

The Goal Stack Browser displays the hierarchy of goal instances generated by the DSPL++ system while attempting to solve a design problem. The browser is continually updated throughout the design process to reflect the current state of the goal stack.

As described in Chapter III, goals in DSPL++ are implemented as instances of LOOPS classes that represent the various task in the design analysis. The Goal Stack Browser that displays these instances is implemented as a specialization of the LOOPS InstanceBrowser Class. In essence, this browser displays the SubGoals IV of a set of LOOPS instances, beginning with the goal instance that initiated the design process.

The user may control almost every aspect of the goal interpreter's behavior throughout the design process. The interpreter may be halted at any point, either at the explicit request of the user, or under program control. While halted, the user may inspect and modify the current state of the design and restart problem solving in the new state. At any point, the user may backup to a previous point in the design process, select a different course of action than that selected by the system, and restart design on the new course. The user may also completely abort a design at any point.

The user may also halt processing in order to enter knowledge about the domain that may assist in producing a finished design. In fact, the control that the user is capable of exercising over the design process makes it reasonable to build a knowledge based system by simply requesting the system to design the desired object, and
allowing the system to direct the user to input the necessary domain knowledge as it is needed during design.

A typical goal stack browser with its attached interpreter control menu is shown in Figure 60. The example is taken from the STILL knowledge based system. The top goal in the browser was created by a "Design" request for a Distillation Column. The system is in the middle of the PVCM method for design, and has found three methods for proposing a design candidate. The menu attached to the bottom left of the browser is the interpreter control menu, and is described below. This menu remains available to the user throughout the design process to control the interpreter's behavior. When the design is complete, the menu is removed. The menu overlapping the bottom right corner of the menu is a temporary pop-up menu generated by the system to allow the user to override the system's own selection process. This selection menu only appears during the method selection phase of design, and only when the user has set the interpreter's behavior to force manual selection of methods.

The rest of this section describes the commands available to the user beyond those available in the standard LOOPS InstanceBrowser Class.

**Title Menu Commands.**

**Crank** Starts the interpreter on the currently boxed goal instance. A goal instance is boxed using the standard LOOPS BoxNode command in the Left Button Commands for this browser.

**SetDisplayMode** Determines the level of detail of the goal stack display.
Design a Distillation Column

Initialize the Distillation Column object

Retrieve Select

PVCM Design Method

Propose Candidate Design

Retrieve Select

Case 3452
Klein
Simple Distillation Column

Please select one of the PERFECT items:
- Case 3452 Case is INDIFFERENT
- Klein Plan is PERFECT
- SimpleDistillationColumn Decomposition is PERFECT
- Use the FAIL Method
- Show all suitabilities
- Select another method
- Add new method
- Auto selection

Figure 60: An example of a DSPL++ goal stack browser
SetDisplaySize Determines the font size of the goal stack display.

DestroyCase Destroys the goal stack and the associated case, including all browsers.

Warning: The user is not prompted for confirmation on this action.

DestroyGoalStack Destroys the goal stack and its associated browser, but leaves the associated case intact.

Middle Button Commands.

SubBrowser Creates a goal browser of the goal stack beginning with the selected node. The initial display mode for the subbrowser is to display all goal instances.

ShowHistory Displays the selected goal’s execution history in the default print window.

ShowPrimitive Inspects or displays a primitive of the language from this context.

The user is prompted from a menu of all of the language primitives defined for this goal context.

ShowMethod Displays the initial and current state of the currently active method for this goal, if any. All output is directed at the default print window.

ListMethods Shows all methods listed in the SelectionSet for this goal.

ShowRetrievalKeys Shows the arguments to the index unpacker used to retrieve the methods listed in the SelectionSet. All output is directed to the default print window.
A.6.1 Controlling the Interpreter

At the beginning of each design session, a menu is created that allows the user to control the behavior of the interpreter. The commands available to the user from this menu are as follows:

**Pause** Temporarily halts processing. Processing is continued by the “Resume” command.

**Resume** Continues processing after it was halted by the “Pause” command.

**Single stepping** Determines whether or not the interpreter halts before processing each goal. The user must select “Resume” to continue processing.

- **Single step ON** Turns on single stepping.
- **Single step OFF** Turns off single stepping.

**Task step** Determines whether or not the interpreter halts before each “Design” goal, and not between other goals, such as the Retrieval or Selection goals. The user must select “Resume” to continue processing.

- **Task step ON** Turns on task single stepping.
- **Task step OFF** Turns off task single stepping.

**Retrieval Failure Trap** Determines whether or not the interpreter initiates knowledge acquisition when a *Retrieve* goal fails to retrieve anything from the knowledge base.
Retrieval Failure Trap ON  

Turns on retrieval failure trapping.

Retrieval Failure Trap OFF  

Turns off retrieval failure trapping.

Input Mode Determines how input is received during problem solving. There are three modes that may be selected by the three subitems.

Use old case data Input requirements are taken from a previous case. The user is prompted with a list of the currently loaded cases in the system.

Prompt as needed Input requirements are taken from the user as they are used during problem solving.

Use defaults Default values are used for inputs to the design problem, and no interaction is required from the user during design unless an input attribute has no default value defined.

User Control Level Determines the level of confidence the system requires before deferring to the user’s selection of a method. There are several levels of discrimination, corresponding to the possible suitability ratings determined by a method’s sponsor.

Perfect Allow user interaction even when Perfect methods are available.

Suitable Allow user interaction even when Suitable methods are available.

Indifferent Allow user interaction only when Indifferent methods are the best choice available.
None  Never go to the user for help. This places the system completely under its own control.

All Displays ON  Turns on all displays for tracing the design problem solving behavior. The subitems allow each tracing mode to be turned on individually.

All Displays ON  Same as above.

Goal Display ON  Brings up a goal browser so that the user can visually trace the progress of the goal stack as design progresses.

Case Display ON  Brings up a case browser on the current problem so that the user can see the device relationships developing during the design process.

Attribute Display ON  Brings up a user friendly inspector for each device as it is added to the current case, and allows the user to visually inspect the values of each attribute of each device as the design progresses.

Knowledge Base Trace ON  Flashes methods in the knowledge base browser as they are retrieved during problem solving.

Attribute Trace ON  Prints a list of each attribute accessed or stored during problem solving in the default printing window.

Goal Trace ON  Prints the name of each goal as it is entered or exited during problem solving. This mode is somewhat faster than the Goal Display mode, but not quite as pretty.
All Displays OFF Allows the user to turn off display modes. The subitems allow each mode to be turned off individually.

All Displays OFF Same as above.

Goal Display OFF Turns off the goal display.

Case Display OFF Turns off the case display.

Attribute Display OFF Turns off the Attribute display.

Knowledge Base Trace OFF Turns off knowledge base tracing.

Attribute Trace OFF Turns off attribute tracing.

Goal Trace OFF Turns off goal tracing.

Show modes Shows the current interpreter modes, as set by the commands in this menu, in the top level prompt window.

Abort Halts the design process and destroys the interpreter. The current design case and goal stack are left intact. Processing can be resumed at any goal in the goal stack through the "Crank" command in the title commands of the Goal Stack browser.

Abort Same as above.

Abort and Destroy Halts the design process and destroys both the current design case and the associated goal stack. Warning: The user is not prompted for a confirmation before the command is carried out.
Figure 61: A typical DSPL++ working environment
Figure 61 shows a LOOPS screen during a typical work session. At the upper left corner is the DSPL++ icon. Just below it is the Lisp console window, displaying the LAIR copyright notice. Below that is a goal stack browser, showing the state of a design problem from the STILL domain in progress. Below the goal stack browser is the interpreter control menu. Just below that is the LOOPS icon, that allows access to various LOOPS functions. To the right of the interpreter control menu is a pop-up menu, waiting for the user to assist in the selection of an appropriate method for proposing a distillation column design. The pop-up menu overlaps the DSPL++ database browser. Above the database browser is the knowledge base browser, showing all available proposers in the knowledge base. Around the right and bottom edges of the screen are icons of file browsers for various pieces of the DSPL++ architecture.
Appendix B

User’s Manual

This chapter describes the language primitives available to the DSPL++ programmer for writing methods in the DSPL++ language.

B.1 Support Functions

All of the functions defined in this section may be used in any method body. They are recognized by the DSPL++ compiler by being listed in the top level variable AdeptSupportFns.

B.1.1 Method Selection

The following functions are used primarily in sponsors and selectors to support the selection of methods.

ALREADY-TRIED? method Returns T if the method has already been applied to the currently active goal context, and NIL otherwise.

ANY? list Returns T if the list is not empty, NIL if it is.

ASK-MENU list title Prompts the user to select an item from the specified list.
**ASK-VALUE attribute** Prompts the user to enter the specified attribute. Any access routines for the attribute are applied.

**ASK-USER message** Prompts the user for a yes/no answer. If the user selects "no", the currently active goal terminates with failure.

**COMBINE s1 s2** Returns the combination of two suitabilities.

**COMPLAIN message** Prints the specified message with appropriate fanfare in the prompt window.

**DESIGNER-PREFERENCE** Selects a method from a group of methods. Depending on the interpreter modes, this method may automatically select a method or present the user with a list of methods allowing the user to make the selection.

**FAIL** Causes the currently active goal to terminate with failure.

**METHODS-RATED-AS suitability** Returns all methods in the current context having the specified suitability.

**ONLY-ONE? list** Returns T if the list contains a single element, NIL otherwise.

**SUCCESS** Causes the currently active goal to terminate with success.

### B.1.2 Accessing The Design Database.

The following functions define how the design database can be accessed from methods in DSPL++. The following functions are available to all methods.
ATTRIBUTES-SET? ObjectName  This function checks all attributes set by the specified method and returns one of Some, None, All, or NA, depending on how what attributes are currently set. NA is returned if the method changes no attributes. This function is used in method sponsors.

COMPONENT-COMPLETE?  A Boolean function that checks the output attributes of the current artifact to see if they are all set. Returns T if all attributes are set, and NIL otherwise.

INITIALIZE ObjectName  Explicitly creates a new instance of a database object and places it in the currently active case.

INPUT-AVAILABLE? Method  This function checks all inputs required by the specified method and returns one of Some, None, or All, depending on what attributes are currently set in the current case. This method is used in method sponsors.

KB-FETCH ObjectName Attribute no-trigger  Seeks out an artifact instance of the specified name in the design database and returns the current value of the specified attribute. If no value is currently available, any access functions are executed to generate a value and the new value is stored before being returned. The no-trigger argument inhibits any access functions from being executed.

KB-STORE ObjectName Attribute Value  Stores the value of an attribute in the design database under the specified object. If no artifact instance is found of the specified type, then a new instance is initiated.
**KB-VALUE-SET**? Object\-Name Attribute A function used by COMPONENT- COMPLETE? to check the values of the output attributes of the named object.

**REPORT-ON** Object\-Name Puts up a display of a database object.

### B.2 Language Primitives

This section lists the reserved tokens that have special meaning to the DSPL++ compiler. They may be used in any expression.

**ARTIFACT** Returns the name of the object currently being designed.

**ARTIFACT-INSTANCE** Returns the instance pointer to the object currently being designed.

**CURRENT-CONTEXT** The currently active goal context.

**EXECUTION-MODE** The list of modes governing the interpreter’s execution, such as what aspects of the design process are being traced, and the level of interaction with the user during design.

**EXPLANATION** This term is bound to the text associated with reason for entering the current goal state.

**FAILED-METHOD** Within a plan during redesign, this term identifies the plan item that caused redesign to be invoked.

**FAILED-METHOD-TYPE** The type of FAILED-METHOD.
FAILING-CONTEXT The goal context that has spawned the currently active Resolve Failure goal. This term is NIL except within the Resolve Failure context.

FAILING-CONTEXT-METHODS A list of methods that have been attempted within the FAILING-CONTEXT.

FAILING-METHOD Extracted from the FAILING-CONTEXT.

FAILING-METHOD-NAME The name of the FAILING-METHOD.

FAILING-METHOD-TYPE The type of the FAILING-METHOD.

FAILING-VALUE This term is set during constraint failure by a constraint that describes a simple Boolean relationship on a numerical value. It contains the value of the left-hand side of the relationship.

FAILURE-AMOUNT This term is set during constraint failure by a constraint that describes a simple Boolean relationship on a numerical value. It contains the absolute value of the difference between the left and right-hand sides of the relationship, i.e., \( \text{ABS}(- \text{FAILING-VALUE} \text{SUCCESS-IF}) \). earlier version of DSPL.

METHOD The currently active method, or, within a sponsor, the method being sponsored.

METHOD-NAME The name of METHOD.

METHOD-TYPE The type of METHOD.
METHODS-ATTEMPTED Within a plan, this term lists the items of the plan that have been attempted during plan execution. This list is used by the LEAST-BACKUP and other redesign strategies to determine what plan items should be informed that redesign is in progress.

METHODS-TRIED A list of the methods used to try to satisfy the CURRENT-CONTEXT.

MODE The current design mode, as in Design, RoughDesign, etc.

PLAN This term is bound to any plan that is currently active somewhere in the active context.

PROBLEM-SOLVER The currently active knowledge base. In earlier versions of DSPL, a different problem solver was associated with each domain. In DSPL++, all methods are contained within a single knowledge base, and hence this term does not provide any useful information. It is provided for compatibility with older DSPL systems converted to the DSPL++ language.

PROBLEM-SOLVER-NAME The name of PROBLEM-SOLVER.

REDESIGNER-TYPE Describes the kind of suggestions that a redesigner can handle, such as INCREASE, DECREASE, CHANGE, etc. This is used inside the default sponsor for redesigners to determine whether or not the redesigner matches the current suggestion for redesign.
RELEVANT-SUGGESTIONS The set of suggestions within a redesign context as determined by the GET-RELEVANT-SUGGESTIONS statement.

SELECTION-SET Returns the list of pointers to the methods retrieved by the current design context as relevant to the current task.

SPONSORED-METHOD Only available within a sponsor body, this term refers to the method associated with the current invocation of the sponsor.

STEP-VALUE Indicates the current value of an attribute provided to a step redesigner.

SUCCESS-IF This term is set during constraint failure by a constraint that describes a simple Boolean relationship on a numerical value. It contains the value of the right-hand side of the relationship.

SUGGESTED-AMOUNT This is the minimum amount that a numerical value of an attribute should be changed as specified in a redesign step, or FAILURE-AMOUNT, whichever is greater.

SUGGESTION The current suggestion driving redesign.

SUGGESTION-TYPE The type of SUGGESTION, such as INCREASE, DECREASE, CHANGE, etc. This value is only available during redesign.

SUGGESTIONS The current list of failure suggestions as generated by the GENERATE-SUGGESTIONS statement. This value is NIL except during redesign.
B.3 The Goal Behavior Definition Language

This section describes GGBL, the task-specific, high-level language for specifying goal behaviors in DSPL++, as well as design and failure methods. This language was introduced in Chapter III.

**APPLY** Applies the currently selected method to the active goal context. Uses the value of the IV SelectedKnowledgeSource from the goal stack.

**APPLY-IN-FAILING-CONTEXT** Statements Places a list of statements on the top of the agendas of the failing goal. Only used by methods used during failure processing, since only then will there be a threatened goal on the goal stack. An error occurs if this statement is used outside of failure processing.

**APPLY-IN-PARENT-CONTEXT** Statements Places a list of statements on the top of the agenda of the parent of the current goal. This command is sometimes used to optimize performance in cases where a subgoal wishes to short circuit the failure handling mechanisms and, for example, terminate its parent goal directly.

**EVALUATE IV | MethodList** Executes a list of methods, taken either from the arguments, or, if the argument is an ATOM, from the IV named by the argument in the currently active method.

**EVALUATE-SPONSORS** Puts a SUB-GOAL on the agenda of the currently active goal to run the sponsor of each item in the SelectionSet.
FAILURE-METHODS OFF | MethodList Puts a list of method descriptions in the FailureRetrievalKeys IV of the currently active goal context.

GENERATE-SUGGESTIONS Pulls failure suggestions and the failure message off the currently active method and puts them into the goal stack.

LOCAL MethodName Runs a (LOOPS) method local to the goal type.

MENU-SELECTION IV items title Pops up a menu of items using the specified title and places the user-selected item in the specified IV on the goal stack.

PARALLEL IV | StatementList Spawns an instance of the PursueInParallel class to run a list of specified statements in parallel. If the argument is an IV, then the list of commands is taken from the named IV on the goal stack. This happens in the Decomposition behavior definition.

PRINT-WARNING message Prints a warning to the user in the top prompt window.

PUT-ON-STACK IV value Places a value on the goal stack.

RESTART GoalType Restarts an instance of the named goal type.

RESUME GoalExpression Sends a Resume message to the specified goal. The argument is evaluated before being used.

RETRIEVE Spawns the Retrieve subgoal, using the arguments as characteristics for the index unpacker.
SELECT \{USING Selector \mid USING-SELECTOR-IN IV\} Causes a selector to be applied to select a method from the methods listed in the SelectionSet IV of the currently active goal. The result is placed in the SelectedMethod IV.

SUB-GOAL GoalClass \{Method \{IVinitializations\}\} Spawns a subgoal to the currently active goal of the type specified. An optional method may be specified that becomes the active method. The new goal instance may be parameterized by specifying a list of IV/value pairs.

SUSPEND-METHOD \{message\} Suspends the currently active goal and places the optional message on the goal stack. Control is passed to the parent goal.

TERMINATE-AND-HALT \{message\} Displays the message and halts processing. Processing may be resumed by the interpreter control.

TERMINATE-WITH-FAILURE \{message\} Terminates the current goal with failure and marks it with the specified message in the goal history stack. The state of the current goal is set to “Failed.”

TERMINATE-WITH-SUCCESS \{message\} Terminates processing of the current goal with success and marks it with the specified message in the goal history stack. The state of the current goal is set to “Satisfied.”

B.3.1 Primitives For Controlling Execution Flow

The following commands manage the flow of control of execution within the active agenda of a goal. They are available to all goal types.
DO Statements Takes a group of commands and puts them on the agenda of the currently active goal.

IF expr THEN statementList {ELSE statementList} If the expression evaluates to T, places the first statement list on the agenda of the currently active goal, otherwise places the second list on the agenda, if specified.

POP-COMMAND-ITEM Statements Removes one statement from a list of statements and places both back on the agenda. This command is used to implement a stack of multiple command streams, each of which is popped off the command list as it is executed. This type of action is useful for associating, for example, a list of commands for each of a different suggestion for fixing a failure. If each of the statements ends with a SKIP command, then the first command that successfully completes pops the rest off the agenda.

REPEAT Statement (UNTIL Test | WHILE Test | FOREVER) Simple iteration loop. All tests are performed at the beginning of the loop. The Statement is placed on the agenda of the currently active goal followed by the complete REPEAT statement whenever the test is evaluated to be true.

SKIP Skips over the next item in the agenda. This command is used in conjunction with the POP-COMMAND-ITEM in the implementation of plan redesign strategies.
B.4 The Method Behavior Definition Language

This section describes the additional commands that, together with the commands listed above, make up GMBL, a simple language for specifying the control behaviors of methods in DSPL++, as described in Chapter III. These commands are available to all method types.

**EVALUATE-PAIRS-BODY** The function describing the body of the currently active method is retrieved from the method and executed. The returned value of the function, indicated by the REPLY in the body of the method, is placed in the Result IV of the currently active goal context.

**PUSH-ONTO-AGENDA IVName** Pushes a list of statements to be executed onto the agenda of the currently active goal. The statements are taken from an IV of the currently active method.

B.4.1 Plan Language Extensions

The GMBL can be extended to include commands specific to a particular method type. The following extensions are defined within the Plan Method Class.

**DESIGN ArtifactName** Creates a Design Artifact subgoal for the specified object, with DesignMode = Design.

**EXECUTE-AGENT NameTypePair** Looks in the knowledge base for the indicated method, and applies it to the current context. An error message is printed if the specified method does not exist.
PLAN-REDESIGN FromFailedItem | Item {Suggestions} Used only by the Plan Redesign Failure Method to invoke redesign on plans and tasks.

ROUGH-DESIGN ArtifactName Same as DESIGN, but sets DesignMode = RoughDesign.

STEP-REDESIGN (OnCurrentStep | Item) {Suggestions} Used only by the PlanRedesign and StepRedesign Failure Methods to invoke redesign on steps.

B.5 Accessing the Goal Stack

This section describes several LOOPS methods useful in accessing the values stored in the goal stack from within both method and goal behavior definitions. They may be used in any expression within the method and goal behavior definitions.

GetLanguagePrimitiveValue PrimitiveName Extacts the current value of one of the GGBL or GMBL language primitives, as listed in section B.2 above.

GetValueFromGoalStack IVName seek? Extracts the value of an IV from the goal stack. The seek flag indicates whether or not to halt when a NIL value is found.

GetValueFromMethodOnGoalStack IVName Similar to above, but also checks the IVs of any active methods on the way up the stack.

PutLanguagePrimitiveValue PrimitiveName Saves the value of a language primitive. Note that only certain values may be set this way, as described in the section B.2 above.
PutValueOnGoalStack IVName Value  Sets the value of an IV on the goal stack, wherever it is found. If the current goal does not have the IVName listed, the parent goal is checked.
Appendix C

Implementation Notes

This chapter contains details about the implementation of the DSPL++ architecture useful to the user attempting to extend either the architecture's method or task definitions structure.

C.1 Goal State Transitions

The DSPL++ architecture defines several goal state to help organize the processing of goals. Each of these states are processed by a corresponding LOOPS method defined in the goal class. Default state processing methods are provided in ($ Goal$). These methods are inherited by all DSPL++ goal classes. New methods may be added to specialize the behavior for a particular goal type.

Figure 62 describes the valid state transitions for goals as recognized by the goal interpreter in the DSPL++ architecture. All terminal states are shadowed in the figure.

All new goals begin in the Initialize state. Any attempts to make a transition not listed in Figure 62 are invalid and cause the interpreter to terminate all processing immediately, at the point where the transition was attempted, with a system failure.
Note that most of the states and transitions described are handled automatically by the goal interpreter and are otherwise transparent to the DSPL++ programmer. The Initialize state, for example, is entered whenever a new goal is created. The DSPL++ programmer can only indirectly manipulate these states through the DSPL++ behavior definition language. Other state transitions are effected more directly. For example, a programmer may explicitly terminate a goal with success or failure, thus causing a transition of an active goal to the corresponding state.
C.2 Method Syntax and the DSPL++ Compiler

This section describes the details of the DSPL++ compiler, including the concrete syntax for DSPL++ methods.

All methods entered into the DSPL++ knowledge base by creating source code forms that are handed to the DSPL++ compiler for syntax checking and translation. Both methods that are entered through the knowledge base browser as well as those added via the menu-driven knowledge acquisition are handed over to the compiler for translation. The result of the translation process is a LOOPS instance that is cataloged in the DSPL++ knowledge base.

The DSPL++ compiler makes various syntax and semantic checks on the method source code. For example, all references to devices and device attributes within methods are checked against the corresponding definitions in the design database. The user is warned when references to undefined attributes and devices are detected.

The compiler's parsing is driven by a series of grammars that define the syntactic forms acceptable to the DSPL++ architecture. There are five interrelated grammars that define the DSPL++ syntax, each of which describes a different portion of the language syntax.

- Common: The top level grammar is common to all methods, and receives all compilation requests. This grammar essentially describes a language that consists of lists of features to be filled for each of the different method types. This grammar is fairly simple, and hands off certain sub parsing problems to the other four grammars.
• Suggestions: This grammar defines the syntax of a suggestion in the DSPL++
language. All suggestion forms are handled by this grammar.

• Plan: The plan grammar defines the forms that a plan may take. The plan
grammar describes the valid plan actions that may appear in a plan.

• Body: The body grammar defines the familiar DSPL step body syntax. The
body of a step consists of a KNOWN section, where attribute values may be
retrieved from the design database, and the DECISION section, where com­
putations are described. The syntax of each of these sections consists of pairs
of items, the first being a variable that is assigned the value of the second
item, which is an expression to be evaluated. The output of this translator is a
function that implements the body specification. This grammar is used for all
methods that have a "pairs list" syntax.

• Expression: A basic prefix expression grammar. All expressions are handed off
to this grammar for parsing. The output of this grammar is a lisp expression
that can be evaluated.

Figure 63 shows the relationships among the grammars.

The following tables list the definitions of the five subgrammars. Boldface items
are keywords in the language. Lower case items are lexical primitives. All capitalized
items are meta-symbols in the grammar and do not appear in the surface forms.
Figure 63: The structure of the DSPL++ modular grammar

Table 2: The main grammar

<table>
<thead>
<tr>
<th>Construct</th>
<th>→ Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>→ ( Label List )</td>
</tr>
<tr>
<td>Definition</td>
<td>→ ( Label List Body )</td>
</tr>
<tr>
<td>Body</td>
<td>→ ( BODY S-expressions )</td>
</tr>
<tr>
<td>S-expressions</td>
<td>→ s-expression</td>
</tr>
<tr>
<td>S-expressions</td>
<td>→ S-expressions s-expression</td>
</tr>
<tr>
<td>Label</td>
<td>→ Agent'Type</td>
</tr>
<tr>
<td>List</td>
<td>→ Item</td>
</tr>
<tr>
<td>List</td>
<td>→ List Item</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( agent-slot-label NONE )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( agent-slot-label Identifiers )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( discard-slot )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( discard-slot S-expressions )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( FAILURE-HANDLER FailureHandlerType IS identifier )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( FAILURE-HANDLER NONE )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( FAILURE-HANDLER identifier )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( NAME identifier )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( string-slot-label string )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( SUGGESTIONS NONE )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( SUGGESTIONS S-expressions )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( token-slot-label Identifiers )</td>
</tr>
<tr>
<td>Item</td>
<td>→ ( TYPE identifier )</td>
</tr>
<tr>
<td>Identifiers</td>
<td>→ identifier</td>
</tr>
<tr>
<td>Identifiers</td>
<td>→ Identifiers identifier</td>
</tr>
</tbody>
</table>
Table 3: The decision body grammar

Construct → Body
Body → ( CONSTRAINT-BODY Knowns Test )
Body → ( CONSTRAINT-BODY Test )
Body → ( FAILURE-BODY Statements )
Body → ( SPONSOR-BODY Knowns Decisions )
Body → ( STEP-BODY Decisions )
Body → ( STEP-BODY Knowns Decisions )
Decisions → ( DECISIONS *** )
Decisions → ( DECISIONS AssignmentPairs )
AssignmentPairs → AssignmentPair
AssignmentPairs → AssignmentPairs AssignmentPair
AssignmentPair → REPLY Statement
AssignmentPair → identifier Statement
Statement → ( DO Statements )
Statement → ( IF Expression THEN Statement )
Statement → ( IF Expression THEN Statement ELSE Statement )
Statement → ( TABLE ColumnSetUp Matches )
Statement → ( TABLE ColumnSetUp Matches Default )
Statement → Expression
Default → ( OTHERWISE Statement )
ColumnSetUp → ( DEPENDING-ON Expressions )
Expressions → Expression
Expressions → Expressions Expression
Matches → ( MATCH Rows )
Rows → Row
Rows → Rows Row
Row → ( IF ( RowExpressions ) THEN Statement )
RowExpressions → RowExpression
RowExpressions → RowExpressions RowExpression
RowExpression → ( Expression >=< Expression )
RowExpression → ( RelOp Expression )
RowExpression → Expression
Expression → ( S-expressions )
Expression → AnyToken
S-expressions → s-expression
S-expressions → S-expressions s-expression
Statements → Statement
Statements → Statements Statement
Knowns → ( KNOWN *** )
Knowns → ( KNOWN AssignmentPairs )
Test → ( TEST TestExpression )
TestExpression → s-expression
Table 4: The plan body grammar

Construct → Body
Body → (BODY StatementList )
Body → (DECOMP-BODY Component
DECOMPOSES-TO ComponentList )
Component → identifier
ComponentList → Component
StatementList → Statement
StatementList → StatementList Statement
Statement → (DESIGN Component )
Statement → (EXECUTE s-expression )
Statement → (EXECUTE-AGENT identifier )
Statement → (INITIALIZE Component )
Statement → (PARALLEL StatementList )
Statement → (REPORT-ON Component )
Statement → (ROUGH-DESIGN Component )
Statement → (TEST-CONSTRAINT identifier )
Statement → identifier

Table 5: The suggestion grammar

Construct → Suggestion
Suggestion → Conditional
Suggestion → Suggest
Suggest → (SUGGEST Command )
Suggest → Command
Command → (CommandType Attribute OF Component )
Command → (CommandType Attribute OF Component BY Amount )
Command → (CommandType Attribute OF Component BY CommandType Amount )
Amount → Expression
Amount → FAILURE-AMOUNT
Expression → s-expression
Attribute → identifier
Component → identifier
Conditional → (IF Expression THEN Suggest )
Table 6: The expression grammar

<table>
<thead>
<tr>
<th>Construct</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>( LISP s-expression )</td>
</tr>
<tr>
<td>Expression</td>
<td>( QUOTE s-expression )</td>
</tr>
<tr>
<td>Expression</td>
<td>( RelOp Expression Expression )</td>
</tr>
<tr>
<td>Expression</td>
<td>( SUGGEST s-expressions )</td>
</tr>
<tr>
<td>Expression</td>
<td>( SupportFn )</td>
</tr>
<tr>
<td>Expression</td>
<td>( SupportFn Expressions )</td>
</tr>
<tr>
<td>Expression</td>
<td>Constant</td>
</tr>
<tr>
<td>Expression</td>
<td>Primitive</td>
</tr>
<tr>
<td>Expression</td>
<td>identifier</td>
</tr>
<tr>
<td>Constant</td>
<td>Number</td>
</tr>
<tr>
<td>Constant</td>
<td>String</td>
</tr>
<tr>
<td>Constant</td>
<td>SymbolicResult</td>
</tr>
<tr>
<td>Expressions</td>
<td>Expression</td>
</tr>
<tr>
<td>Expressions</td>
<td>Expressions Expression</td>
</tr>
</tbody>
</table>

C.2.1 Lexical Processing in the Expression Grammar

Many of the terminal symbols in the Expression grammar have lexical processors associated with them that determine what types of tokens will be accepted in the input source as an instance of that token. This section describes these lexical details.

The “Number” terminal accepts any token accepted by the NUMBERP lisp predicate. “String” uses the STRINGP lisp predicate.

The “SymbolicResult” terminal check the token for membership in the top level variable UserSymbolicResults.

The “SupportFn” terminal accepts any of the support functions as described in Appendix B. In addition, the following math operations are interpreted by the expression grammar as SupportFns, and have the usual meaning. All math operators
accept tolerated lengths, as in the original version of DSPL:

$$+,-,\ast,/,\div,\mod,$$

The following relational operators are interpreted by the expression grammar as "RelOps", and have the usual meaning. In addition, all relational operators accept tolerated lengths, as in the original version of DSPL:

$$<,\leq,\neq,=,\geq,$$

For non-numeric operators, the following relational operators are also accepted:

\texttt{IS, IS-NOT, IS-NOT-ONE-OF, IS-ONE-OF}

The "Primitive" terminal in the expression grammar accepts one of the language primitive tokens as described in Appendix B.
Appendix D

A Trace Showing The Use of Multiple Problem Solving Methods

This chapter demonstrates the action of the DSPL++ architecture during the design of a hydrocarbon distillation column. The domain knowledge used is part of the STILL knowledge based system ([Myers et al., 1988]), a design system implemented on top of the DSPL++ architecture.

This particular design run uses the PVCM method for design at five different points during problem solving. The first application is used to address the overall design of the column. The other invocations are for the four major subdesign problems encountered within the column design. Four different methods representing three different method types are used within the PVCM method applications to propose solutions to the different design problems encountered. The top level problem is tackled using a decomposition for distillation columns. The the upper and lower sections of the column are each designed using a design plan, while the condenser design problem is handled using a solution to a previous condenser design problem. The reboiler design problem in this example is stubbed out. Each different method application is appropriately noted in the trace.

In addition to the above methods for proposing designs, three verification methods
are also applied throughout the trace: two verification plans and the default AskUser method.

Finally, a DSPL style plan selection specialist is used for the design of the sieve trays in the column. The plan selection specialist represents a compiled design strategy used in place of the PVCM method for design that shortcuts that method’s mechanisms for verifying a completed design.

The goal tree resulting from the sample run is shown in Figure 64. The top level goal that initiated problem solving is shown at the top of the figure. The task-level goals described in Chapters III and IV are shown in this figure in boldface type. Methods used to pursue the task-level goals are shown attached to their goals in a lighter type. The detail of the selection and retrieval goals within each of the task-level goals is suppressed in this figure to aid in clarity and brevity.

Figure 69 shows the detail of the goal tree within the design of the Stripping Section of the column. Again, the task-level goals are bold-faced. In this figure, the retrieval and selection goals have been included.

In Figure 70 can be seen the results of temporary design failure and redesign that occurred within the design of the sieve tray in the Stripping Section of the column. A Plan Redesigner associated with the final tray design task was used to successfully recover from the failure.

A partial list of the results of design are listed in Figure 65. This figure shows the attributes and values set during the design of one of the sieve trays.
Figure 64: The final goal stack
<table>
<thead>
<tr>
<th>Enriching section sieve tray &amp; Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>active area (ft**2)</td>
</tr>
<tr>
<td>average liquid flowpath width (ft)</td>
</tr>
<tr>
<td>downcomer chord height (ft)</td>
</tr>
<tr>
<td>derating factor</td>
</tr>
<tr>
<td>diameter of tray (ft)</td>
</tr>
<tr>
<td>downcomer area (ft**2)</td>
</tr>
<tr>
<td>flood factor</td>
</tr>
<tr>
<td>liquid flowpath length (ft)</td>
</tr>
<tr>
<td>number of passes</td>
</tr>
<tr>
<td>tray spacing (in)</td>
</tr>
<tr>
<td>total area of tray (ft**2)</td>
</tr>
<tr>
<td>weir length (ft)</td>
</tr>
</tbody>
</table>

Figure 65: Part of the design results

D.1 The Design Trace

The following output was generated by the DSPL++ system using the Attribute Trace and Goal Trace features as described in Appendix A. These features record the results of every state change within each goal, the results of every access to the design database, and the result of every attempt to retrieve methods from the DSPL++ knowledge base during design.

Each of the task-level goals placed on the goal stack is introduced in the trace with the string "New Goal:". All activity taken by the interpreter on behalf of that goal will occur at the same level of indentation as this initial indication of the goal.

The level of indentation of the following text indicates the depth from the root of the goal tree: the greater indentation level corresponds to a greater distance from the root.

The text of the trace is set in a typewriter font. The text is interspersed with
annotations on the progress of the design, as well as figures indicating any interactions
the DSPL++ architecture initiates with the user during problem solving.

New goal: design a DistillationColumn
Deactivating the DesignArtifact goal
Beginning to Initialize the DistillationColumn object
Reporting on the DistillationColumn
Initialization successful
Reactivating the goal to design a DistillationColumn
Deactivating the DesignArtifact goal
Looking for methods matching (MetaMethodType DesignStrategy)
(Artifact DistillationColumn) (DesignMode DESIGN)

Note that these particular features used by the Retrieve goal to find relevant methods
are determined by the behavior definition of the Design Artifact goal. (See Chapter
IV for the listing of the Design Artifact goal behavior.)

Retrieve successful Methods = (Trade Study DesignMethod) (PVCM
DesignMethod) (KRITIK DesignMethod)

There are three general-purpose strategies for design available at this point. However,
as will be seen, only the PVCM strategy is every considered for use because of the
evaluations of the method sponsors.

Reactivating the goal to design a DistillationColumn
Deactivating the DesignArtifact goal
Beginning to select an appropriate method
Deactivating the Default Selector
Beginning to apply the Bad method Sponsor
Application of the Bad method Sponsor was successful Result =
UNSUITABLE
Reactivating the Default Selector
Deactivating the Default Selector
Beginning to apply the Suitable Sponsor
Application of the Suitable Sponsor was successful Result =
SUITABLE
Reactivating the Default Selector
Deactivating the Default Selector
Beginning to apply the Default Sponsor
Application of the Default Sponsor was successful Result =
INDIFFERENT
Reactivating the Default Selector
Application of the Default Selector was successful Result =
/>.($& DesignMethod (SW[0.0X:.1j3.ZQ9 . 1470))
Reactivating the goal to design a DistillationColumn
Deactivating the DesignArtifact goal
Beginning to apply the PVCM DesignMethod
The first activation of the PVCM method. It begins, as all applications of this method, with an attempt to propose an appropriate design candidate.

Deactivating the PVCM DesignMethod
New goal: ProposeCandidateDesign
Deactivating the ProposeCandidateDesign goal
Looking for methods matching (MetaMethodType Proposer)
(Artifact DistillationColumn) (DesignMode DESIGN)

Looking for methods that propose distillation column designs. There are three methods available, a case, a decomposition and a plan.

Retrieve successful Methods = (DistillationColumn-3452 Case)
(SimpleDistillationColumn Decomposition) (Klein Plan)
Reactivating the goal to ProposeCandidateDesign
Deactivating the ProposeCandidateDesign goal
Beginning to select an appropriate method
Deactivating the Proposer Selector
Beginning to apply the DistillationColumn-3452 Sponsor
Application of the DistillationColumn-3452 Sponsor was successful Result = INDIFFERENT
Reactivating the Proposer Selector
Deactivating the Proposer Selector
Beginning to apply the SimpleDistillationColumn Sponsor
Fetching the DistillationColumn NumberOfFeeds = 1
Fetching the DistillationColumn NumberOfSideStreams = 0
Application of the SimpleDistillationColumn Sponsor was successful Result = PERFECT
Reactivating the Proposer Selector
Deactivating the Proposer Selector
Beginning to apply the Klein Sponsor
Before determining the suitability of the Klein plan, the sponsor checks several input features that have not been specified in the input specifications. The system detects this and prompts the user for them by presenting a calculator-like input menu that describes the needed input. After the user enters the necessary data, problem solving resumes. Figures 66 and 67 show the menus presented to the user at this time.

Fetching the DistillationColumn NumberOfFeeds = 1
Fetching the DistillationColumn NumberOfSideStreams = 0
Application of the Klein Sponsor was successful Result = SUITABLE
Reactivating the Proposer Selector
Application of the Proposer Selector was successful Result = #.(SN[0.0X:.153.2Q9 . 1519])
Reactivating the goal to ProposeCandidateDesign

The decomposition looks like the best alternative. It is selected and applied to the current context. Several constraints are now checked by the decomposition to insure that all preconditions for the decomposition have been met.
Figure 66: Prompting for a missing attribute value

Figure 67: Prompting for a missing attribute value
Deactivating the ProposeCandidateDesign goal
Beginning to apply the SimpleDistillationColumn Decomposition
Deactivating the SimpleDistillationColumn Decomposition
Beginning to apply the HydroCarbonFeed? Constraint
Fetching the DistillationColumn Components =
(#.($& HydroCarbon (SN[0.0X:.lj3.ZQ9 . 1718))
#.(## HydroCarbon (SN[0.0X:.lj3.ZQ9 . 1719))
#.(## HydroCarbon (SN[0.0X:.lj3.ZQ9 . 1720))
#.(## HydroCarbon (SN[0.0X:.lj3.ZQ9 . 1721))
#.(## HydroCarbon (SN[0.0X:.lj3.ZQ9 . 1722))
#.(## HydroCarbon (SN[0.0X:.lj3.ZQ9 . 1723)))
Application of the HydroCarbonFeed? Constraint was successful Result = T
Reactivating the SimpleDistillationColumn Decomposition
Deactivating the SimpleDistillationColumn Decomposition
Beginning to apply the SingleFeed? Constraint
Fetching the DistillationColumn NumberOfFeeds = 1
Application of the SingleFeed? Constraint was successful Result = T
Reactivating the SimpleDistillationColumn Decomposition
Deactivating the SimpleDistillationColumn Decomposition
Beginning to apply the NoSideStreams? Constraint
Fetching the DistillationColumn NumberOfSideStreams = 0
Application of the NoSideStreams? Constraint was successful Result = T
Reactivating the SimpleDistillationColumn Decomposition
The system now begins to map distillation column specifications into the specifications of its components...

Deactivating the SimpleDistillationColumn Decomposition
Beginning to apply the ReboilerFlow SpecificationMapper
Fetching the DistillationColumn FeedFlowRate = 350
Storing 350 as the FeedFlowRate of Reboiler
Application of the ReboilerFlow SpecificationMapper was successful Result = 350
Reactivating the SimpleDistillationColumn Decomposition
Deactivating the SimpleDistillationColumn Decomposition
Beginning to apply the CondenserFlow SpecificationMapper
Fetching the DistillationColumn FeedFlowRate = 350
Storing 350 as the FeedFlowRate of Condenser
Application of the CondenserFlow SpecificationMapper was successful Result = 350
Reactivating the SimpleDistillationColumn Decomposition
Deactivating the SimpleDistillationColumn Decomposition
Beginning to apply the StrippingSectionData SpecificationMapper
Fetching the SimulatedDistillationColumn FeedPlateNumber = 6
Fetching the SimulatedDistillationColumn PlateData =
Storing `SimulatedSieveTray` as the `SimulatedSieveTray` of `StrippingSection` Application of the `StrippingSectionData SpecificationMapper` was successful Result = `SimulatedSieveTray`

Reactivating the `SimpleDistillationColumn Decomposition` Deactivating the `SimpleDistillationColumn Decomposition` Beginning to apply the `StrippingSectionVacuum` SpecificationMapper

Fetching the `DistillationColumn VacuumOperation?` = Unknown Storing Unknown as the `VacuumOperation?` of `StrippingSection` Application of the `StrippingSectionVacuum` SpecificationMapper was successful Result = Unknown Reactivating the `SimpleDistillationColumn Decomposition` Deactivating the `SimpleDistillationColumn Decomposition` Beginning to apply the `StrippingSectionFoam` SpecificationMapper

Fetching the `DistillationColumn FoamingTendency?` = nonfoaming Storing nonfoaming as the `FoamingTendency?` of `StrippingSection` Application of the `StrippingSectionFoam SpecificationMapper` was successful Result = nonfoaming Reactivating the `SimpleDistillationColumn Decomposition` Deactivating the `SimpleDistillationColumn Decomposition` Beginning to apply the `EnrichingSectionData` SpecificationMapper

Fetching the `SimulatedDistillationColumn FeedPlateNumber` = 6 Fetching the `SimulatedDistillationColumn PlateData` =

(`SimulatedSieveTray`)

Reactivating the `SimpleDistillationColumn Decomposition` Deactivating the `SimpleDistillationColumn Decomposition`
The system now addresses in turn each of the sub design problems of the distillation column, beginning with the Stripping Section. All of the design subproblems can theoretically be pursued in parallel, although the current serial implementation handles the problems one at a time.

Now must find methods for designing the Stripping Section.
Beginning to select an appropriate method
Deactivating the Default Selector
Beginning to apply the Bad method Sponsor
Application of the Bad method Sponsor was successful
Result = UNSUITABLE
Reactivating the Default Selector
Deactivating the Default Selector
Beginning to apply the Suitable Sponsor
Application of the Suitable Sponsor was successful
Result = SUITABLE
Reactivating the Default Selector
Deactivating the Default Selector
Beginning to apply the Default Sponsor
Application of the Default Sponsor was successful
Result = INDIFFERENT
Reactivating the Default Selector
Application of the Default Selector was successful
Result = #.($& DesignMethod (SN[0.0X:.ij3.ZQ9 . 1470))
Reactivating the goal to design the StrippingSection
Deactivating the DesignArtifact goal

Using the PVCM method for the second time. This time, only a single method is available for proposing a candidate design, the LowCostSection plan.

Beginning to apply the PVCM DesignMethod
Deactivating the PVCM DesignMethod
New goal: ProposeCandidateDesign
Deactivating the ProposeCandidateDesign goal
Looking for methods matching (MetaMethodType Proposer) (Artifact StrippingSection) (DesignMode DESIGN)
Retrieve successful Methods = (LowCostSection Plan)
Reactivating the goal to ProposeCandidateDesign
Deactivating the ProposeCandidateDesign goal
Beginning to select an appropriate method
Deactivating the Proposer Selector
Beginning to apply the Suitable Sponsor
Application of the Suitable Sponsor was successful
Result = SUITABLE
Reactivating the Proposer Selector
Application of the Proposer Selector was successful
Result = #.($& Plan (SN[0.0X:.ij3.ZQ9 . 1518))
Reactivating the goal to ProposeCandidateDesign
Deactivating the ProposeCandidateDesign goal
Beginning to apply the LowCostSection Plan
Deactivating the LowCostSection Plan

This plan directly invokes a plan selection specialist for designing a sieve tray. The SieveTray specialist behaves much like a design specialist in the original DSPL, selecting a plan from a set of plans. Note that this method selection occurs only over
plans, and that the retrieval from the knowledge base names methods directly, rather than asking for general method characteristics.

Beginning to apply the SieveTray PlanSelectionSpecialist
Deactivating the SieveTray PlanSelectionSpecialist
Looking for methods matching OR ((Name SieveTray) (MethodType Plan))
Retrieve successful Methods = (SieveTray Plan)
Reactivating the SieveTray PlanSelectionSpecialist
Deactivating the SieveTray PlanSelectionSpecialist
Beginning to select an appropriate method
Deactivating the Default Selector
Beginning to apply the Suitable Sponsor
Application of the Suitable Sponsor was successful
Result = SUITABLE
Reactivating the Default Selector
Application of the Default Selector was successful
Result = 
$.($& Plan (SN[0.0X:.1j3.Z9# . 1477))
Reactivating the SieveTray PlanSelectionSpecialist
Deactivating the SieveTray PlanSelectionSpecialist

The structure of the sieve tray plan is also listed in Figure 56, in Appendix A. At this point the system begins to set the values of attributes in the components of the design.

Beginning to apply the SieveTray Plan
Deactivating the SieveTray Plan
Beginning to apply the PreliminaryCalculations PrimitivePlan

Several attributes values are computed in preparation for the design of the sieve tray proper. For brevity's sake, this section has been omitted. The task completes without problem.

... Application of the PreliminaryCalculations PrimitivePlan was successful
Reactivating the SieveTray Plan
Deactivating the SieveTray Plan
Beginning to apply the InitialTrayDesign PrimitivePlan
Deactivating the InitialTrayDesign PrimitivePlan
Beginning to apply the InitialSpacing DesignStep
Storing 12.0 as the Spacing of SieveTray
Application of the InitialSpacing DesignStep
was successful Result = 12.0
Reactivating the InitialTrayDesign
PrimitivePlan
...

More initial values are computed.
...
Application of the InitialTrayDesign
PrimitivePlan was successful
Reactivating the SieveTray Plan
Deactivating the SieveTray Plan
Beginning to apply the FinalTrayDesign
PrimitivePlan
Deactivating the FinalTrayDesign PrimitivePlan
Beginning to apply the Spacing DesignStep
Storing 12.0 as the Spacing of SieveTray
Application of the Spacing DesignStep was
successful Result = 12.0
Reactivating the FinalTrayDesign PrimitivePlan
Deactivating the FinalTrayDesign PrimitivePlan
Beginning to apply the DowncomerArea
DesignStep
Fetching the SieveTray ActiveArea = 13.02509
Fetching the SieveTray GPMLiquidFlowRate =
140.7335
Fetching the SieveTray FloodFactor = .82
Fetching the SieveTray DeratingFactor = 1.0
Fetching the SieveTray Spacing = 12.0
Fetching the SimulatedSieveTray
LiquidDensity = 28.687
Fetching the SimulatedSieveTray VaporDensity
= 1.478
Storing 1.43276 as the DowncomerArea of
SieveTray
Application of the DowncomerArea DesignStep
was successful Result = 1.43276
Reactivating the FinalTrayDesign PrimitivePlan
Deactivating the FinalTrayDesign PrimitivePlan
Beginning to apply the TotalArea DesignStep
Fetching the SieveTray ActiveArea = 13.02509
Fetching the SieveTray DowncomerArea =
1.43276
Storing 15.89061 as the TotalArea of
SieveTray
Application of the TotalArea DesignStep was
successful Result = 15.89061
Reactivating the FinalTrayDesign PrimitivePlan
Deactivating the FinalTrayDesign PrimitivePlan
Beginning to apply the Diameter DesignStep
Fetching the SieveTray TotalArea = 15.89061
Storing 4.5 as the Diameter of SieveTray
Application of the Diameter DesignStep was successful Result = 4.5
Reactivating the FinalTrayDesign PrimitivePlan
Deactivating the FinalTrayDesign PrimitivePlan
Beginning to apply the Spacing Constraint
Fetching the SieveTray Diameter = 4.5
Fetching the SieveTray Spacing = 12.0
Failed to (apply the Spacing Constraint)
Reactivating the FinalTrayDesign PrimitivePlan

A problem has arisen! The spacing of the sieve trays is not compatible with the tray's diameter. A suggestion (not shown) implicates the tray spacing. An instance of the Resolve Failure goal attempts to decide what to do next...

Deactivating the FinalTrayDesign PrimitivePlan
New goal: ResolveFailure
Deactivating the ResolveFailure goal
Looking for methods matching OR ((Name PlanRedesign) (MethodType FailureMethod))
((Name FailWithSuggestions) (MethodType FailureMethod))
Retrieve successful Methods =
(PlanRedesign FailureMethod)
(FailWithSuggestions FailureMethod)
Reactivating the goal to ResolveFailure
Deactivating the ResolveFailure goal
Beginning to select an appropriate method
Deactivating the Default Selector
Beginning to apply the PlanRedesign Sponsor
Application of the PlanRedesign Sponsor was successful Result = SUITABLE
Reactivating the Default Selector
Deactivating the Default Selector
Beginning to apply the Default Sponsor
Application of the Default Sponsor was successful Result = INDIFFERENT
Reactivating the Default Selector
Application of the Default Selector was successful Result =
#.($& DesignMethod (SN[0.OX:.1j3.ZQ9 . 1479])
Reactivating the goal to ResolveFailure

Plan Redesign is selected as the appropriate action to take.
Deactivating the ResolveFailure goal
Beginning to apply the PlanRedesign FailureMethod
Deactivating the PlanRedesign FailureMethod
New goal: PlanRedesign
Deactivating the PlanRedesign goal
Beginning to apply the FinalTrayDesign PrimitivePlanRedesigner
Application of the FinalTrayDesign PrimitivePlanRedesigner was successful
Reactivating the goal to PlanRedesign
Deactivating the PlanRedesign goal

The plan redesign instigates the redesign of the tray spacing.

New goal: StepRedesign
Deactivating the StepRedesign goal
Looking for methods matching OR
((Name Spacing) (MethodType RedesignStep))
Retrieve successful Methods =
(Spacing RedesignStep)
Reactivating the goal to StepRedesign
Deactivating the StepRedesign goal
Beginning to select an appropriate method
Deactivating the Default Selector
Beginning to apply the DefaultRedesignStep Sponsor
Application of the DefaultRedesignStep Sponsor was successful Result = SUITABLE
Reactivating the Default Selector
Application of the Default Selector was successful Result =
#.($& RedesignStep (SN0.0X:1j3.2Q9 1480))
Reactivating the goal to StepRedesign
Deactivating the StepRedesign goal
Beginning to apply the Spacing RedesignStep
Fetching the SieveTray Diameter = 4.5
Storing 18.0 as the Spacing of SieveTray
Application of the Spacing RedesignStep was successful Result = 18.0
Reactivating the goal to StepRedesign
The StepRedesign goal succeeded
Reactivating the goal to PlanRedesign

So far, so good. The tray spacing has been redesigned. Now any changes must be propagated to the rest of the design. The scope of this change is limited to the attributes set between the point of failure (the Spacing Constraint), and the point at which the tray spacing was originally decided.

Deactivating the PlanRedesign goal
Beginning to apply the DowncomerArea DesignStep
Fetching the SieveTray ActiveArea = 13.02509
Fetching the SieveTray GPMLiquidFlowRate = 140.7335
Fetching the SieveTray FloodFactor = .82
Fetching the SieveTray DeratingFactor = 1.0
Fetching the SieveTray Spacing = 18.0
Fetching the SimulatedSieveTray LiquidDensity = 28.687
Fetching the SimulatedSieveTray VaporDensity = 1.478
Storing 1.43276 as the DowncomerArea of SieveTray
Application of the DowncomerArea DesignStep was successful Result = 1.43276
Reactivating the goal to PlanRedesign
Deactivating the PlanRedesign goal
Beginning to apply the TotalArea DesignStep
Fetching the SieveTray ActiveArea = 13.02509
Fetching the SieveTray DowncomerArea = 1.43276
Storing 15.89061 as the TotalArea of SieveTray
Application of the TotalArea DesignStep was successful Result = 15.89061
Reactivating the goal to PlanRedesign
Deactivating the PlanRedesign goal
Beginning to apply the Diameter DesignStep
Fetching the SieveTray TotalArea = 15.89061
Storing 4.5 as the Diameter of
SieveTray
Application of the Diameter DesignStep
was successful Result = 4.5
Reactivating the goal to PlanRedesign
Deactivating the PlanRedesign goal

Finally, the system retries the spacing constraint with the new tray spacing...

Beginning to apply the Spacing Constraint
Fetching the SieveTray Diameter = 4.5
Fetching the SieveTray Spacing = 18.0
Application of the Spacing Constraint
was successful Result = T
Reactivating the goal to PlanRedesign
The PlanRedesign goal succeeded
Reactivating the PlanRedesign FailureMethod
Application of the PlanRedesign FailureMethod was successful
Reactivating the goal to ResolveFailure
The ResolveFailure goal succeeded
Reactivating the FinalTrayDesign PrimitivePlan
...

Success! Now back to the design of sieve tray proper, which completes without further problem.

... Application of the FinalTrayDesign PrimitivePlan was successful
Reactivating the SieveTray Plan
Reporting on the SieveTray
Application of the SieveTray Plan was successful

The sieve tray plan has completed successfully. This implies that the specialist has also met with success. Unlike the PVCM method, the specialist makes no attempt to verify the design.

Reactivating the SieveTray PlanSelectionSpecialist
Application of the SieveTray PlanSelectionSpecialist was successful
Reactivating the LowCostSection Plan

Now wrapping up the design of the stripping section by deciding the section's throughput...

Deactivating the LowCostSection Plan
Beginning to apply the SectionThroughput DesignStep
Figure 68: Verifying the result of design

Fetching the StrippingSection instance
Storing 500 as the MaxThroughput of StrippingSection

Application of the SectionThroughput DesignStep was successful Result = 500
Reactivating the LowCostSection Plan
Application of the LowCostSection Plan was successful
Reactivating the goal to ProposeCandidateDesign
The ProposeCandidateDesign goal succeeded

Success! Now to verify the completed stripping section...

Reactivating the PVCM DesignMethod
Deactivating the PVCM DesignMethod
New goal: VerifyDesign
Deactivating the VerifyDesign goal
Looking for methods matching (MetaMethodType Verifier) (Artifact StrippingSection)
Retrieve successful Methods = (AskUser DesignMethod)
Reactivating the goal to VerifyDesign
Deactivating the VerifyDesign goal
Beginning to select an appropriate method
Deactivating the Verify Selector
Beginning to apply the AskUser Sponsor
Application of the AskUser Sponsor was successful
Result = PERFECT
Reactivating the Verify Selector
Application of the Verify Selector was successful
Result = #.($& DesignMethod (SN[D.OX:.1j3.ZQ9 . 1741))
Reactivating the goal to VerifyDesign
Deactivating the VerifyDesign goal
Beginning to apply the AskUser DesignMethod
Since there are no domain-specific methods in the knowledge base for verifying the stripping section design, the system resorts to a more general strategy: ask the user. The AskUser method temporarily suspends all processing and pops up a menu as shown in Figure 68. At this point, the user may inspect the results of design, and when satisfied, enter her preference for the design status. The options lists have the following effects:

- **Accept**: Verification succeeds and the design process continues.
- **Reject**: The design is rejected. Verification fails, and the *PVCM* method takes over and attempts to patch up the design through its Critique and Modification tasks.
- **Abort**: Immediately terminates the design process. The design state may be saved and restarted at a later time, or simply discarded.
- **No selection**: Take the default action, which in this case is to accept the design.

The user may optionally generate a display of the attributes designed to this point, similar to the display in Figure 65 in order to aid the decision process. In this case, the design is accepted, and problem solving continues.

Application of the AskUser DesignMethod was successful
Reactivating the goal to VerifyDesign
The VerifyDesign goal succeeded
Reactivating the PVCM DesignMethod
Application of the PVCM DesignMethod was successful
Reactivating the goal to design the StrippingSection
The design the StrippingSection goal succeeded
Reactivating the goal to PursueMethodsInParallel
Deactivating the PursueMethodsInParallel goal
New goal: design the EnrichingSection

The design of the EnrichingSection proceeds similar to the Stripping section. For brevity's sake, this section has been omitted. The design of the enriching section completes without any problems.

The design the EnrichingSection goal succeeded
Reactivating the goal to PursueMethodsInParallel
Deactivating the PursueMethodsInParallel goal

Now on to the reboiler component of the distillation column. Again, the *PVCM* method for design is selected.
Figure 69: A detailed display of the stripping section goal stack
New goal: design the Reboiler
Deactivating the DesignArtifact goal
   Beginning to Initialize the Reboiler object
   Reporting on the Reboiler
   Initialization successful
Reactivating the goal to design the Reboiler
Deactivating the DesignArtifact goal
   Looking for methods matching (MetaMethodType
   DesignStrategy) (Artifact Reboiler) (DesignMode DESIGN)
   Retrieve successful Methods = (Trade Study DesignMethod)
   (PVCM DesignMethod) (KRITIK DesignMethod)
Reactivating the goal to design the Reboiler
Deactivating the DesignArtifact goal
   Beginning to select an appropriate method
Deactivating the Default Selector
   Beginning to apply the Bad method Sponsor
   Application of the Bad method Sponsor was successful
   Result = UNSUITABLE
Reactivating the Default Selector
Deactivating the Default Selector
   Beginning to apply the Suitable Sponsor
   Application of the Suitable Sponsor was successful
   Result = SUITABLE
Reactivating the Default Selector
Deactivating the Default Selector
Beginning to apply the Default Sponsor
Application of the Default Sponsor was successful
Result = INDIFFERENT
Reactivating the Default Selector
Application of the Default Selector was successful
Result = #.($# DesignMethod (SN[0.0X:.1j3.ZQ9 . 1470])
Reactivating the goal to design the Reboiler
Deactivating the DesignArtifact goal
Beginning to apply the PVCM DesignMethod
Deactivating the PVCM DesignMethod
New goal: ProposeCandidateDesign
Deactivating the ProposeCandidateDesign goal
Looking for methods matching (MetaMethodType
Proposer) (Artifact Reboiler) (DesignMode DESIGN)

The system is now looking for methods to propose a reboiler design. Unfortunately, there are none in the knowledge base. The system generates a Resolve Failure goal to handle the failure.

retrieve relevant methods has Failed because
no relevant knowledge available
Reactivating the goal to ProposeCandidateDesign
Deactivating the ProposeCandidateDesign goal
New goal: ResolveFailure
Deactivating the ResolveFailure goal
Looking for methods matching OR ((Name
AcquireProposerAndRetry) (MethodType
FailureMethod)) ((Name SuccessfulStub) (MethodType
FailureMethod))
Retrieve successful Methods = (SuccessfulStub
FailureMethod) (AcquireProposerAndRetry
FailureMethod)
Reactivating the goal to ResolveFailure
Deactivating the ResolveFailure goal
Beginning to select an appropriate method
Deactivating the Default Selector
Beginning to apply the SuccessfulStub Sponsor

At this point, there are two courses of action available to the user to resolve the failure. The first is to simply ignore the problem for now and proceed with the design as if a candidate design had been proposed. This technique is useful for “stubbing” pieces of the design problem that the user does not wish to address at this time. The sponsor for this method presents a menu to the user to determine if this is a desirable course of action. The menu is presented in Figure 71. In this case the user accepts this option.

Application of the SuccessfulStub Sponsor was
successful Result = PERFECT
Reactivating the Default Selector
Deactivating the Default Selector
Beginning to apply the AcquireMethod Sponsor

A second course of action is to add a new design proposer for this situation. The user declines this option, and this is reflected in the suitability assigned to the AcquireProposerAndRetry method by its sponsor...

Application of the AcquireMethod Sponsor was successful Result = RULE-OUT
Reactivating the Default Selector
Application of the Default Selector was successful
Result = 
#$ DesignMethod (SW[0.0X:.1j3.ZQ9 . 1485])
Reactivating the goal to ResolveFailure
Deactivating the ResolveFailure goal
Beginning to apply the SuccessfulStub FailureMethod
Application of the SuccessfulStub FailureMethod was successful
Reactivating the goal to ResolveFailure
The ResolveFailure goal succeeded
Reactivating the goal to ProposeCandidateDesign
The ProposeCandidateDesign goal succeeded
Reactivating the PVCM DesignMethod

The design has been stubbed, and processing again continues.

Deactivating the PVCM DesignMethod
New goal: VerifyDesign
Deactivating the VerifyDesign goal
Looking for methods matching (MetaMethodType Verifier) (Artifact Reboiler)
Retrieve successful Methods = (AskUser DesignMethod)
Reactivating the goal to VerifyDesign
Deactivating the VerifyDesign goal
Beginning to select an appropriate method
Deactivating the Verify Selector
Beginning to apply the AskUser Sponsor
Application of the AskUser Sponsor was successful
Result = PERFECT
Reactivating the Verify Selector
Application of the Verify Selector was successful
Result = 
#.($& DesignMethod (SN[0.0X:.1j3.2Q9 . 1741))
Reactivating the goal to VerifyDesign
Deactivating the VerifyDesign goal
Beginning to apply the AskUser DesignMethod

Again, no verification knowledge is available. The user accepts the design stub and the design proceeds.

Application of the AskUser DesignMethod was successful
Reactivating the goal to VerifyDesign
The VerifyDesign goal succeeded
Reactivating the PVCM DesignMethod
Application of the PVCM DesignMethod was successful
Reactivating the goal to design the Reboiler
The design the Reboiler goal succeeded
Reactivating the goal to PursueMethodsInParallel
Deactivating the PursueMethodsInParallel goal

This is the fourth subproblem of the distillation column. Again, the knowledge base is canvassed for appropriate design strategies, and again the PVCM method is selected.
New goal: design the Condenser
Deactivating the DesignArtifact goal
Beginning to Initialize the Condenser object
Reporting on the Condenser
Initialization successful
Reactivating the goal to design the Condenser
Deactivating the DesignArtifact goal
Looking for methods matching (MetaMethodType
DesignStrategy) (Artifact Condenser) (DesignMode DESIGN)
Retrieve successful Methods = (Trade Study DesignMethod)
(PVCM DesignMethod) (KRITIK DesignMethod)
Reactivating the goal to design the Condenser
Deactivating the DesignArtifact goal
Beginning to select an appropriate method
Deactivating the Default Selector
Beginning to apply the Bad method Sponsor
Application of the Bad method Sponsor was successful
Result = UNSUITABLE
Reactivating the Default Selector
Deactivating the Default Selector
Beginning to apply the Suitable Sponsor
Application of the Suitable Sponsor was successful
Result = SUITABLE
Reactivating the Default Selector
Deactivating the Default Selector
Beginning to apply the Default Sponsor
Application of the Default Sponsor was successful
Result = INDIFFERENT
Reactivating the Default Selector
Application of the Default Selector was successful
Result = #$ DesignMethod (SN[D.OX:.1j3.ZQ9 . 1470])
Reactivating the goal to design the Condenser
Deactivating the DesignArtifact goal
Beginning to apply the PVCM DesignMethod
Deactivating the PVCM DesignMethod
New goal: ProposeCandidateDesign
Deactivating the ProposeCandidateDesign goal
Looking for methods matching (MetaMethodType
Proposer) (Artifact Condenser) (DesignMode DESIGN)

Looking for a condenser proposer. This time, a single case is available in the knowledge base. It is retrieved, and, even with its INDIFFERENT rating, the case is selected.

Retrieve successful Methods = (Condenser-2047 Case)
Reactivating the goal to ProposeCandidateDesign
Deactivating the ProposeCandidateDesign goal
Beginning to select an appropriate method
Deactivating the Proposer Selector
Beginning to apply the DefaultProposer Sponsor
Figure 74: Verifying the result of design

Application of the DefaultProposer Sponsor was successful Result = INDIFFERENT
Reactivating the Proposer Selector
Application of the Proposer Selector was successful Result = #$ Case (SN[0.0X::1j3.ZQ9 . 1689])
Reactivating the goal to ProposeCandidateDesign
Deactivating the ProposeCandidateDesign goal
Beginning to apply the Condenser-2047 Case
Deactivating the Condenser-2047 Case
New goal: DesignAdaptation
The DesignAdaptation goal succeeded
Reactivating the Condenser-2047 Case
Application of the Condenser-2047 Case was successful
Reactivating the goal to ProposeCandidateDesign
The ProposeCandidateDesign goal succeeded
Reactivating the PVCM DesignMethod
Deactivating the PVCM DesignMethod
New goal: VerifyDesign
Deactivating the VerifyDesign goal
Looking for methods matching (MetaMethodType (Artifact Condenser))
Retrieve successful Methods = (AskUser DesignMethod)
Reactivating the goal to VerifyDesign
Deactivating the VerifyDesign goal
Beginning to select an appropriate method
Deactivating the Verify Selector
Beginning to apply the AskUser Sponsor

Again, the user is prompted to accept or reject the design. The user accepts.

Application of the AskUser Sponsor was successful Result = PERFECT
Reactivating the Verify Selector
Application of the Verify Selector was successful
What is the maximum flow rate of the Reboiler-2056 in lbmole/hr? default = 500

Figure 75: Determining the reboiler throughput

Result =
#$& DesignMethod (SN[0.0X:.1j3.2Q9 . 1741])
Reactivating the goal to VerifyDesign
Deactivating the VerifyDesign goal
Beginning to apply the AskUser DesignMethod
Application of the AskUser DesignMethod was successful
Reactivating the goal to VerifyDesign
The VerifyDesign goal succeeded
Reactivating the PVCM DesignMethod
Application of the PVCM DesignMethod was successful
Reactivating the goal to design the Condenser
The design the Condenser goal succeeded
Reactivating the goal to PursueMethodsInParallel
pursue the PursueMethodsInParallel goal has been Satisfied because the goal agenda was completed without failure
Reactivating the SimpleDistillationColumn Decomposition

All of the components of the distillation column have been (more or less) completed successfully. The individual designs are now pieced together into the resulting column design.

Deactivating the SimpleDistillationColumn Decomposition
Beginning to apply the MaxThroughput SolutionMapper
Fetching the Condenser MaxThroughput = 500

The reboiler design was stubbed, thus no attributes were set. At this point, the recomposition knowledge contained in the solution mappers requests the design database for one of the reboiler attributes, and not finding any, prompts the user for
the appropriate information. The menu presented to the user is shown in Figure 75. The user enters a value of 500, and again problem solving continues...

- Fetching the Reboiler MaxThroughput = 500
- Fetching the StrippingSection MaxThroughput = 500
- Fetching the EnrichingSection MaxThroughput = 500
- Storing 500 as the MaxThroughput of DistillationColumn
- Application of the MaxThroughput SolutionMapper was successful Result = 500
- Reactivating the SimpleDistillationColumn Decomposition
- Deactivating the SimpleDistillationColumn Decomposition
- Beginning to apply the TotalNumberOfTrays SolutionMapper
- Fetching the SimulatedDistillationColumn NumberOfTrays = 10
- Storing 10 as the NumberOfTrays of DistillationColumn
- Application of the TotalNumberOfTrays SolutionMapper was successful Result = 10
- Reactivating the SimpleDistillationColumn Decomposition
- Deactivating the SimpleDistillationColumn Decomposition
- Beginning to apply the EstimatedCost SolutionMapper
- Fetching the DistillationColumn NumberOfTrays = 10
- Storing 6500.0 as the Cost of DistillationColumn
- Application of the EstimatedCost SolutionMapper was successful Result = 6500.0
- Reactivating the SimpleDistillationColumn Decomposition
- Deactivating the SimpleDistillationColumn Decomposition
- Beginning to apply the ThroughputCheck Constraint
- Fetching the DistillationColumn MaxThroughput = 500
- Fetching the DistillationColumn FeedFlowRate = 350
- Application of the ThroughputCheck Constraint was successful Result = T
- Reactivating the SimpleDistillationColumn Decomposition
- Application of the SimpleDistillationColumn Decomposition was successful
- Reactivating the goal to ProposeCandidateDesign
- The ProposeCandidateDesign goal succeeded
- Reactivating the PVCM DesignMethod

The candidate design is finally complete. Now the system attempts to verify the candidate for correctness...

- Deactivating the PVCM DesignMethod
- New goal: VerifyDesign
- Deactivating the VerifyDesign goal
- Looking for methods matching (MetaMethodType Verifier)
  (Artifact DistillationColumn)

This time, several design verifiers are available in addition to the default AskUser method. The suitability of the verifiers is ambiguous, but notice how the suitability of the AskUser method is reduced in the presence of other method. (Up until this
point, the AskUser method has always been rated PERFECT. Now it is reduced to UNSUITABLE.

Retrieve successful Methods = (AskUser DesignMethod) (ColumnCost VerificationPlan) (ColumnConsistency VerificationPlan)
Reactivating the goal to VerifyDesign
Deactivating the VerifyDesign goal
Beginning to select an appropriate method
Deactivating the Verify Selector
Beginning to apply the AskUser Sponsor
Application of the AskUser Sponsor was successful Result = UNSUITABLE
Reactivating the Verify Selector
Deactivating the Verify Selector
Beginning to apply the Default Sponsor
Application of the Default Sponsor was successful Result = INDIFFERENT
Reactivating the Verify Selector
Deactivating the Verify Selector
Beginning to apply the Default Sponsor
Application of the Default Sponsor was successful Result = INDIFFERENT
Reactivating the Verify Selector
Application of the Verify Selector was successful Result = #.($& Plan (SN[0.0X:.ij3.ZQ9 . 1486))
Reactivating the goal to VerifyDesign
Deactivating the VerifyDesign goal
Beginning to apply the ColumnCost VerificationPlan
Deactivating the ColumnCost VerificationPlan
Beginning to apply the ColumnCost Constraint
Fetching the DistillationColumn Cost = 6500.0
Application of the ColumnCost Constraint was successful Result = T
Reactivating the ColumnCost VerificationPlan
Deactivating the ColumnCost VerificationPlan
Beginning to apply the ColumnOperation Simulation
Deactivating the ColumnOperation Simulation
Beginning to apply the SetMaxThroughput DesignStep
Fetching the DistillationColumn MaxThroughput = 500
Storing 500 as the MaxThroughput of CostSimulation
Application of the SetMaxThroughput DesignStep was successful Result = 500
Reactivating the ColumnOperation Simulation
Deactivating the ColumnOperation Simulation
Beginning to apply the TotalCost DesignStep
Fetching the CostSimulation MaxThroughput = 500
Fetching the CostSimulation EstimatedEfficiency = 35
Storing 82500.0 as the TotalCost of CostSimulation
Application of the TotalCost DesignStep was successful
Result = 82500.0
Reactivating the ColumnOperation Simulation
Application of the ColumnOperation Simulation was successful
Reactivating the ColumnCost VerificationPlan
Deactivating the ColumnCost VerificationPlan
Beginning to apply the ColumnOperationCost Constraint
Fetching the CostSimulation TotalCost = 82500.0

One of the method checks the total cost of the distillation column, based on the results of a simple simulation. The costs seem to be excessive, so again the user is interrupted for information. The interaction is guided by the menu shown in Figure 76. Had the cost been below the threshold indicated in the ColumnOperationCost constraint, no interaction would have been necessary. This situation illustrates how the system can be programmed to handle garden variety design situations automatically, and provide a graduated support for progressively more difficult problems. In this case, the cost is accepted, and design again continues.

Application of the ColumnOperationCost Constraint was successful Result = T
Reactivating the ColumnCost VerificationPlan
Reporting on the CostSimulation
Application of the ColumnCost VerificationPlan was successful
Reactivating the goal to VerifyDesign

Another round of verification. The AskUser method continues to be rated lowly, and since the ColumnCost VerificationPlan has been completed successfully, it is ruled out from further consideration.

Deactivating the VerifyDesign goal
Beginning to select an appropriate method
Deactivating the Verify Selector
Beginning to apply the AskUser Sponsor
Application of the AskUser Sponsor was successful Result = UNSUITABLE
Reactivating the Verify Selector
Deactivating the Verify Selector
Beginning to apply the Default Sponsor
Application of the Default Sponsor was successful Result = RULE-OUT
Reactivating the Verify Selector
Deactivating the Verify Selector
Beginning to apply the Default Sponsor
Application of the Default Sponsor was successful Result = INDIFFERENT
Reactivating the Verify Selector
Application of the Verify Selector was successful Result = 
Reactivating the goal to VerifyDesign
Deactivating the VerifyDesign goal

Now it's the ColumnConsistency Plan's turn.

Beginning to apply the ColumnConsistency VerificationPlan
Deactivating the ColumnConsistency VerificationPlan
Beginning to apply the Spacing Constraint
Fetching the SieveTray Diameter = 4.375
Fetching the SieveTray Spacing = 18.0
Application of the Spacing Constraint was successful Result = T
Reactivating the ColumnConsistency VerificationPlan
Deactivating the ColumnConsistency VerificationPlan
Beginning to apply the ConvergedActiveArea Constraint
Fetching the SieveTray ActiveArea = 12.01006
Fetching the SieveTray PreviousActiveArea = 12.01006
Application of the ConvergedActiveArea Constraint was successful Result = T
Reactivating the ColumnConsistency VerificationPlan
Deactivating the ColumnConsistency VerificationPlan
Beginning to apply the ThroughputCheck Constraint
Fetching the DistillationColumn MaxThroughput = 500
Fetching the DistillationColumn FeedFlowRate = 350
Application of the ThroughputCheck Constraint was successful Result = T
Reactivating the ColumnConsistency VerificationPlan
Application of the ColumnConsistency VerificationPlan was successful
Reactivating the goal to VerifyDesign
Deactivating the VerifyDesign goal
Beginning to select an appropriate method
Deactivating the Verify Selector
Beginning to apply the AskUser Sponsor
Application of the AskUser Sponsor was successful Result = UNSUITABLE
Reactivating the Verify Selector
Deactivating the Verify Selector
  Beginning to apply the Default Sponsor
  Application of the Default Sponsor was successful Result = RULE-OUT
Reactivating the Verify Selector
Deactivating the Verify Selector
  Beginning to apply the Default Sponsor
  Application of the Default Sponsor was successful Result = RULE-OUT
Reactivating the Verify Selector
Failed to (select an appropriate method)
Reactivating the goal to VerifyDesign

Everything is OK. There are no more verification methods available (except for the ubiquitous AskUser), and no methods ruled the candidate design out, so the verification task completes with success. At first, the failure to select a method looks bad, but the Resolve Failure goal figures everything out, and the verification task finally terminates properly.

Deactivating the VerifyDesign goal
New goal: ResolveFailure
Deactivating the ResolveFailure goal
  Looking for methods matching OR ((Name SelectionFailureOK) (MethodType FailureMethod))
  Retrieve successful Methods = (SelectionFailureOK FailureMethod)
Reactivating the goal to ResolveFailure
Deactivating the ResolveFailure goal
  Beginning to select an appropriate method
Deactivating the Default Selector
  Beginning to apply the Suitable Sponsor
  Application of the Suitable Sponsor was successful Result = SUITABLE
Reactivating the Default Selector
  Application of the Default Selector was successful Result = #.(S& DesignMethod (SW[0.OX:.1j3.ZQ9 . 1483))
Reactivating the goal to ResolveFailure
Deactivating the ResolveFailure goal
  Beginning to apply the SelectionFailureOK FailureMethod
  Application of the SelectionFailureOK FailureMethod was successful
Reactivating the goal to ResolveFailure
The ResolveFailure goal succeeded
Reactivating the goal to VerifyDesign
The VerifyDesign goal succeeded
Reactivating the PVCM DesignMethod
Application of the PVCM DesignMethod was successful
Reactivating the goal to design a DistillationColumn
The design a DistillationColumn goal succeeded
BIBLIOGRAPHY


314


