AN ABSTRACT OF THE THESIS OF

Brian D. McGinnis for the degree of Master of Science in Mechanical Engineering presented on June 7, 1990.

Title: An Object Oriented Representation for Mechanical Design Based on Constraints

Abstract approved: Redacted for Privacy
David G. Ullman

A representation for the process of mechanical design, along with its computer implementation is presented and discussed. The representation consists of three fundamental concepts: design objects, constraints, and decisions. The design objects are the structures with which the physical artifacts of the design are described. A design object consists of a set of attributes that represent the properties and characteristics of that object. The values of these attributes are specified by the constraints of the design.

The constraints specify the values of and relations between the attributes of the design object. The conglomeration of design objects and their respective constraints define the state of the design. The state of the design can be thought of as a snapshot of the design taken at any particular time in the design process. Changes to this state occur through the introduction of new constraints into the design space.

New constraints are brought into the design by the application of a design decision. The design decision process consists of a set of input constraints, an evaluation performed on those input constraints and the subsequent generation of one or more new resulting constraints. These new constraints in turn affect the attribute values and relations of the design objects, thereby changing the state of the design.

This representation is capable of storing not only the final state of a design, but the initial and intermediate states as well. Maintained also, is the process of change from one design state to the next. By using this representation, one can inspect the evolution of design objects, the propagation and dependencies of the constraints, and the rationale behind the decisions of the design.

This representation was developed from data extracted from mechanical design protocols. These protocols were of mechanical designers solving original design problems and consisted of video recordings, verbal transcripts, and the designer's original drawings.

The representation was implemented in HyperClass, an object oriented programming environment. The implementation is capable of generating and displaying graphical images of the design. The design information extracted from the protocols can be recorded by the implementation developed.

AN OBJECT ORIENTED REPRESENTATION FOR MECHANICAL DESIGN BASED ON CONSTRAINTS

by

Brian D. McGinnis

A THESIS

submitted to

Oregon State University

in partial fulfillment of
requirements for the
degree of

Master of Science

Completed June 7, 1990

Commencement June 1991

APPROVED:

Redacted for Privacy

Associate Professor of Mechanical Engineering in charge of major

Redacted for Privacy

Head of Department of Mechanical Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented June 7, 1990

Typed by researcher for Brian D. McGinnis

ACKNOWLEDGEMENT

I wish to thank Dr. David Ullman and Dr. Tom Diettrich for their many hours of thought, debate, and evaluation throughout the development of this project. I wish to also thank my fellow graduate students of the Design Process Research Group of Mechanical Engineering at Oregon State University, who participated in the formulation and refinement of this research. I also wish to extend my gratitude to Dr. Robert Young and the other researchers of Schlumberger Laboratory for Computer Science in Austin Texas, for granting me the opportunity to work in a friendly and conducive environment.

This research was made possible by grants from Schlumberger Laboratory for Computer Science and the National Science Foundation, grant DMC-8514949.

TABLE OF CONTENTS

Title	Page
1 INTRODUCTION	1
2 DESIGN HISTORY TOOL	6
2.1 Design History Tool Concept	6
2.2 Related Work	9
3 PROTOCOL ANALYSIS	11
3.1 Research Method	12
3.1.1 Constraint Analysis	12
3.1.2 Design Objects, Features, Constraints and Design Decisions	18
3.1.2.a Design Objects	19
3.1.2.b Design Features	21
3.1.2.c Constraints	24
3.1.2.d Design Decisions	28
3.2 Observations	28
3.2.1 Features Identified	29
3.2.2 Constraint Classification	29
3.2.2.a Constraint Source	30
3.2.2.b Constraint Structure	34
3.2.2.c Level of Abstraction	36
3.2.3 Constraint Mapping	38
3.3 Conclusions of Protocol Analysis	45
4 DESIGN PROCESS REPRESENTATION	47
4.1 Design Process Model	47
4.2 Basic Design Process Representation	49
4.2.1 Design Objects	49
4.2.2 Constraints	51
4.2.3 Design Decisions	52
4.3 Constraint Representation	53
4.3.1 Constraint Source	53
4.3.2 Constraint Role	54
4.3.2.a Numeric Parameter Role	55
4.3.2.b Spatial Role	55
4.3.2.c Function Role	56
4.3.2.d Production Role	57
4.3.2.e Form Role	57
4.3.2.f Status Role	58
4.3.2.g Unclassified Role	58
4.3.3 Constraint Language Representation	59
4.3.3.a Equational Language	59
4.3.3.b Graphical Language	62
4.3.3.c Textual Language	63
4.3.4 Constraint Causality	65

5 IMPLEMENTATION 67
5.1 HyperClass Basics 67
5.2 Design Process Knowledge Base 70
5.2.1 Design Object Implementation 73
5.2.1.a Design Object 73
5.2.1.b Design Primitives 75
5.2.1.c Attribute Slots 79
5.2.2 Decision Implementation 80
5.2.3 Constraint Implementation 82
5.2.3.a Constraint Source 83
5.2.3.b Constraint Role 84
5.2.3.c Constraint Language 90
5.2.4 Constraint Implementation Summary 106

6 CONCLUSIONS 108
6.1 Conclusions of Constraint Analysis 108
6.2 Representation Conclusions 109
6.3 Current Capabilities of Implementation 110
6.4 Future Recommendations and Suggestions 111

7 BIBLIOGRAPHY 113

APPENDICES

I. Original Problem Statement 117
II. Features From Protocol Analysis 119
II.A All Features Observed 119
II.B Feature Attributes 122
III. Implementation Code 123
III.A Knowledge Base Printout 123
III.B Support Files 155
III.B.1 Design-Object.lisp 155
III.B.2 General-Constraints.lisp 161
III.B.3 Graphical-Constraints.lisp 165
III.B.4 Equational-Constraint.lisp 178
III.B.5 Textual-Constraint.lisp 182
III.B.6 New-Functions.lisp 187
III.B.7 Draw-Function.lisp 191
III.B.8 Menu-Interface.lisp 195

LIST OF FIGURES

Figure	Page
1. Design History Tool Breakdown	7
2. CAD Drawing of S6 Flipper Dipper	14
3. S6 Outer Frame Assembly Drawing	16
4. S6 Drawing from Protocol	17
5. S6 Design Object Tree	20
6. Expanded Design Object Tree	22
7. Feature Diagram	23
8. Constraint Source Percentages	33
9. Constraint Structure Percentages	35
10. Level of Abstraction Percentages	37
11. Constraint Map Feature List	39
12. Constraint Map Section	41
13. Constraint Loop of Functional Patching	42
14. Constraint Loop of Component Interfacing	44
15. Design Process Model	48
16. Design Process Basic Representation	49
17. Design Object Representation	50
18. Constraint Roles	54
19. Constraint Languages	60
20. Structured Textual Constraints	63
21. HyperClass Example Frame	68
22. Design History Template	71
23. Design Object Class Frame	73
24. Table Object Instance Frame	75
25. Design Primitives Class Frame	76
26. Slab Class Frame	77
27. Table Height Attribute with Facets	79
28. Decision Class Frame	80
29. Constraint Source Class Frame	83
30. Constraint Role with Subclasses	85
31. Role - Language Relation	86
32. Function Module Instance Frame	88
33. Constraint Language with Subclasses	90

34. Constraint Language Class Frame 91
35. Equality Constraint Instance Frame 94
36. Typical Equations 96
37. Conditional Constraint Instance Frame 99
38. Spatial Orientation Instance Frame 101
39. Surface Restriction Class Frame 103
40. Structured Language Class Frame 104
41. Simple Language Instance Frame 104
42. Object Form Language Instance Frame 105

AN OBJECT ORIENTED REPRESENTATION FOR MECHANICAL DESIGN BASED ON CONSTRAINTS

1. INTRODUCTION

Mechanical design is the process of developing physical objects or systems that fulfill some desired need. In creating these mechanical designs, some means or methods of conveying the design information must be established. The design information consists of two distinct parts: 1) a description of the design states, and 2) the process by which those states were achieved.

The design state refers to a snapshot of the design taken at any specific time during the design process. This temporal description of the design contains all the values and relations characterizing that design. This design characterization includes only the information that has been specified up to a given point in the design process. As the design progresses, the state of the design changes as well. The accumulation of these changes to the design states is what constitutes the process of the design.

To fully relate the information in a design, a design representation must include a detailed description of the final design, the intermediate states, and a description of the procedures used to progress from one design state to the next. A detailed description of the design, at any state, includes information on the dimensions, configuration, functions, production, and any other important properties of the objects of the design that have been specified by the designer. This design information is expressed in term of constraints that define the values and relations of the objects in the design. This constraint-based description includes only information specified by the designer up to that

particular design state.

To fully describe the process of the design, the design representation must contain information on the inputs, rationale and result of each decision made within the design. The inputs to a decision are the previously stated constraints that were considered or evaluated when making that decision. The rationale of a decision refers to what motivated or forced that particular decision to be reached. The decision result is the change in state of the design produced by that decision. This decision result is in the form of a new constraint. Therefore design decisions are the mechanisms by which a design changes from one design state to the next.

The representation of the design states and design process must relate both the physical description of the design and the intent of the original designer(s) during that design effort [Kuffner 89]. The intent of the designer gives insight into how and why a particular design configuration was achieved. Design representation is needed for the purposes of design communication. Design communication refers to the exchange of design information from one party to another. An effective means of design communication is required for the successful completion of the following tasks:

1. Design understanding
2. Design evaluation
3. Design of integrated or related systems
4. Redesign

Each task is explained in the following paragraphs.

Design understanding refers to the perception and comprehension of the design. To understand a design, the important properties of the design must be clearly expressed and presented. Understanding the design is important in the manufacture and production of the design. To manufacture a design, a complete description of the physical objects of the design is required. Valuable

production information can be extracted by examining the function of the objects along with the designer's intent. An efficient and adequate means of design understanding should therefore decrease the production time of a design or at least reduce errors.

Design evaluation is the process of reviewing, inspecting, or appraising the result of a design effort. Evaluation of the design could be performed by associated engineers or by a design manager. To achieve an adequate evaluation of a design, it is necessary to examine not only the final state of the design, but also the major decisions involved in reaching that final state. By identifying problem areas of a design early in the design process, corrections to the design can be made prior to the development of poor design solutions.

Integrated design involves the design of two or more systems or parts that have interacting or related pieces. In such situations, designers working on related problems need to examine each other's work and focus directly on those specific aspects of the design that interact. This examination entails not only the inspection of the physical description of the objects of the design, but also the intended function of those objects. By allowing an efficient means of collaborative design, many designers could work simultaneously on different aspects of a single design problem without interrupting the activities of others.

Redesign is the process of modifying or changing an existing design because of a design failure or in order to meet some new design requirement. Redesign cannot always be performed by the original designer. To perform the process of redesign, a designer should ideally be able to trace the states of the design from the design's initial inception to the finalized form. It would be helpful to a designer if he were able to determine the major decisions

of the design and note their rationale. If a designer is not able to retrieve this design information, the old design may have to be abandoned and a completely new design generated. This new design generation can be costly in terms of time and resources, especially when a modification of the old design would have sufficed.

From the explanations of these four design tasks, it is apparent that an effective and efficient means of design communication is desirable for productive mechanical design. Current forms of mechanical design communication include drawings, notebooks, and prototypes.

Design notebooks are hard to maintain and are often incomplete. Prototyping is expensive and cannot be performed until the design has progressed to a fairly detailed level. Consequently, design drawings have become the predominant form of documenting and communicating design. With the advent of computers, the generation of these drawings is now often achieved through the use of some form of Computer Aided Drafting (CAD) software.

Most CAD systems enable a user to record and annotate designs in the form of finalized design drawings. These drawings relate the basic shape, dimensions, and configuration of the design. Sometimes notes on manufacture and assembly are included. Although these drawings are adequate for expressing the physical properties of the design, they lack the ability to relate the functionality of the design and the process from which the design resulted. Consequently there is a large amount of information that is lost or unaccounted for in these mechanical drawings.

Current CAD systems are not fulfilling or meeting all the needs of the design communication tasks stated above. This deficiency in current CAD systems can be remedied by establishing a new form of design documentation. To achieve better communication of mechanical designs, a more

complete form of design documentation needs to be developed.

This research has been devoted to the issue of developing a representation for mechanical designs that meets the communication and representation requirements stated above. The representation developed is capable of containing not only description of the physical objects of the design, as is accomplished with current CAD systems, but the functionality of those objects as well. The representation also contains information on the process of the design, so that the rationale behind the decisions of the design can be expressed. This representation contains the design information in a structure that is consistent with the way it was generated by the designer. To implement this representation, an object-oriented programming environment was used.

The object-oriented programming environment chosen for this research was the HyperClass system. The accumulation of information within HyperClass is referred to as a knowledge base. The design representation is used as the basis for the embodiment of the knowledge needed for a design history tool.

The following section discusses the concept of a design history tool, what it is, what it does, and why it is needed. The next section explains the protocol analysis used and presents the results of that analysis. This leads into a section on the design process representation developed by this research. Finally, documentation on the implementation of the representation is presented.

2. DESIGN HISTORY TOOL

2.1 Design History Tool Concept

A design history is a means of recording, storing, and reviewing the important information generated during the process of designing a mechanical component or system. The design is recorded in a representation so that, not only can the initial and final states of the design be reviewed, but also all the important intermediate states as well. Therefore the evolution of a design system, an individual component of that system, or an attribute of a component can be traced from the initial design specifications to its final, manufacturable form. Maintained in the design history knowledge base are all the major decisions made throughout the design process. Included also is information on proposals for the design that were rejected by the designer as well as the reasons behind their rejection. Within the knowledge base, it is possible to determine the decision making processes, the constraint dependencies and propagation, and the design object evolution. Therefore, with the aid of a design history all important aspects of the design and its development process can be inspected either during the design or after its completion.

It has been speculated that a design history will greatly aid in the processes of design communication [Brown, 89] [Ullman, 87] [Kuffner, 89]. The design tasks mentioned previously, design understanding, design evaluation, integrated design, and redesign, would all benefit from improved methods of design communication. By providing a tool that supports the direct querying of a design, design communication can be made more expedient and more complete. Design communication is also facilitated by being able to efficiently examine a design from any desired viewpoint. A design history therefore

relates not only the "what" of a final design, but also the "how" and "why" that were involved in reaching that design.

The development of the design history tool was

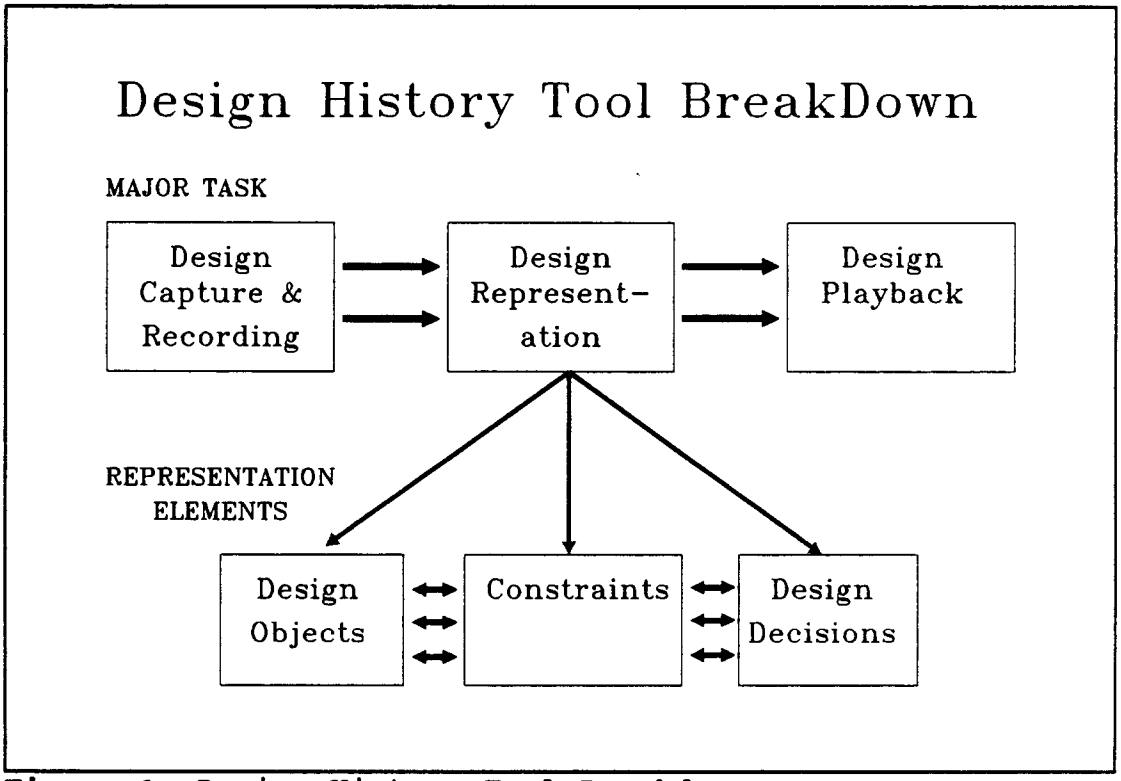

Figure 1 Design History Tool Breakdown

subdivided into three major tasks; design capture, design representation, and design playback, see Figure 1. In terms of the type of information they are dealing with, the development tasks are interdependent, since information flows from one task into the next. In terms of implementation, the tasks are considered independent, (i.e., the representation implementation is not concerned with the capture or playback method used).

Design capture refers to the means or methods used to input design information into the design history. Ideally this information would be captured from the designer, as

he designed at a customized design workstation [Hwang 1990] [Queisser 89] [Lakin 89]. Unfortunately this capture of design information is a non-trivial task, and in itself requires much research and work to achieve. For the purposes of this research, the design capture was achieved by using a simple design recorder. The design recorder achieved the task of recording design information through the use of a user interface, which facilitated the entering of design data, that was extracted from verbalized protocols.

The development of a design process representation is the major focus of research discussed in this thesis. To accomplish the task of developing a mechanical design representation, a model for the process of mechanical design was established. To generate an accurate and complete model, working mechanical designers were observed and studied. The means of studying the mechanical designer was through protocol analysis. This analysis was used as the basis in the development of the representation for the design process. The protocol analysis used in this research and a discussion of the results can be found in Section 3. The formal design process representation developed is presented in Section 4.

The playback of the recorded design information is achieved through the use of a design playback tool, which is being developed in a related research effort. The purpose of this tool is to retrieve and present the design information in a form and at a level that can be easily understood by mechanical designers. This playback tool allows a designer to sit at a computer workstation and examine a stored design at its final states or at any of its intermediate states of development. The playback tool also presents information on the decisions in the design: what forced them, why they were made, and what was their result.

2.2 Related Work

There are other efforts currently underway to develop tools that are similar to the design history. Primary among these is the NASA-funded effort to record the important aspects of the design of part of the Space Infrared Telescope facility [Leifer, 87]. Part of this effort is the VMACS program [Lakin, 89]. VMACS is a prototype electronic design notebook. It is developed to support conceptual design by giving designers the freedom and agility they find with pencil and paper. It records and maintains all efforts performed by designers during the conceptual design: sketches, calculations, and design notes. VMACS does not record or represent the rationale behind design decisions, nor does it capture the constraint dependencies or propagation. In order to obtain this information, the development of Design Rationale Inferencing System is underway. This system is used to infer the rationale and monitor the satisfaction of constraints.

Also of interest is the gIBIS effort at the Microelectronic and Computer Technology Corporation MCC [Conklin, 1988]. The gIBIS (graphical Issue-Based Information System) is an on-line hypertext facility for interactively posting the issues, positions, and arguments for a problem being solved by many users. Although not intended to record a temporal history of the posting of positions on an issue and arguments for the positions, it does make extensive use of hypertext in a problem solving situation.

The MIKROPLIS system [McCall, 1989], which is similar to gIBIS, extends the issue, position, and argument structure by allowing sub-issues, sub-positions, sub-arguments and so on. Therefore MIKROPLIS can be considered a 3-dimensional text outliner with retrieval

capabilities specifically aimed for use in design problems. Again, although it records no temporal history of the design process, it is a good example of the use of a hypermedia system as a design aid.

Three points differentiate the research described in this thesis from those mentioned above:

1) The design history tool has been developed from the bottom up. Data extracted from the video taping of designers solving problems was used to generate the model employed in the design history representation. The others are prescriptive models or methods developed from the researcher's knowledge.

2) The design history tool implementation is based on HyperClass and incorporates the solid modelling package, Vantage [Balakumar, 1988]. This implementation allows the objects of the design to be graphically represented and directly queried by the user's mouse interactions. Although VMACS does allow for graphic images, it does not provide an interactive query system that displays the temporal design information. Neither of the other systems mentioned currently support graphic images of the design.

3) The design history records not only the states of the design but the temporal and dependency information as well. In this way a chronological history of the design is documented so that the evolution of the design can be inspected. The other research efforts mentioned are concerned with only the current states of the design and have no easy means of accessing previous design states.

3. PROTOCOL ANALYSIS

There have been many postulations and theories on how the design process of a mechanical part proceeds from initial specifications to completed artifact. Most of these design theories [Jones, 1970] [VDI-221 87] [Paul & Bietz 88] [Hubka 84] are prescriptive techniques rather than descriptive models derived from empirical data. A design method derived from empirical data can be beneficial in many areas, such as design instruction, design automation, and in the advance of computer design tools. An empirically based design method should be more natural and instinctive for a designer to use. It can be speculated that by empirically basing computer design tools, the acceptability and usefulness of the tools increases, since these tools are easier to incorporate into the designer's normal design activities.

The general purpose of the ongoing research at Oregon State University has been to study the design process and extract pertinent empirical data. This data can then be used to determine a satisfactory design model that can emulate the actual process employed by design engineers. The research at Oregon State University has been devoted to the design process of the mechanical engineer. For the formulation of the representation, one part of this research included the tracking of the constraints (see Section 3.1.2.c) in the design of a single part and determining the constraint use, growth, and evolution. This research used empirical data as a basis to define the use of features (see Section 3.1.2.b) in a design and how features and constraints interact within the design process.

To better understand the process of constraint development and propagation, the identification and classification of all the constraints that affected a

particular component of a design was performed. This was achieved by reviewing a design protocol transcript while viewing the designer on a video tape [Stauffer et al., 1987]. Therefore the constraints captured consist of only explicitly stated constraints that were either spoken or drawn by the designer. These captured constraints were then classified into different categories and displayed on a constraint map to observe patterns and regularities. Further reduction was then performed on the constraint statements to arrive at individual feature relationships, which were used to aid the feature analysis.

Before describing the terminology employed, it is important to explain the method of protocol analysis and relate the background of the component studied. Therefore the following sections are devoted to describing the methodology of the research performed. This is followed by the terminology, which will present the definition of a constraint and a feature and discuss the way in which they interact. This leads into a description of the classification used in the protocol analysis, which is followed by the constraint mapping technique employed and a discussion on observations made from the constraint classification.

3.1 Research Method

Protocol analysis was achieved by assigning an initial set of criteria for the design constraints. Then by reviewing a design protocol, the initial criteria were developed into a complete and adequate constraint classifications set. The following sections explain the method of protocol analysis used and how the constraints were extracted from the design protocol.

3.1.1 Constraint Analysis

To understand the performance of a design engineer, it

was necessary to observe engineers engaged in the design process. This was accomplished in this research through a technique called protocol analysis. Design protocols were obtained by video and audio taping of engineers as they solved specific design problems. During their solutions they were asked to "remain verbal" [Stauffer et. al., 87] [Ullman et. al., 87]. The engineers' verbalizations were then transcribed. By reading the transcripts and viewing the designers gestures and drawing actions on video tape, many types of information can be extracted from the data [Stauffer et. al., 87] [Ullman et. al., 88]. Here, the protocol data was used to extract the features and constraints from the design effort.

Protocol data is the best way known to this researcher to gather information on how humans solve complex problems. Since the major use of the data developed here is for the development of a design representation, it is important to know how human designers behave in order to develop an acceptable model of the design process. However, protocol data is limited to what the designer verbalizes, gestures, or draws. It is conceded that not all the design activity is verbalized in protocol data, because humans think faster than they can talk. But past research [Stauffer et. al., 87] has shown that what is recorded gives enough information on the overall activity to aid in understanding the design process.

In previous studies [Stauffer et. al., 87] [Ullman et. al, 87], a total of five protocols were recorded. Two experienced design engineers solved a problem concerned with the packaging of small batteries in a computer and three design engineers solved the problem used for this study. The design problem presented to the engineer whose protocol is used in this study (the selection criteria used follows this paragraph) was that of coating both sides of an aluminum plate with a thin film. The design

specifications (see Appendix I for the original problem statement) presented to the subject gave the required operation which included a worker loading a 10" x 10" x .063" aluminum plate into the machine to be designed. The machine then must lower the plate to the surface of a water bath. On the water surface was a sticky, thin chemical layer, which adheres to the plate on contact. The plate must then be lifted off the surface, flipped over and the process repeated for the other side. Finally, the worker removes the plate with the film on it and loads a new one to be coated. Based on the action required by the machine, this problem has been dubbed the "flipper-dipper" problem. Along with the functional requirements above, specific form constraints on the handling of the plate and the allowable space for the machine were also given.

Since the focus of this protocol analysis was to track one component through its evolution, it was necessary to

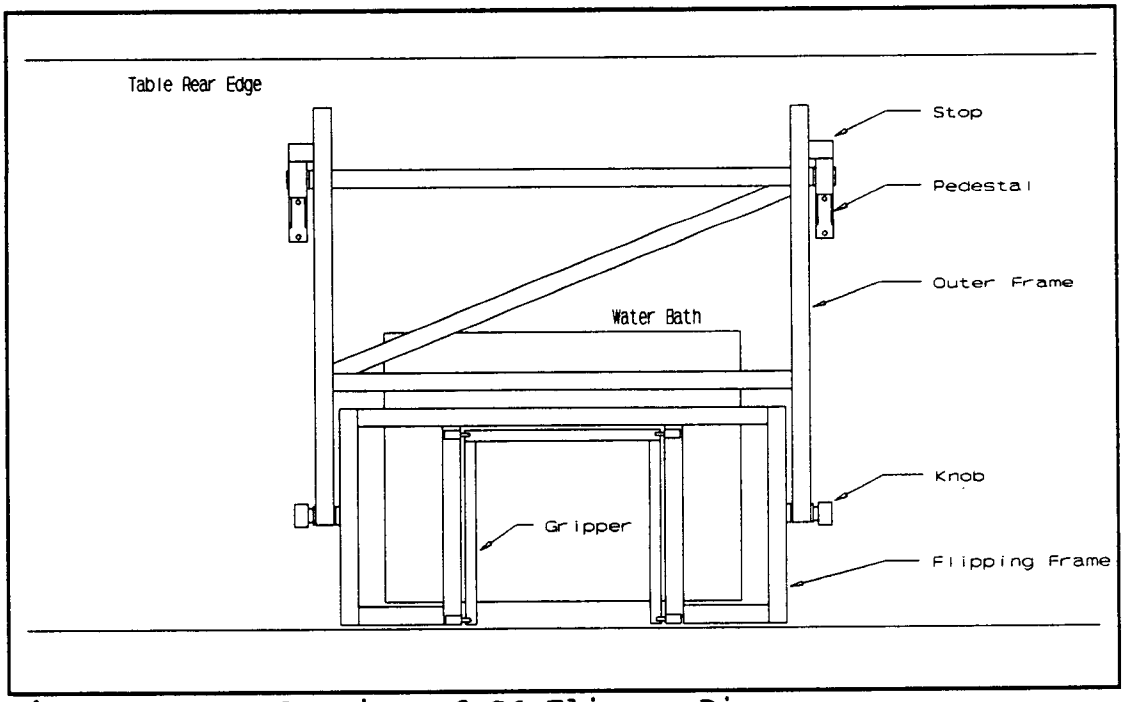

Figure 2 CAD Drawing of S6 Flipper Dipper

select a single subject's protocol for study. The selected protocol was chosen because of the clarity of the protocol data and the relative straightforward nature of the design process and of the finished design itself. The chosen design consists of a pedestal, outer frame, flipping frame, and gripper as is shown in Figure 2. This figure is a cleaned up CAD drawing of the final design drawing generated by the designer. The operation of the mechanism is as follows: the operator places an aluminum plate into slots located on the gripper assembly, the gripper which is attached to the flipping frame is then lowered to the water surface for the chemical adhesion, the plate is then raised above the water bath and rotated about a pivot between the flipping frame and outer frame, and the plate is again lowered with the opposite side down. Once the chemical adhesion has taken place, the plate is again raised and removed from the gripper assembly. To account for the thickness of the flipping frame, the gripper assembly slides on rods so that the lower edge of the plate is always below the lower edge of the flipping frame.

Due to its simplicity, the design object chosen for the study was that of the outer frame assembly of the machine. The outer frame, along with its decomposition into its major components, is shown in Figure 3. The main function of the outer frame assembly is to raise and lower the flipping frame over the water bath. The components that comprise the outer frame are the lever arms, the front cross member, the back cross member, the diagonal cross member, the front and back pivot points, the doublers, and the bushings. Each of these components has certain features that are pertinent in the design. The constraints associated with these individual features are the ones focussed on in this analysis.

The protocol studied encompassed a total design period

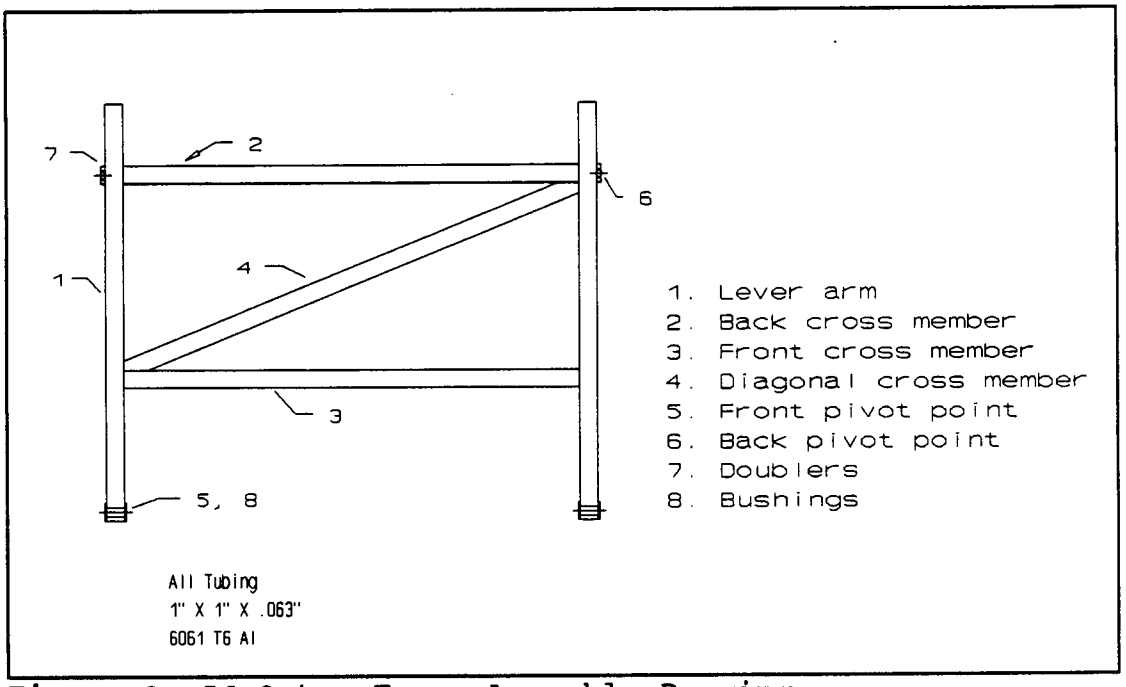

Figure 3 S6 Outer Frame Assembly Drawing

of 4 hours and 38 minutes, recorded over three separate design sessions on subsequent days. The transcript for this protocol consisted of 85 pages of typed, double-spaced verbalizations and 28 separate design drawings made by the subject (all drawings were done by hand). These range from conceptual sketches to complete detailed drawings from which Figures 2 and 3 were taken. The development of the outer frame assembly was contained in 40% of the total transcript and was represented on 34% of the drawings.

As an example of the data, consider the section of transcript presented below, which is about one minute long. In this example the subject is about 1/3 of the way through the design. He is trying to position the front pivot point (the pivot between the outer frame and the flipping frame) and subsequently determine the lever arm length. This topic is reconsidered four other times during the protocol, each time with new design information having been generated in the interim. The numbers

preceding the lines refer to page and line number of the protocol from which the data was extracted. In this section the designer is referencing a sketch he made that

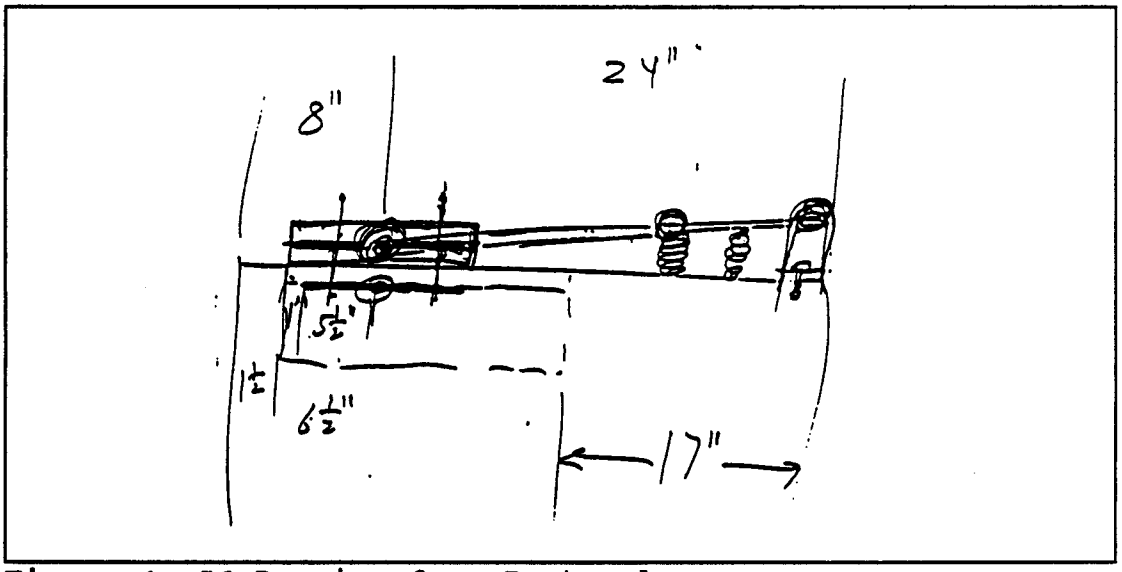

Figure 4 S6 Drawing from Protocol

is shown in Figure 4.

The text, the figure, and the associated video tape for this and other similar segments of the protocol that refer to the outer frame assembly are the raw data on which this study is based. In order to deal with the data it was first reduced to a more intelligible form, that of the individual constraint phrases. This reduction directly follows the segment of transcription shown below.

> "Let's see how much we've got in the back. We've got 32" minus 15, 17" here to play with, so we need, surface of the water is here, 1/2" below and if we make most of this take place near the front of the tank, say 1" from the front of the tank, here's a plate, that'd give us the longest moment arm there, or fulcrum, so we get the maximum elevation pivot point, pivot point is up above the, okay, if we have to we can make it up here. Okay. And the rods here, rod here, and it has to go equal distance past here, so that it slides equal distance from both sides of the pivot point. And the pivot point will also be, on the same plane as the pivot point we'd also have to have

our guide so that it didn't extend below or equal distance on each side of the pivot point, like so, the framework. Okay, here we go. And, that would take care of that arm coming back. Okay."

19.17 we've got .... 17" here to play with [from back of sink to back edge of table] (drawing 4.2)

19.23 most of this [machine process] take place near the front of the tank, 1" from front of tank

19.27 longest moment arm there, or fulcrum

19.27 get the maximum elevation in the [front] pivot point

19.31 [front] pivot point is up above the..... make it up here (drawing 4.2 A)

19.45 that arm coming back (drawing 4.2 B)

It should be noted that not all of the information available in the transcript sample was extracted. Only information relevant to the outer frame was considered. The numbers preceding each constraint phrase refer to the page and line number on the protocol transcript, from where the constraint was extracted.

The important points to note about the above reduced text are that 1) this reduction does capture the design progress of the protocol, 2) it is easier to follow than the original text, and 3) the six statements are all of a similar structure. After examining the constraint statements made on the outer frame assembly, it is hypothesized that all the data can be represented in terms of features and constraints and that design decisions can be observed through the changes occurring in them.

3.1.2 Design Objects, Features, Constraints and Design Decisions

One of the main objectives of this protocol analysis

was to establish an acceptable terminology, that could be used in describing the design process. By following the progression of constraints and the evolution of features in the design protocol, the formulation of this terminology was achieved. Therefore the following terminology was arrived at through considerable review of design data, with the objectives of being completely inclusive and relatively basic so that all data could be easily modeled.

3.1.2.a. Design Objects

During the design effort an engineer decomposes the design into assemblies, components, and features of assemblies, and components. All aggregations of components are termed an "assembly" [Tikerpuu & Ullman, 88]. Assemblies can be thought of as parent or child in nature, in that an assembly can have a larger parent assembly or have children or (sub)assemblies. Since there is no limit to the number of assembly generations, the term "assembly" is used for all component aggregations. "Components" are the individual parts of a design.

A design object tree, which graphically displays the hierarchical breakdown for the design studied, is shown in Figure 5. This diagram shows the breakdown of the entire assembly into its children assemblies. Also depicted is the breakdown of the outer frame assembly into its eight components.

The next section focuses on the features. However, before fully defining features it must be noted that the breakdown of design objects into assemblies, components and interfaces is inadequate in that it is too coarse. Usually it is necessary to deal with the features of a design object or commonly, the features of a feature. For example; the lever arm is a component. However, much design activity was expended on the front pivot point, a

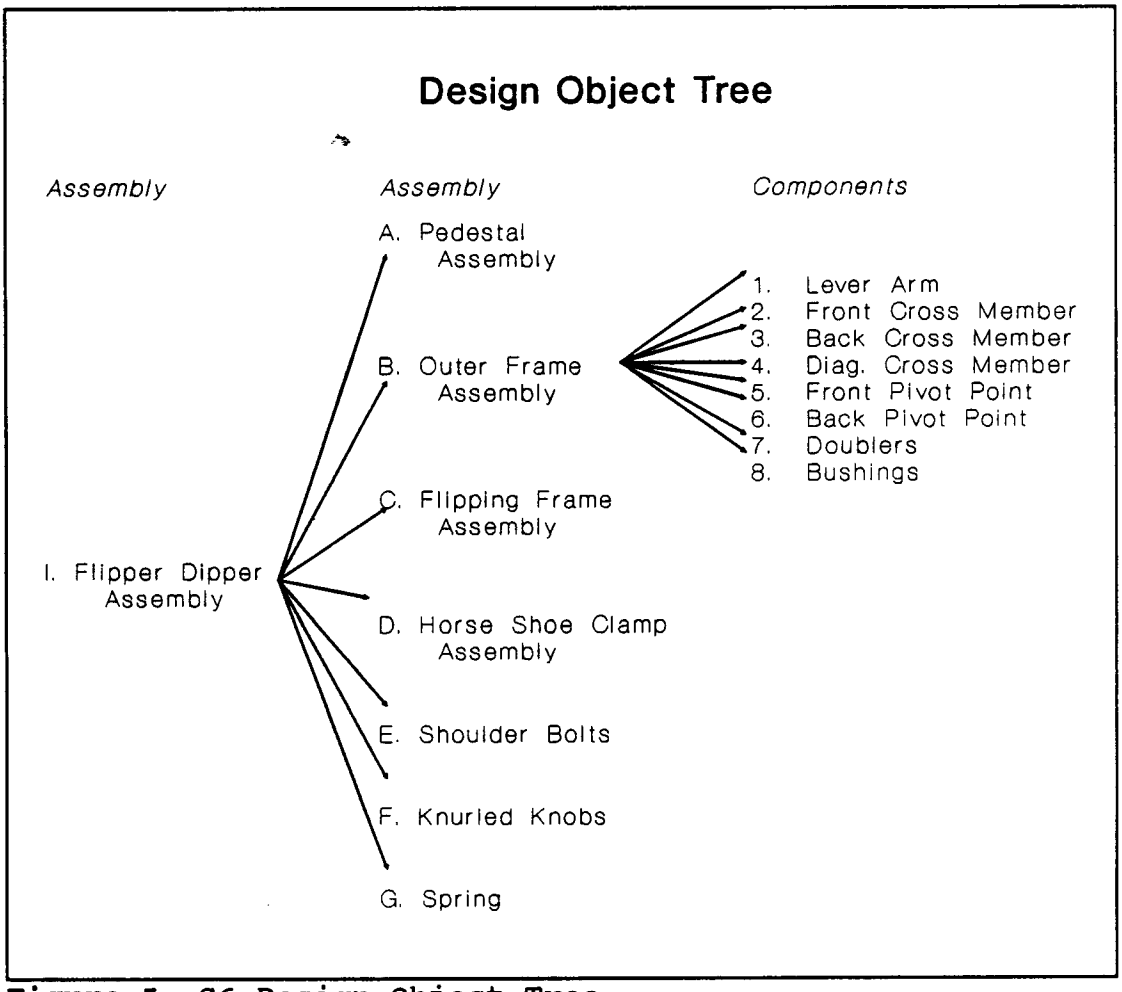

Figure 5 S6 Design Object Tree

feature of the lever arm. The front pivot point, in turn, has many features such as its location. Thus, features themselves must be treated as design objects.

It is important to define all the entities of a design as design objects, in order to discuss them as a group, and because this breakdown is consistent with the way in which a designer refers to the design. At any given moment in the design, the designer focuses his attention on one of the design objects. The distinct properties of the design object that the engineer deals with when working on it are commonly called design features.

3.1.2.b Design Features

There has been a substantial amount of work done in the area of design feature definition and use [Cunningham & Dixon, 88] [Dixon et. al., 88] [Unger & Ray, 88] [Shah & Wilson]. These efforts represent a variety of different backgrounds and differing points of view. This has resulted in a variety of definitions for the term "feature". These definitions all have validity in their respective fields or areas of interest. However, here the concern is for the terminology used by the designer and so the definition must be very broad. Thus, this research's formal definition for a feature is:

> A feature is any particular or specific characteristic of a design object that contains or relates information about that object. It is verbally represented in the form of a noun or noun phrase.

From the above definition it can be surmised that anything and everything about a mechanical design object can be considered a feature, such as location, length, material, operation, etc. Depending on the point of view taken, some features are obviously more important than others. In other words, the relevancy of a feature is dependant on the focus at the time. For example, the manufacturing feature of a surface finish may not be important to a designer during the conceptual phase of the design process, but is important to manufacturing. Likewise some functional features on how a design behaves, which may be critical to a designer, may not be important from a manufacturing point of view. An expanded object tree displaying some of the features of the components used in the design of the outer frame assembly is shown in Figure 6. The features shown in this tree are those that the engineer devoted a considerable amount of time to. A

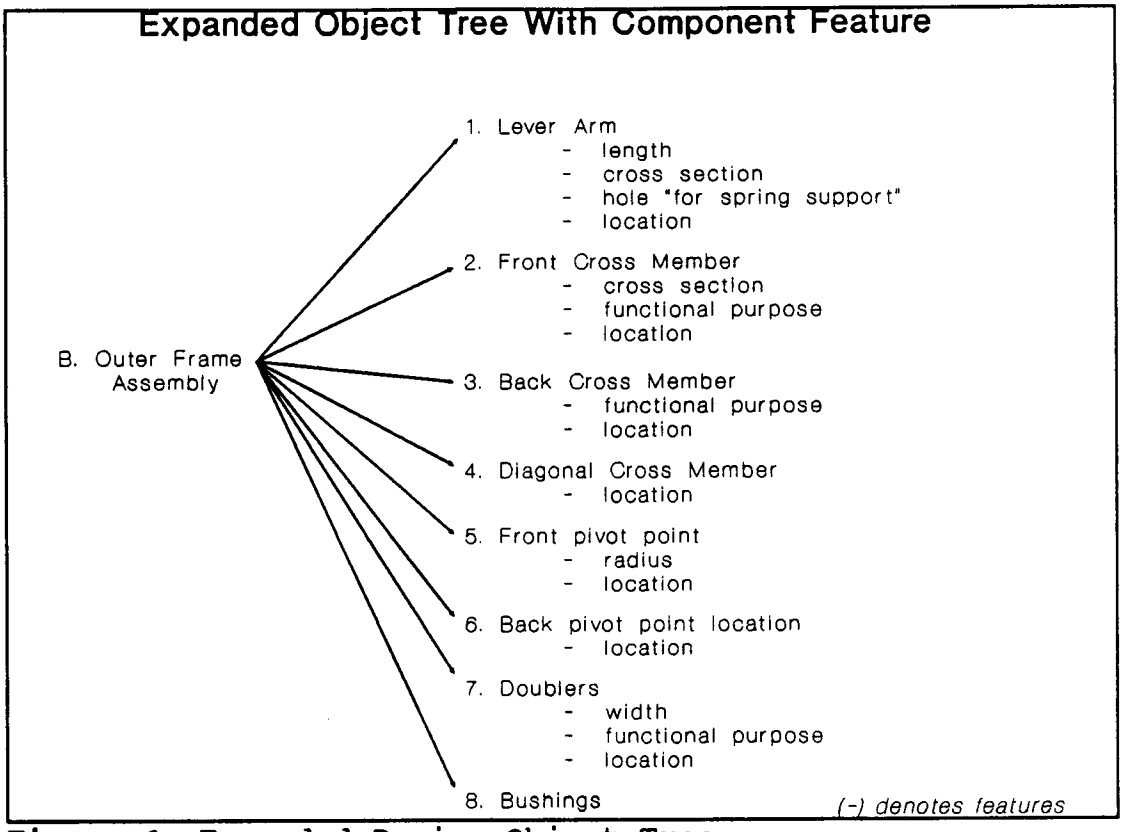

Figure 6 Expanded Design Object Tree

complete list of all 277 features and 38 design objects used by the designer in the sections of the protocol studied are given in Appendix II.A. Out of the 277 features only 95 (34%) of them directly describe the outer frame. The remainder of the features were of components, interfaces or assemblies that acted as constraints on the outer frame.

There are two distinct types of features used by the designer: form features and function features. Form features include geometrical, topological, manufacturing and tolerance features along with any other features used to describe the physical structure of the design object. Functional features include both the function purpose of the design object such as support, stability, or strength and the function behavior that the design object performs like lifting, gripping, or rotating. The form features

define or describe the physical characteristics of design objects in a design, while the functional features explain what purposes the design objects achieve and what behaviors they exhibit in the design. A breakdown of these different feature types can be found in Appendix II.B.

The feature diagram, in Figure 7, displays how the features are segregated into form or function. To better understand the way in which form and function features describe an object, consider the mechanism of the flipper dipper in the design studied. Some important form features include the shapes, dimensions, orientations and manufacturing methods. The functional features of the mechanism are the purpose and behavior, which are described by the constraint phrases "to coat both sides of the plate" and "gripping, lowering, raising, rotating, and so on", respectively.

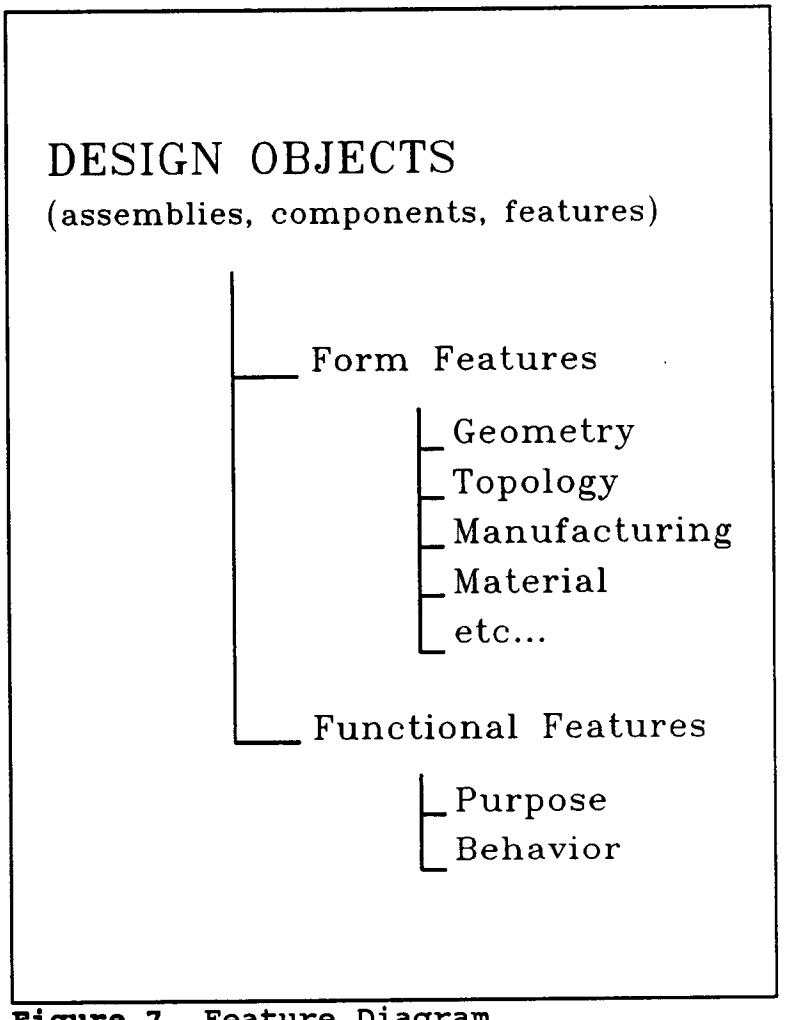

Figure 7 Feature Diagram

3.1.2.c Constraints

In the protocols, constraints are statements made about a design's features, such as: 'the length is 19"', 'the knob diameter is smaller than the outer arm height.' In the first example the feature "length" is constrained by the value "19". The second example relates two features, "the knob diameter" and "the outer arm height" by a conditional value of "smaller". In reviewing the constraint statements like those at the end of Section 2, for all the protocol sections focused on the design of the outer frame assembly, the 277 features were involved in 725 constraint relations. These have been classified into the two types of semantic structures observed. By using these structures every constraint in the protocol has been modeled. Outside of the studied protocol, the constraint types given below are thought to represent all types of constraints possible.

The first semantic structure is of the form [feature-instantiation]. Here the term "feature" is as defined in the previous section and term "instantiation" refers to the value related to the feature. From the protocol sections reviewed, two types of feature relationships of this structure were found. These are:

1. <function feature>-<instantiation>
2. <form feature>-<instantiation>

Below are examples of these two constraint types found in the protocol. Preceding the feature relationship is the number referring to the specific type of the relationship involved. The dashes (-) signify the separation between the feature and instantiation.

[1 machine operation time - is 40 seconds]

Here the feature being instantiated is the functional feature of "machine operation time". It is being

instantiated with the descriptive value "is 40 seconds". This instantiation comes from the given design specifications and is therefore a reference made by the designer as opposed to his changing or giving some new value to the feature.

[2 flipping frame depth - needs to be at least 10 1/2"]

The form feature here "flipping frame depth" is instantiated with the value "at least 10 1/2" ". Here a feature is given an inequality value that must met.

Of the above two examples the first came from the given specifications and the second from the design itself. Many of the instantiations observed involved the initialization or modification of features in the design itself. It is these instantiations that give the current value of the feature. By observing the changes in the value of the feature instantiation, the history of the feature development can be tracked.

For example, the next two occurrences of the flipping frame depth are given below as:

[2 flipping frame depth - is 12"]
[2 flipping frame depth - is 12" overall]

The feature is first given a value of "12" ", which is a refinement of the previous instantiation. The feature value is then further refined by the instantiation "12" overall". This type of feature development is typical of the protocol studied, and it shows how the feature progresses with respect to the constraints.

The other semantic structure observed for the constraints was of the form: [dependent feature-relation-independent feature]. As there are both form and function features and there can be both form and functional relations, there are potentially eight different feature

relationships of this structure possible. These are:

3. <form feature>-<form relation>-<form feature(s)>
4. <form feature>-<function relation>-<form feature(s)>
5. <function feature>-<form relation>-<function feature(s)>
6. <function feature>-<function relation>-<function feature(s)>
7. <form feature>-<form relation>-<function feature(s)>
8. <form feature>-<function relation>-<function feature(s)>
9. <function feature>-<form relation>-<form feature(s)>
10. <function feature>-<function relation>-<form feature(s)>

In each of these, one or more features constrain, through the relationship, a single feature. Examples of each of these constraint types are listed below:

[3 knob location - in middle of - flipping frame outer arm location]

Here the form feature, "knob location", is constrained by the relation, "in middle of", to the form feature "flipping frame outer arm location". This was used as a form constraint on the positioning of the knob.

[4 outer frame orientation - stops when at right angles with - pedestal stand position]

The first form feature, "outer frame orientation", is related functionally by "stops when at right angles" to the second form feature "pedestal stand position". This relation acts as functional constraint on the outer frame.

[5] <function feature>-<form relation>-<function feature>

This relationship was not found in the section of protocol that was codified. This relationship is theoretically correct but cannot be substantiated by the data reviewed.

[6 front cross member purpose - prevent - outer frame racking]

Here the functional feature of "front cross member purpose" is related to another functional feature of "outer frame racking", by the relation of "prevent". Here the functional purpose of preventing outer frame racking was satisfied by the front cross member.

[7 front pivot point elevation - would be 8" to allow for - flipping frame rotation]

Here the form feature, "front pivot point elevation", is subject to the conditional form relation of "would be 8" to allow for" regarding the function feature, "flipping frame rotation". This is used as a form constraint on determining the location of the front pivot point.

[8 back pivot point location - limits - framework side travel]

This relationship shows that the form feature of "back pivot point" has a functional relationship of "limits" on the function feature of "framework side travel".

[9 flipping frame behavior - occurs 1" from -tank front edge]

In this relationship, the functional feature of "flipping frame behavior" is related to the form feature of "tank front edge" by the form relation of "occurs 1" from". Thus the position of the water tank is imposing a form constraint on the flipping frame operation.

[10 spring purpose - hold out of water - flipping frame position]

Here the function feature, "spring purpose", is related to the functional relation of "hold out of water" with respect to the form feature "flipping frame". The

only difference between this relationship and that shown above in (6) is that the independent feature is of a form type.

This breakdown of constraints into basic structures assisted in the formulation of the constraint representation, which will be discussed later. This breakdown also leads to the development of a more formalized representation of design decisions.

3.1.2.d Design Decisions

In this model of the design process, design decisions are defined in terms of constraint changes.

> A design decision is defined as occurring each time a value or relation stated about a feature of a design object is changed or initialized or when a new feature is introduced.

Design decisions may be driven by several different constraints emanating from separate design objects, but the decision itself directly affects only one specific object feature. The result of the design decision can affect any level in the design object tree but, it usually modifies a specific feature value on which the decision was focused. Resulting constraints are the direct consequences of design decisions and represent new information generated in the design. Once a design decision is made, the new information can be used as input constraints in future design decisions.

3.2 Observations

From the aforementioned constraint analysis, several new and interesting observations were made. A list of common features was generated that gave insight into what type of parameters needed to be modelled. A constraint classification was developed that helped categorize the

different aspects of the constraint information and also gave insight on the refinement of the constraints. A constraint mapping technique was developed that indicated how the constraints were propagated within the design. The mapping technique also showed that the design process is non-linear process. The following sections discuss these observations.

3.2.1 Features Identified

In the sections of the protocol that were focused on, the design of the outer frame assembly, 277 separate features were observed. A list of these features is given in Appendix II.A. In reviewing these 277 features it should be noted that they are of the form {design object-specific feature}. Specific features are object attributes of interest. If the design process were totally geometric, then the specific features would all be geometric primitives such as length, height, etc. In the entire 277 features there are only 58 different specific feature attributes. A listing of these feature attributes, categorized by type, can be found in Appendix II.B. These are by no means a complete list of all possible design features, but represent only the features explicitly used by this designer on one subassembly in the design. It is observable from this list that the majority of the feature attributes are form oriented. This is consistent with the results of earlier research [Ullman et.al 1988], namely, designers focus on form much more than function.

3.2.2 Constraint Classification

Once the definitions were completed and the constraints extracted from the design protocol, the task of classifying the constraints was initiated. The purpose of classifying the constraints is to characterize them so

that their effect on the different design features can be observed. By classifying the constraints, a pattern for their propagation and growth can be documented.

The classification of the constraints was performed not only for purposes of analysis but also in hopes of developing a more formal constraint language upon which the representation would be built. Even if the classification only provides an elaborate bookkeeping tool, it would still be beneficial in the development of a constraint management system. The classification also gives insight into how the designer uses constraints in a design and relates the significance of some constraints as opposed to others. Constraints are classified by three different measures: constraint source, constraint structure, and level of abstraction. Below is an explanation of each of the different measures and statistics from the protocol reduction that was taken for the outer frame assembly.

3.2.2.a Constraint Source

A constraint can originate from one of three different sources: it can be given, introduced, or derived. Given constraints are those furnished to the designer, which provide an idea of what is required by the design. A given constraint is usually received by the designer at the beginning of the design as part of the design specifications and requirements. These can be either form or function oriented and can originate from clients, designers of adjacent design objects, or initial specs. Given constraints can be used at any time in the design process, whenever the designer requires information on the initial state of the design.

An introduced constraint is one which is brought into the design from the designer's domain knowledge, handbooks, or other "domain knowledge" sources. It is a

constraint that is brought in from outside the current problem solution space and has not been derived from any other constraints. Introduced constraints often are vague and general by nature. Examples of introduced constraints are "keep everything as light as possible" or "make maximum use of the table". Other introduced constraints include equations and hard data used in analysis such as material properties, but none of these were observed in the single protocol examined.

Derived constraints are generated inside the design space and are intrinsic to the design being worked on. Derived constraints result from and can be changed by the outcome of design decisions. Although design decisions can be affected by given, introduced, or derived constraints, the result of a decision is always the creation or modification of a derived constraint.

The process of design for a mechanical part can be thought of as the creation and refinement of a set of constraints. Initially these constraints consist of a group of given constraints that define a desired function and/or form for a particular application. The given constraints comprise the initial specifications of the design. As the design process proceeds, this body of constraints grows and develops until a final concrete solution is achieved.

In the beginning of the design, the design engineer is presented with given constraints that are primarily functional; they describe the desired operation of the design. Along with the functional definition of the design, very specific form constraints are often presented in the given specifications (e.g., the dimensions of the aluminum plate).

As the designer proceeds, the given constraints become less abstract and develop into concrete form and function instantiations for specific design objects. In the

protocol studied, the engineer took the given specification of "coating both sides of an aluminum plate" and developed it into "a person would load the plate, dip it, rotate it, dip it, and then a person would have to flip it back to unload it". This revision of the functional constraints allows the engineer to begin the conceptualization of the design objects needed for the design function.

The next three paragraphs walk through the three design stages of the design (conceptual, layout, and detail) in reference to the constraint source: given, introduced, or derived. In analyzing the data, the concept of three distinct design stages was adopted: conceptual, layout, and detail. These stages are really management labels given to the level of design detail and are hard to apply, since the design process is so interleaved. While one feature may be in the layout stage, other adjacent components or features of the object in question may still be conceptual. Therefore the design stage divisions assumed in this research were based on the drawing level produced by the designer during the protocol. Rough sketches were treated as conceptual design representations. After establishing a basic layout of the design with rough sketches, the designer used a straight edge to generate scale drawings with rough dimensions. These drawings were taken as representations of layout design. The designer then made drawings with detailed dimensions and manufacturing notes. These were considered as representations of detailed design. These stages are important when trying to understand how the constraints affect the design and how the constraints are propagated through the design. A graph of the constraint source percentage for the three design stages and for the total design is given in Figure 8.

As the design's functional constraints become more

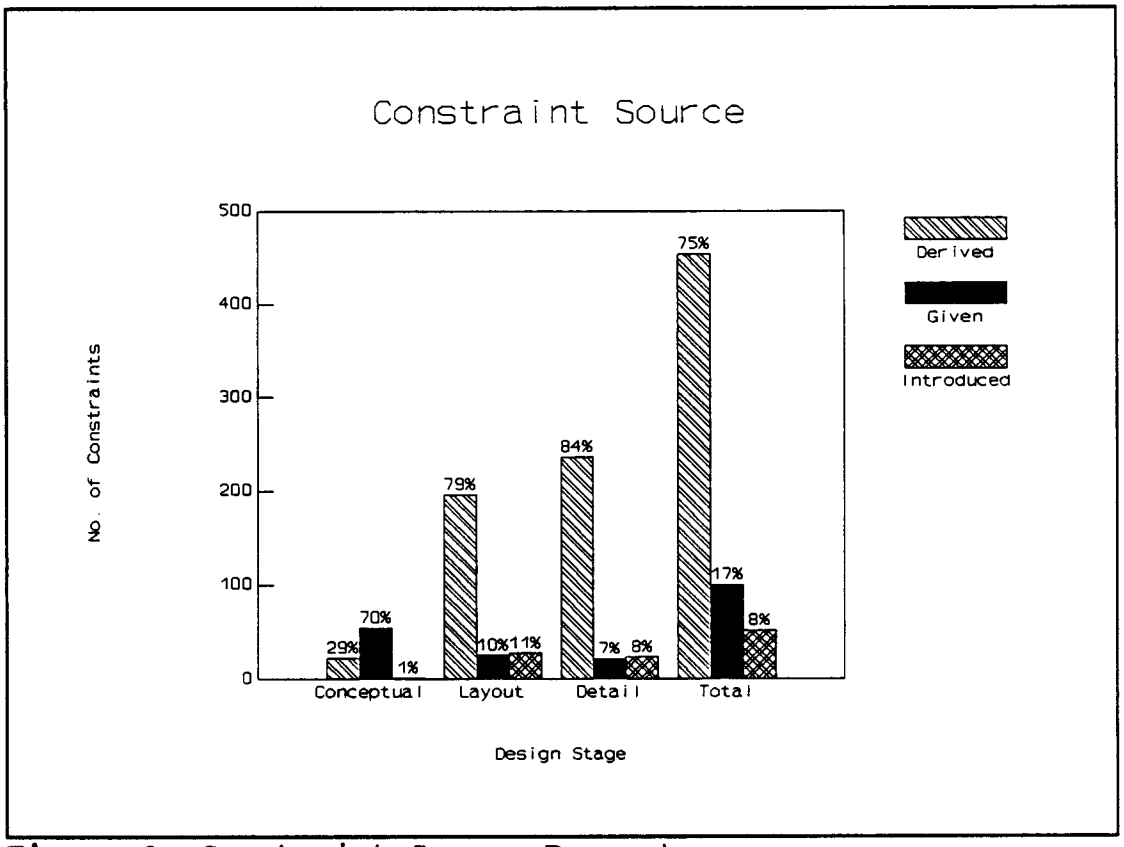

Figure 8 Constraint Source Percentages

concrete, design objects are created to meet these new functional requirements. During the conceptual stage of the design, the majority of the constraints (70%, see Figure 8) are given constraints. From the functional constraints are born the conceptual form of the design. In the development of the conceptual design, form features are created out of functional requirements and then verified by simulation of the design operation.

Once a completed conceptual design is created, the designer begins the layout stage of the design process. In the layout stage, the designer patches and refines his design and checks to see that there are no conflicts with any of the form constraints given in the design specifications. The main driving force of the layout stage are the derived constraints (79%). It should be

noted here that almost half of these derived constraints originated from design objects outside the outer frame assembly.

In the detail stage of the design, the engineer is primarily refining the design to ensure that all the design objects work together. The main drivers of this design stage are previously derived form constraints (84%). This refinement process consists of comparing different form constraints to ensure that there are no conflicts in the design and changing or "patching" the constraints when conflicts arise. The resulting product of this constraint propagation and feature evolution is a formalized detailed design from which the mechanism can be constructed.

When examining the percentages for the overall design, the dominant value is of the derived constraints (75%). This value suggests that the designer deals primarily with constraints generated by the design itself rather than constraints brought in from outside the design space.

3.2.2.b Constraint Structure

As discussed earlier there were ten different constraint structures possible in the protocol. The percentage of each type of structure was calculated and plotted in Figure 9. By far the largest percentage of relationships (51%) fell into the second type, [form feature - instantiation]. The next highest percentage (20%) was for the third relation, [form feature - form relation - form feature]. This supports the earlier notion that the majority of the work performed was concentrated on the form features. It should be noted that most constraints of the first relationship type, [functional feature - instantiation], were verbal simulations of the machine's behavior (i.e., [machine behavior - operator slides plate in, pivots it and turns

the frame that holds it]).

The 10 constraint structures can be divided into two separate categories: form constraints and function

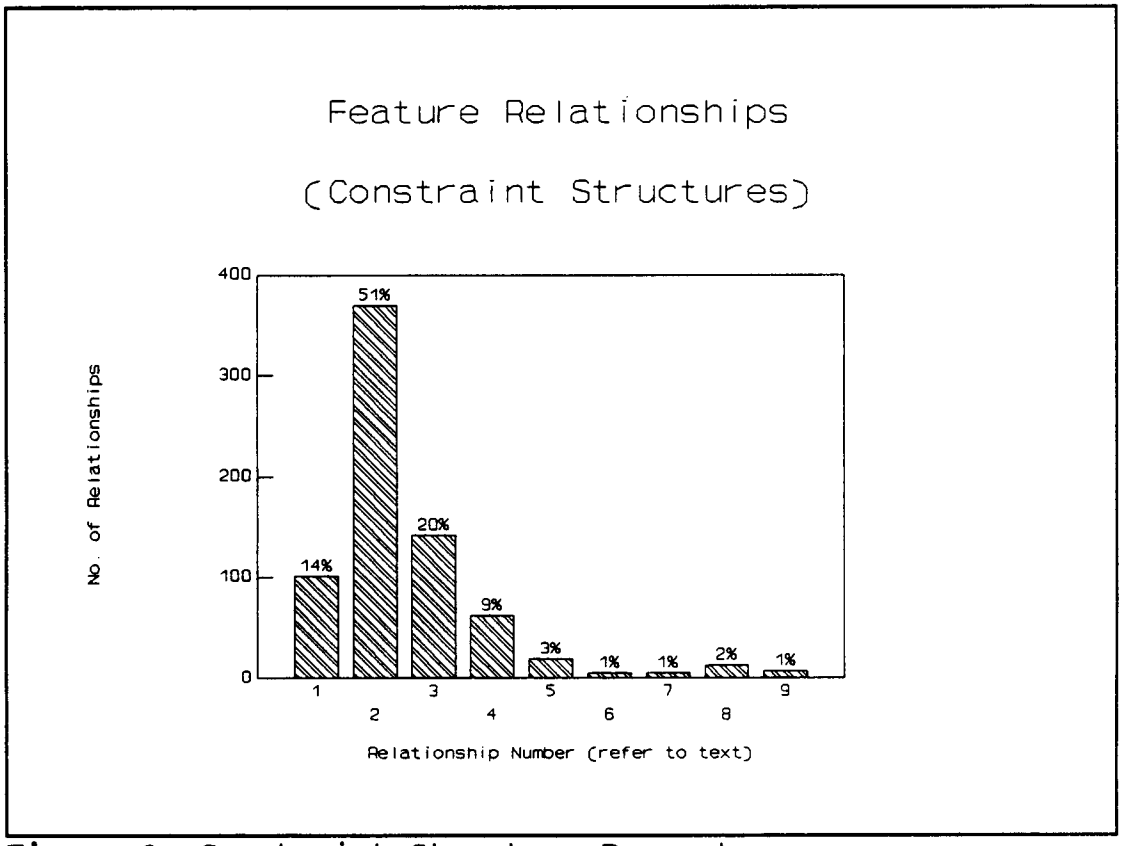

Figure 9 Constraint Structure Percentages

constraints. Structures 2,3,6,9 and 10 are form constraints. Structure 2 gives the instantiation of a form feature, whereas structures 3,6,9 and 10 give a form relation between either form or function features. Likewise, structures 1,4,5,7 and 8 are function constraints. Structure 1 gives the instantiation of a function feature, whereas structures 4,5,7 and 8 give functional relationships between either form or function features.

Nearly three-quarters (72%) of the constraints are of the form type and the other quarter (28%) are functional.

These values are not surprising, since by nature design is a very form-oriented process. Since the design problem studied was driven primarily by functional requirements, it might be expected that for a design with more form-oriented specifications, the percentages observed would have an even wider spread.

3.2.2.c Level of Abstraction

The level of abstraction for a constraint can fall into one of three sub-divisions: abstract, intermediate, or concrete. The concept of abstraction is actually a continuous function rather than the discrete form used here. However, to codify abstraction is difficult and thus the three divisions are very loosely bound and may overlap each other in certain instances. Also note that the abstraction level refers to the "instantiation" or "relation" the features attain rather than the representation of the feature. Therefore the current abstraction of a feature is represented by its most recent instantiation.

Abstract constraints are very general and are usually concentrated in the conceptual stage of the design process. An example of an abstract constraint is "small axle on a framework [the front pivot point]", which is the first conceptualization of the front pivot.

Intermediate constraints are more descriptive in nature but still not fully defined in a precise quantitative notation. These constraints are predominantly found in the layout stage of the design, but they can be dispersed throughout the design effort. An intermediate constraint is "our front pivot point we want right in the middle of the tank". This constraint tells approximately where the pivot should be (in the middle of the tank), but not exactly.

Concrete constraints are very specific, giving exact

quantitative values or specific functional qualifications. As expected, these constraints are found primarily in the detail stage of the design. A concrete constraint is "our front pivot point will be 6" from the center". Constraints tend to start as abstract and develop through intermediate into concrete specifications. However, this is not always the case, as many design objects are brought into the design (introduced) at a concrete level.

As with the constraint sources, the levels of abstraction were also grouped into the three different design stages: conceptual, layout, and detail. The resulting percentages can be seen graphically in Figure

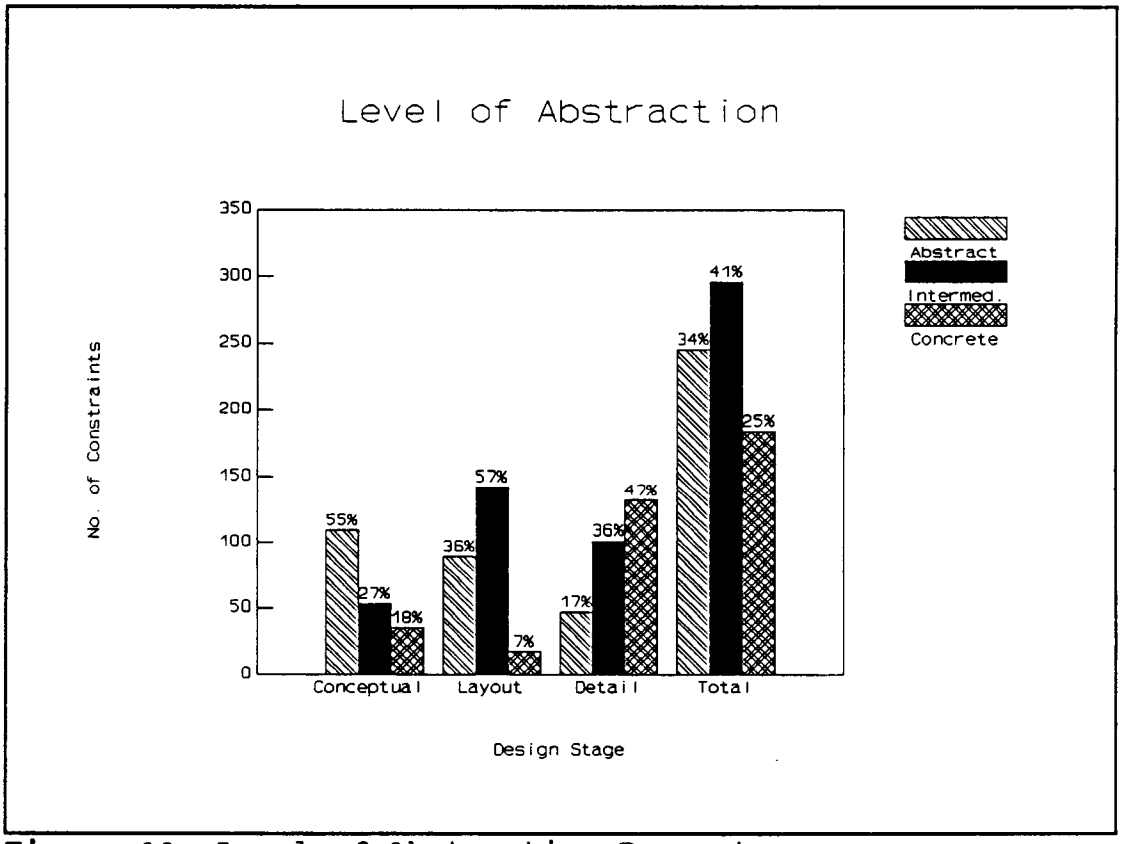

Figure 10 Level of Abstraction Percentages

10. In the conceptual stage, the majority of the constraints were abstract (55%), which should be expected.

In this stage, the concrete constraints appear to be more numerous than what might be anticipated (18%), but this can be accounted for when realizing that most of the given constraints are stated in a concrete form. In the layout stage, the intermediate constraints are predominant (57%), which is not surprising considering the above definitions. In the detail stage of the design, the concrete constraints have the largest magnitude (47%). They are followed somewhat closely by the intermediate constraints (36%). Since the design studied was not completed through to the level of full detailed drawings but instead to the high level layouts with manufacturing notes, it is expected that with more detailed design, the concrete constraints would increase greatly in the detail stage of the design.

The percentages for the entire design can be seen in the total grouping in Figure 10. The first observation that can be made of these values is that they are fairly well distributed. It can also be noted that the largest percentage occurs at the intermediate level, which suggests that the design was developed significantly on a transitional basis, that is neither concrete or abstract. As stated above, the concrete constraints would increase significantly if the design were carried out to complete detailed drawings.

3.2.3 Constraint Mapping

Since there were 725 constraints identified during the examination of the protocol data, a graphical form of representation was created to facilitate easier examination. The representation developed distinguishes between feature development, input constraints, and resulting constraints. It also makes tracing the development of individual features fairly easy.

Since it was impossible to graphically represent all 95 of the outer frame features, only those that attracted considerable designer attention were examined. This reduced the features to the more manageable amount of 23. These features describe the 8 components of the outer frame assembly, with each component having anywhere from 1 to 5 features associated with it. Another reduction was made by combining all the function features into one feature. The list of the features diagrammed in the constraint map can be found in Figure 11. This list can also be referenced in Figure 6, the feature tree.

1. Machine function
2. Outer frame function
3. Outer frame location
4. Lever arm general
5. Lever arm length
6. Lever arm cross-section
7. Lever arm hole
8. Lever arm location
9. Front cross member general
10. Front cross member
11. Front cross member function
12. Front cross member location
13. Back cross member function
14. Back cross member location
15. Diagonal cross member location
16. Front pivot point general
17. Front pivot point diameter
18. Front pivot point location
19. Back pivot point location
20. Doubler thickness
21. Doubler function
22. Doubler location
23. Bearing general

Figure 11 Constraint Map Feature List

The particular technique for constraint mapping was arrived at through several iterations. This graphical form of the constraints is not meant to be a formal representation but only a means to observe trends and patterns in the constraint-feature relationships. A section of the constraint map generated is given in Figure 12. The constraint map consists of horizontal lines, with each line representing a different design object feature. The evolution of the feature progresses to the right with respect to time. Each design decision made on the feature

is placed on the line at the time it occurred; these represent the change in the resulting constraint. The limiting constraints (forcing constraints) that affected the design decisions are indicated by vertical lines originating at the constraining feature and terminating with an arrow at the constrained feature. These vertical lines are either incoming or outgoing constraints depending on whether they began or terminated on the feature line under examination. Incoming constraints can be given, introduced, or derived constraints. Outgoing constraints are all derived constraints. This constraint map allows the study of any specific feature from its initial conception to its final completion with all the incoming and outgoing constraints easily observed.

The constraint map for the outer arm assembly was based on the 23 features discussed above. The development of these features, the constraints, and the design decisions required a piece of paper nearly thirty feet long. As this is not easily presentable, a small portion of the map is shown in Figure 12. This section is derived from the section of protocol presented and reduced earlier. In this section of the map there are two features that are worked on: the "lever arm length" and the "front pivot point location". The top row of constraints are those given in the design specifications. The second row are those which are introduced by the designer. The third row contains the derived constraints which originate from features outside the outer frame assembly. The design progresses to the right with respect to time.

This segment of the design begins with the accumulation of two given constraints followed by a previously derived constraint on the machine process and an introduced constraint dealing with the longest moment arm. These four constraints all act as limiting

constraints on the length of the lever arm. The last introduced constraint also affects the location of the front pivot point. The design decisions made on the front pivot point are now used to limit the length of the lever arm, which is drawn to scale in a layout drawing.

Although this small portion of the constraint map may appear complicated, it greatly clarified the complex relationships that existed between the constraints and

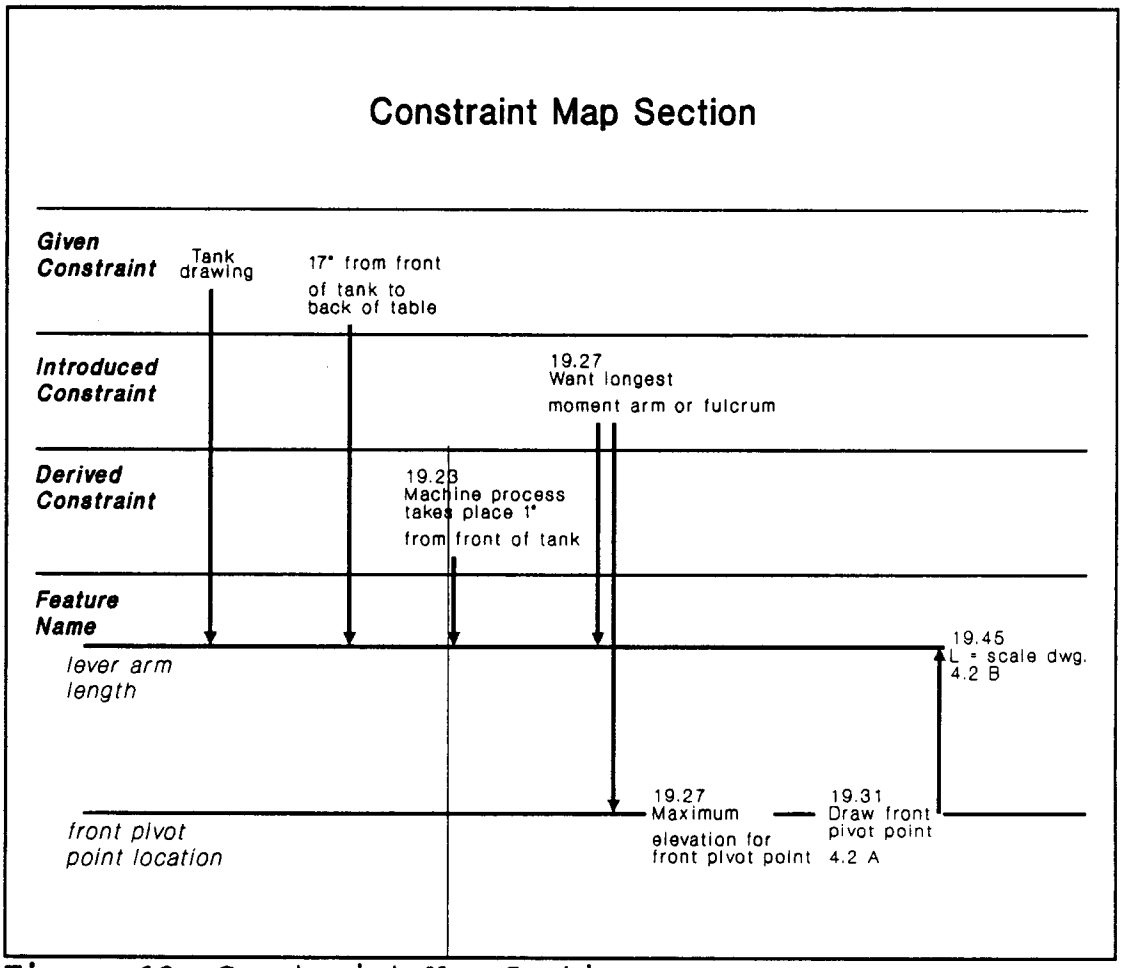

Figure 12 Constraint Map Section

features. Another aid to the study of the design objects was the placement of actual drawings made by the designer on the constraint map at the time intervals at which they occurred. This allowed for a better understanding of how a design object was progressing at any given moment in the

design.

One interesting pattern noted on the constraint map was that of constraint looping. A constraint loop occurs when feature A acts as a limiting constraint on a design decision made on feature B which in turn acts as a limiting constraint on a new design decision made on the original feature A. Although the above explanation is made for only two features, constraint loops were noted that involved as many as four different features. Constraint loops imply that the interrelationship of design decisions and the process of mechanical design is not easily decomposable into independent design problems.

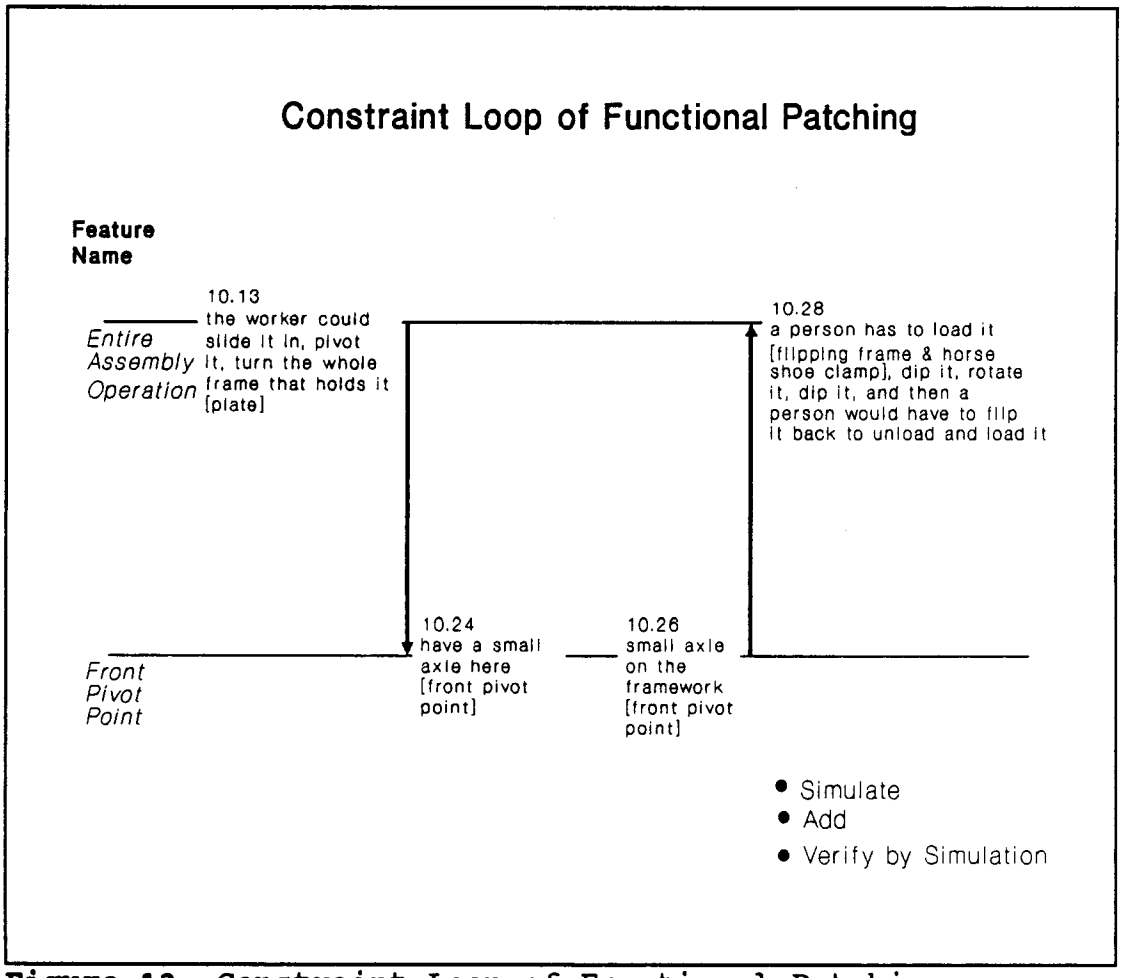

Figure 13 Constraint Loop of Functional Patching

An example of a simple constraint loop taken from the protocol can be found in Figure 13, which is discussed below.

Of the constraint loops identified, two major reasons for their occurrence were observed. The first, functional patching occurred primarily in the initial conceptual stage. Functional patching consists of a behavioral simulation of the design which results in an addition or modification of a form constraint. These changes or alterations in the design objects then alter the behavior of the design and are verified by a new revised behavioral simulation. The new behavioral simulation was usually less abstract than the original simulation. The result of functional patching was the introduction of a new design object and a buildup of the functional behavior of the design rather than the decomposition of the machine process into separate functions. The functional behavior is simply the behavior exhibited by the design object.

Figure 13 presents an example of functional patching from the constraint map. Initially there exists a behavioral description for the entire assembly. To perform this behavior an axle (the front pivot point) is placed in the framework. The behavior of the assembly is then re-simulated to ensure that it performs satisfactorily. Five occurrences of this type of constraint looping were observed for the outer frame subassembly.

The second reason for the use of constraint loops was that of component interfacing. This occurred when the designer was working on a feature and realized that a design decision on an interfacing feature needed to be made. The designer would then postpone work on the first feature, make the needed decision on the second feature, and then return to the first feature. These loops were seen mainly in the layout stage, and the features involved

were primarily spatial relationships.

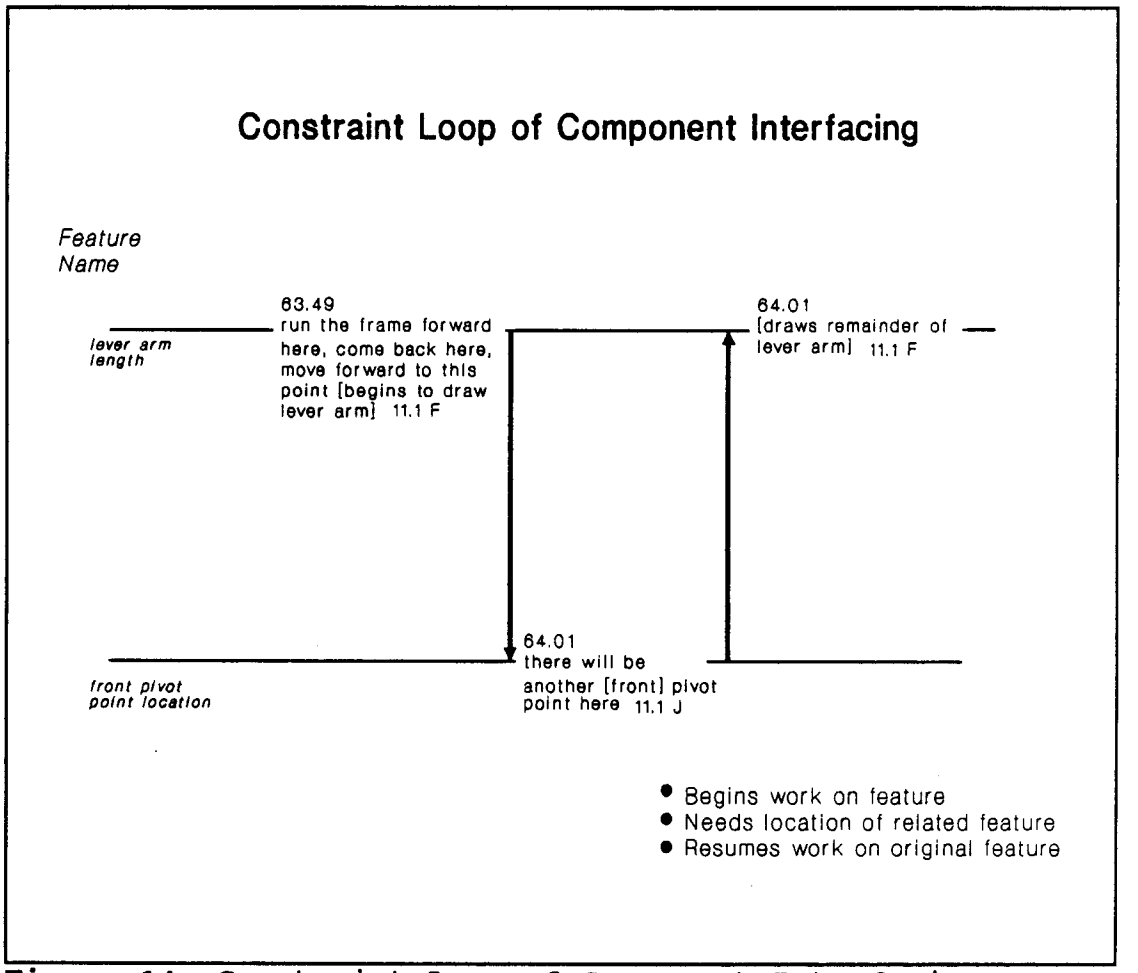

Figure 14 Constraint Loop of Component Interfacing

One example of this type of loop can be found in Figure 14. This excerpt from the constraint map begins by the designer starting to draw the lever arm length. But the location of the front pivot point needs to be determined first. Once a design decision on the front pivot point is made, it acts as a limiting constraint on the lever arm. Six occurrences of this type of loop were observed. Many of those dealt with the pivot point location. This is because the pivot points were the

primary interfacing components for most of the machine subassemblies.

3.3 Conclusions of Protocol Analysis

The analysis performed resulted in a substantial amount of insight gained on the process of mechanical design. Using knowledge gained from the protocol analysis and from the observations made above, a list of conclusions was produced.

1. Features are created on an as-needed basis. They are driven by both form and function requirements, and they are usually evolved from the domain knowledge of the designer.
2. A design begins as an abstract functional need, subject to form restrictions. By changing levels of abstraction, these needs evolve into a finalized specific design. This advancement of the abstraction level is a direct result of the constraint refinement.
3. The mechanical design process is a non-linear procedure that contains constraint loops that closely interrelate different features of the design.
4. A design object representation must posses the following properties:
   a. Be hierarchical in structure.
   b. Allow for interfacing between feature nodes.
   c. Allow for both form and function properties.
   d. Be general enough to allow for various types of features.
   e. Be closely coupled to the design's constraints.
   f. Allow for textual, numerical and graphical notations.
5. A constraint representation must possess the following properties:
   a. Must represent both form and function instantiations and relationships.

b. Allow for varying levels of abstraction.
c. Must be able to link relevant constraints, which will allow for constraint propagation and interaction.
d. Be coupled to both the features of the design and to the design decisions made on those features.
e. Allow for easy input of both graphical and textual constraints.
f. Allow for easy reference to constraints that are directly related to particular features and design decisions.

4. DESIGN PROCESS REPRESENTATION

The conclusions reached from the analysis of the protocol helped to establish a clear picture of requirements needed for the design process representation. This study served as the basis for the development of the design process representation, which will now be presented and discussed. The next subsection explains the basic design process model that was used for the representation. This is followed by a description of representation elements: design objects, constraints, and decisions (see Figure 1). Lastly, the representation for the constraints is expanded and explained.

4.1 Design Process Model

The overall mechanical design process is described as follows. The design generally begins with an abstract functional description that is confined by some given geometric constraints. This functional description along with the given constraints is then utilized by the designer in the formation of the final design artifact. The initial functional description can consist of either some perceived mechanical behavior or some required purpose. The given geometric constraints usually consist of some spatial or geometric limitations or boundaries set on physical properties such mass, strength, and load, or a combination of both. The final design artifact can exist as either the final detailed drawings or actual physical components. The design effort is the process by which the initial specifications are transformed into the design artifact. This transformation is the process of interest in the formation of the design process representation.

A model of the design process discussed above can be seen in Figure 15. Here the design effort is broken down

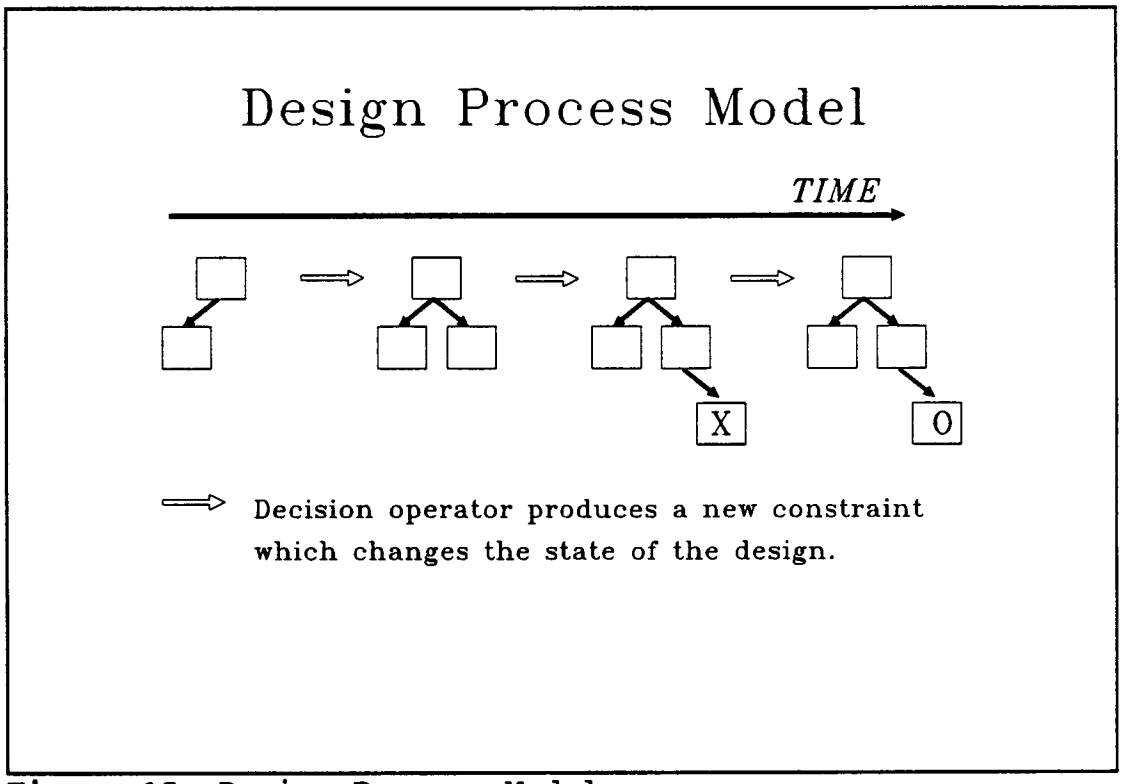

Figure 15 Design Process Model

into design episodes [Stauffer, 87]. Each design episode is depicted by a graph of boxes, representing the different states of the design. Each state is created as a modification or change from the previous state. A modification includes the addition or deletion of information into the design state (the addition of a box to the graph) or the revision of the values or relations defining that state (the change from X to O in a graph box). These additions and modifications to each design state come about by the application of the decision operator. This decision generates new information that affects and changes the state of the design. This set of new information comprise the new constraints of the design. By chronicling the generation of new constraints through the use of decisions, a history of the design process is recorded. From this model of the design process, the representation was developed.

4.2 Basic Design Process Representation

The design history representation is comprised of

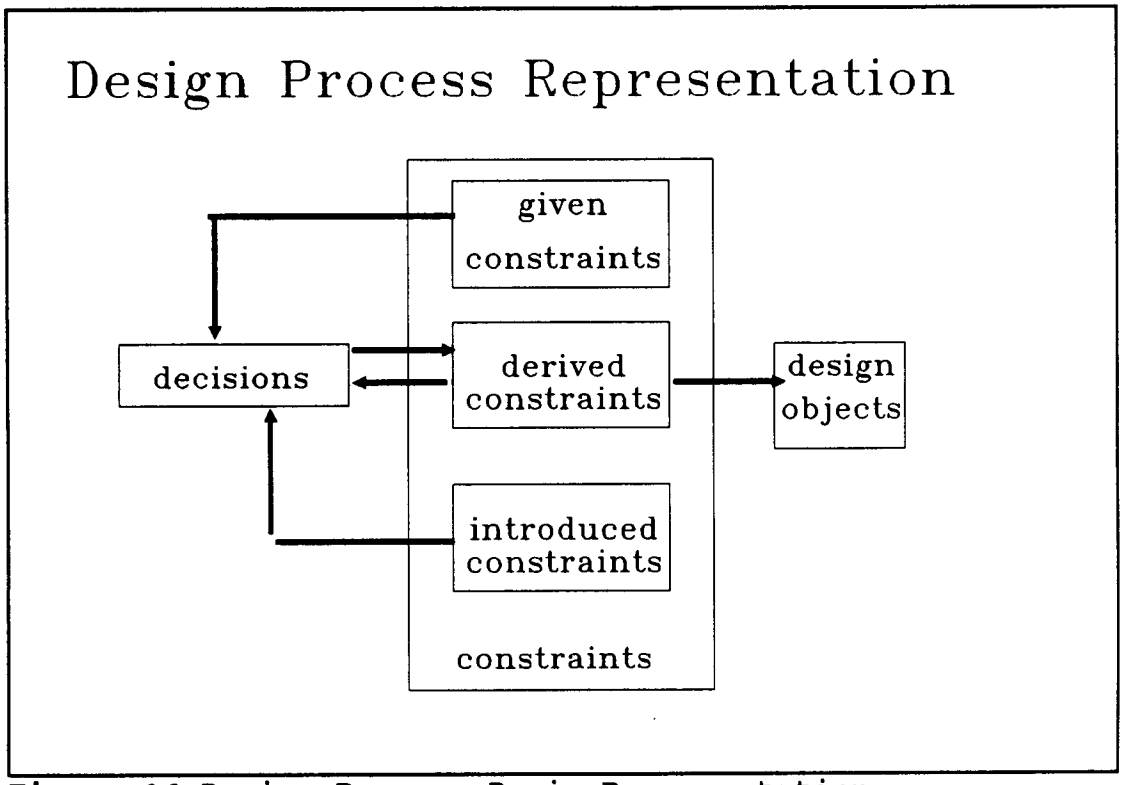

Figure 16 Design Process Basic Representation

three main elements: design objects, constraints, and design decisions. These three elements interact together as shown in Figure 16. Each of these elements is detailed below.

4.2.1 Design Objects

Design objects are the physical artifacts of the design. They represent the components of the design as well as the assemblies created by those components. Each design object is comprised of a set of attributes that are used to describe it, see Figure 17. Attributes are defined as any specific characteristic of a design object

that relates information about that object. The attributes usually consist of physical parameters, such as

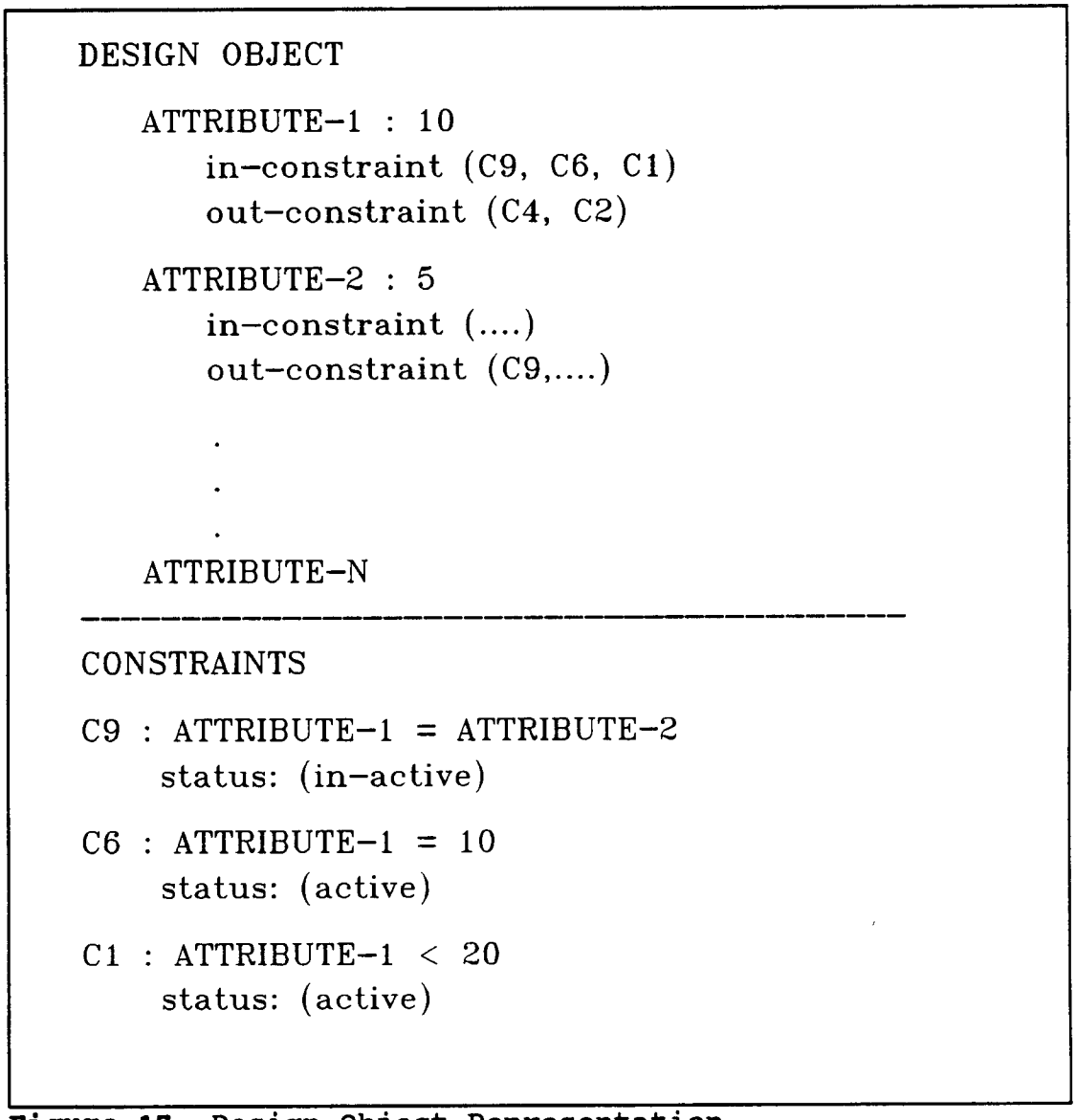

Figure 17 Design Object Representation

length, width, mass, location, etc.; functional descriptions, such as purpose or behavior; or any other property the designer may wish to define.

Attached to each attribute of the design object is an in-constraint list and out-constraint list, see Figure 17. The in-constraint list indicates which constraints were used to specify that attribute's value. The first

constraint in this list is the most recent. The out-constraint list denotes other constraints that have used the attribute's value in a computation or relation. The value of the attribute is determined by finding the first active constraint in the in-constraint list and then evaluating for that constraint. A constraint is marked as active or inactive as the result of some previous evaluation performed by the designer, (see status constraint role below). Evaluation of the constraint returns the value or relation specified by that constraint, for example, evaluating for the constraint A=B+C would return the sum of B and C. By specifying the attribute values of the design objects, a basic description of the design can be established. The constraints are the mechanism by which these attribute values are specified.

An example of how a design object attribute is specified is shown in Figure 17. Here attribute-1 contains in its list of in-constraints, C9, C6, and C1. We can see from the description of the constraints that constraint C9 is marked as inactive. Therefore it is disregarded when determining the value of Attribute-1. The attribute value is instead determined by evaluating constraint C6, since it is the first constraint in Attribute-1's in-constraints marked with an active status. The evaluation of constraint C6 returns the value of 10, which is the value specified by that constraint. An example of an out-constraint, in Figure 17, is C9 with respect to Attribute-2. The value of Attribute-2 is used to determine the value of attribute-1 in the solving of constraint C9, therefore C9 is placed as an out-constraint for Attribute-2. This accounting of the constraint is used in determining the constraint dependencies.

4.2.2 Constraints

The constraints are used to define the attributes of the design objects as well as the relationships that exist among those attributes. The design objects are the basic structure from which the constraints describe the specific instances of the design. The constraints contain the essence of the design, in that they specify the specific values and relations that exist within any particular design. In a design history, for example, a design object of the assembly 'bottom case' and its components of 'side wall', 'middle wall' and 'bottom', would have no meaning without its conglomeration of constraints defining the height, length, width, as well as the spatial relations of the walls and bottom. Thus the design objects with their respective constraints define the specific state of the design at whatever stage the design may be in.

4.2.3 Design Decisions

Design decisions are the processes by which new, derived constraints are created to change the design state, see Figure 16. The decisions are made by considering some previously existing constraints and applying some evaluation operator to them to produce a new derived constraint(s). The constraints input into the decision can be given, derived, or introduced. The resulting constraint is always a new constraint that when added to the existing constraint set affects or changes the state of the design in some way. Through this decision process, design objects and their attribute values are defined. In this way, the design decisions represent the processes of changing the state of the design. Thus the design objects represent the structure of the design, the constraints define this structure, and the decisions relate the process by which the design acquires and changes these constraints.

4.3 Constraint Representation

From the above explanation, it is apparent that the constraint representation is the most basic element of the design process representation. The constraint representation has to be flexible enough to model many different types of constraints and at the same time concise enough express the values and relations of those different constraints. Another consideration, is to allow for the easy modification or expansion of the constraint representation, which gives the design knowledge base the potential as a testbed for future research projects. The constraint representation use also provides information at a level sufficient enough to generate graphical images of the design. These images are for use in the graphical user interface developed for the design playback tool [Charon, 89].

From the previous discussion on the constrain analysis (Section 3.1.2.c) 10 basic constraint structures were presented. From these structures and the subsequent constraint classifications, three important aspects of constraint information were identified:

1. Constraint source, which specifies the origins of the constraint.

2. Constraint role, which defines what type of design object attribute was affected by the constraint.

3. Constraint language, which defines the form the constraint relation or instantiation was expressed in.

These three aspects, which were developed from design protocol data, are used as the basis on which the constraint representation is formulated. Each aspect will be expanded and discussed in the following sections.

4.3.1 Constraint Source

The constraint source denotes the origin of the constraint, and can be one of three types: given,

introduced, or derived (see Section 3.2.2.a for more detail). This classification of constraints by source within the constraint representation helps in determining the origin of the constraints. For example, whether they were produced from design textbooks or were derived by the designer in the decision making process.

4.3.2 Constraint Role

The second aspect of the constraint information is that of the constraint role. The need for this design information was developed from the examination the 10 basic syntactic constraint structures discussed previously

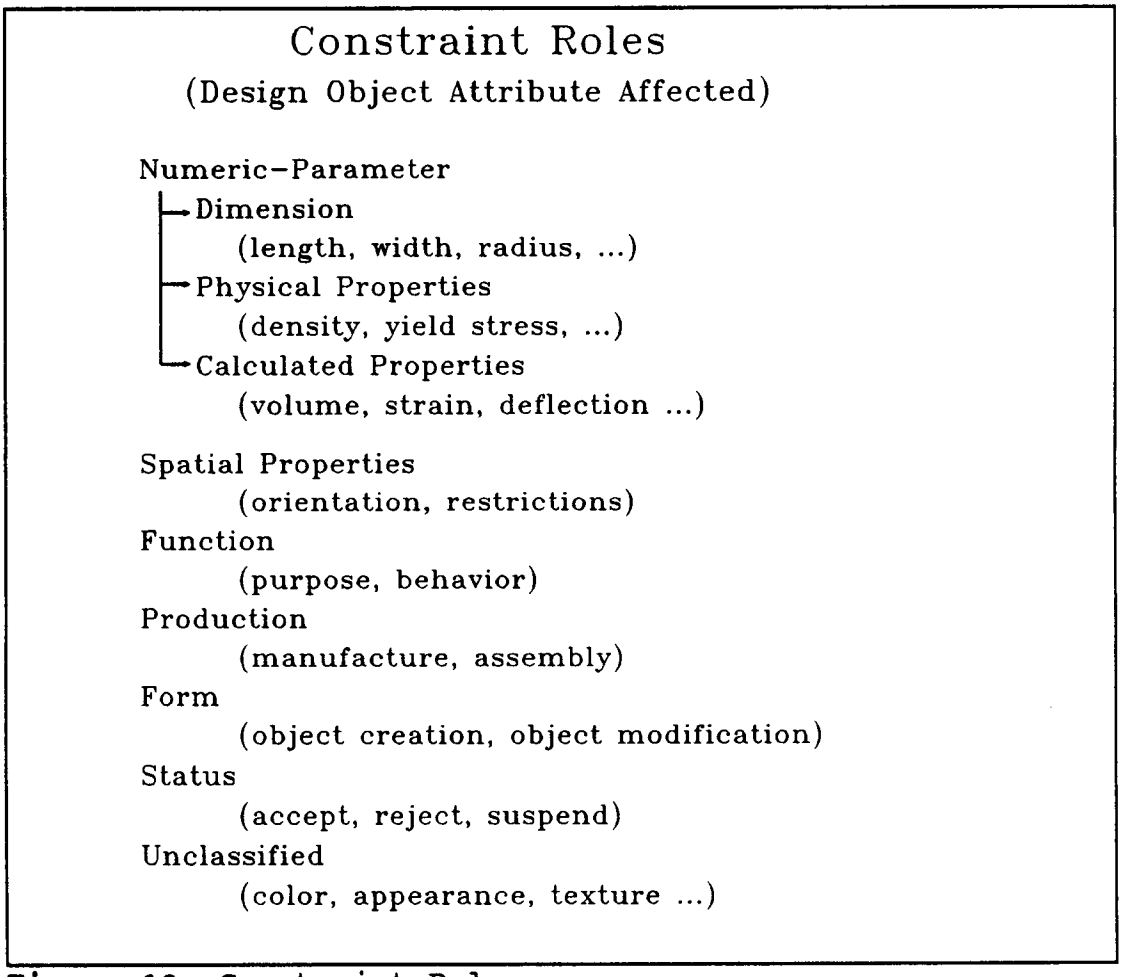

Figure 18 Constraint Roles

(section 3.1.2.c). By examining the dependent attributes of these syntactic structures, the constraint roles were developed. The constraint role refers to what attribute of the design object is being affected. The role of the constraint is primarily concerned with what parameter, property, or characteristic of the design object is being specified or defined, see Figure 18. This figure contains a list of the more common roles identified in the design protocols, but new roles can be added to this list as needed. Each role will be discussed below.

4.3.2.a Numeric Parameter Role

The numeric parameter constraints are those which deal with the numeric valued attributes of the design objects. This role is a grouping of the first three constraint attribute types shown in Figure 18. They include geometric dimensions (i.e., length, width, radius), physical properties (i.e., strength, weight, quantity), calculated properties (i.e., volume, stress, deflection), or any other property that can be specified by a numeric quantity.

4.3.2.b Spatial Role

The spatial constraints are concerned with the spatial relations of the design objects. This constraint role has been subdivided into orientation and restriction. The orientation constraints are used to describe locational relations between the different topologies of the design objects. These topologies include surfaces, edges, axis, and combinations thereof. Examples of these relations are "coplanar surfaces", "colinear edges", or "angle between axis". The orientation constraints are used to generate the x,y,z translations and rotations of the design objects.

The boundary or restriction constraints define what

spatial areas of the design cannot be entered or occupied. These constraints limit where the design objects can be located. They are often stated in the original specifications of the design but can also be generated by the designer himself. Examples include, "machine cannot touch within 1 1/2" of the table edge" or "plate cannot enter more than 1/2" into water surface". Although these constraints do not directly describe the spatial relations among the design objects they indirectly affect where the design objects can exist. Because of its difference from the spatial orientation constraints, the restriction constraints have been broken out into a special and separate subclass of the geometric constraints. This separation of the restriction constraints gives the capacity for the addition of future restriction checkers.

4.3.2.c Function Role

The function constraints describe the behavior or purpose of the design objects and have thereby been separated into two subclasses; purpose constraint and behavior constraints. This separation of function constraint has also been identified and used in [Hubka, 84]. Purpose constraints describe the intended requirements of the design objects. Where behavior constraints relate the manner in which the design objects act or behave within the design. In other words, the purpose constraints refer to what a design object is to do and behavior constraints refer to how the design object is to behave. For example, the purpose of fan blade may be to move air, but the behavior is to rotate at a specific speed about some specified axis. Another example from the protocol data is that of a spring contact. Its purposes are to hold the battery in place and to provide an electrical conductor but its behavior is to exert a force upon the battery's end. This distinction between types of

function constraints provides greater structure within the representation.

It should be noted here that traditional definitions of design form and function are not used in determining the constraint roles. The traditional concepts of design form and function often overlap [Rinderle, 88] [Eastman, 89]. The functionality of a design can be present in both the configuration of the design objects as well as the parameters of those objects. Therefore design functionality can be implicitly present in the form of the design. There is no definite distinction between design form and function, but a distinction can be made between the form and function attributes of the design objects. The form attributes describe the physical aspects of the design objects, where the function attributes define what the behavior and/or purpose of those objects is to be.

4.3.2.d Production Role

Production constraints contain information on the assembly and manufacture of the design objects. These constraints are used to describe how an object is to be produced or made. Examples of production constraints extracted for the protocol data are "weld at (flipping frame) back cross member and inner arm" and "shoulder bolt should be locktited in place". The subclasses of the production constraint are assembly and manufacture. The manufacture describes how the design object is to be fabricated, and assembly describes how the design is to be constructed or put together.

4.3.2.e Form Role

The form constraints are used to account for the creation and modification of the design objects. These constraints are used to specify the names, shapes, parents, and components of the design objects. Simple

examples are, "there will be a lever arm" or "this gripper will be in the shape of a U". The components of the design objects refers to what sub-parts comprise or make that particular object. The parent object is just the reverse, this denotes what assembly or component the design object belongs to. This constraint role has been subdivided into create object and modify object form. This division was made to clarify the difference between creating a new design object and just modifying an existing one.

4.3.2.f Status Role

Status constraints are used to denote the acceptance, rejection, or suspension of other constraints. A constraint is accepted, rejected, or suspended through the process of a design decision evaluation. By marking a constraint as rejected, it becomes inactive within the design space. As a result of being marked as inactive, the constraint is no longer solved for when determining an attribute value of a design object, see Figure 17 and discussion in section 4.2.1. Status constraints indicate that an attribute value or relation was evaluated against some measures. The result of this evaluation was the creation of a status constraint that accepted, rejected, or suspended the constraint that originally specified the attribute value or relation. In this way rejected constraints are never deleted from the design space but are instead set to an inactive status.

4.3.2.g Unclassified Role

The last category of constraint roles is the unclassified constraint role, which is used as a catch-all for any constraints that are not yet explicitly represented. These constraints are employed to specify design object characteristics such as color, rigidity,

appearance, power and so on. Because of the abstract nature and diversity of these constraints, they have not been subdivided into subclasses. These constraints include "chemical bonding takes place instantaneously" and "flipping frame is no effort to handle". Further research projects may provide subdivisions within in this class or create new classes of constraint roles.

Tied closely to the constraint role is the constraint language. Where the role indicates what kind of design object attributes are affected by the constraint, the language determines how the constraint is expressed (and represented in the computer).

4.3.3 Constraint Language Representation

The constraint language representation is concerned with what form the constraints are expressed in: equational, graphical, or textual. This should not be confused with the how the constraints were stated in the protocols by the designer (e.g., verbally or drawn on paper). For example, a designer could just as easily state an equational constraint verbally as draw it on paper, but the basic expressional form of the constraint, that of an equation, is still the same. Each constraint is expressed in terms of specific language operators (e.g., =, <, coplanar, supports, etc.). These operators can be grouped together according to type to form different subclasses for each constraint language, see Figure 19.

4.3.3.a Equational Language

The equational language is used to constrain the numeric parameters of the design objects. Five primary subclasses of the equational language have been observed: equality, inequality, qualitative, conditional, and other.

Constrain Languages

(and language operators used)

Language	Language Operators
Equational	
equality	(=)
inequality	(<, >, <>, ...)
quality	("small", "maximize")
coditional	(if <cond> then <set value>)
other	(table or graph lookup)
Graphical	
spatial-orientation	(co-planar, co-axial, ...)
spatial-restriction	(away from, within, ...)
Textual	
structured	(predicate phrase)
simple	(any verb phrase)
special	constaint role dependent

Figure 19 Constraint Languages

It should be noted here that all the equational subclasses discussed below can incorporate standard algebraic operators such as +, -, /, and *.

Equality constraints are used to specify exact values for parameters; the primary operator used is the equality sign (=). Examples of the equality constraints are "the beam height is equal to twice the width" (height = 2 * width), or calculating the stress as function of load, length, height, and width (Q = f(P,l,a,b)).

Inequality constraints are used to specify parameter values as consisting of a range or subset of values. The operators used in this equational language subclass include (<, >, <>). Examples of inequality constraints are "the lever arm length must be less than table depth"

(length < depth), or "the load should be kept below 10 lbs." (load < 10).

Qualitative constraints are used to specify parameter values in abstract or relative terms. Examples for these constraints include "keep the weight as small as possible" or "want this gripper width to be short". It is hard to consider "small as possible" or "short" as equational operators, but in terms of specifying numeric values, they do describe a desired range or relative value to be obtained. Therefore, they warrant separation from the standard class of qualitative measures such as color, texture, appearance and so on. This concept of a qualitative language is associated with the fields of qualitative physics and qualitative reasoning. One reason for distinguishing the equational qualitative constraints as a separate subclass is to allow for the future development of computational solvers, that could solve for these types of equational operators.

Conditional constraints are used to express constraint relations that are stated in conditional terms. These include statements like "if the spring constant is high, this hole can be moved back" or "if the flipping frame rotation is a problem, we can shorten the lever arm length". The main provision for a conditional expression is that the relation or value expressed must be contingent upon a conditional phrase.

The last equational language identified is "other". The "other" constraint language is used to represent constraints whose value or relation cannot be expressed in simple computational terms. For example, if a constraint's resulting value was dependent on an FEA or other numerical method, it would fall in this category. Constraints dependent on table or graph lookups are also considered under this constraint category. The main consideration for these constraints is that they specify

or define a numeric parameter value by some complex computation and that the computation is not easily expressed in simple terms.

4.3.3.b Graphical Language

The next basic constraint language is the graphical language. This language is used to describe the spatial relationships that exist among the design objects as well as the restrictions that exist on the space the design objects may occupy. Therefore it has been divided into two subclasses: spatial orientation and spatial restriction. The operators used in the spatial orientation constraints are dependent on the topology of the design objects that is being used. Since this research is concerned with the 3-dimensional environment used by designer, the operators describe relations between surfaces, axis, and edges of design objects. Therefore the operators employed by these constraints define spatial relationships such as coplanar, perpendicular, co-axial, colinear edges, or above. Typical constraints include "this outer arm front cross member is 1" from this flipping frame rear cross member" or "these two datum lines, front and back pivot, are parallel". The subclassification made on the spatial orientation language is based on the topology of the design objects and consists of surface constraints, axial constraints, edge constraints, and combinations thereof. Other or more complicated constraint subclasses can be added as required.

The spatial restriction subclass of the graphical language is also connected to the topology of the design objects. These constraints specify either areas or surfaces that should not be occupied or touched by the design objects. They include statements like "do not go more than 1/2" under water surface" or "keep this table

side area free". A subclassification is also made here based on whether the constraint is dealing with a surface or a volume.

4.3.3.c Textual Language

The last basic constraint language, the textual language, is used to describe relations or values of the design that cannot be stated with equational or graphical operators. The textual language operators consist of all the verbs in the english language. Because of this infinite quantity of possible operators, the textual language was subclassed according to the different representation schemes used rather than trying to classify the constraints by the operators used. The three textual subclasses include structured, simple, and special. As with the other constraint languages, additional textual subclasses can be added as needed.

The structured textual constraints are those constraints which express a value or relation in a predicate form. A predicate expression contains a verb

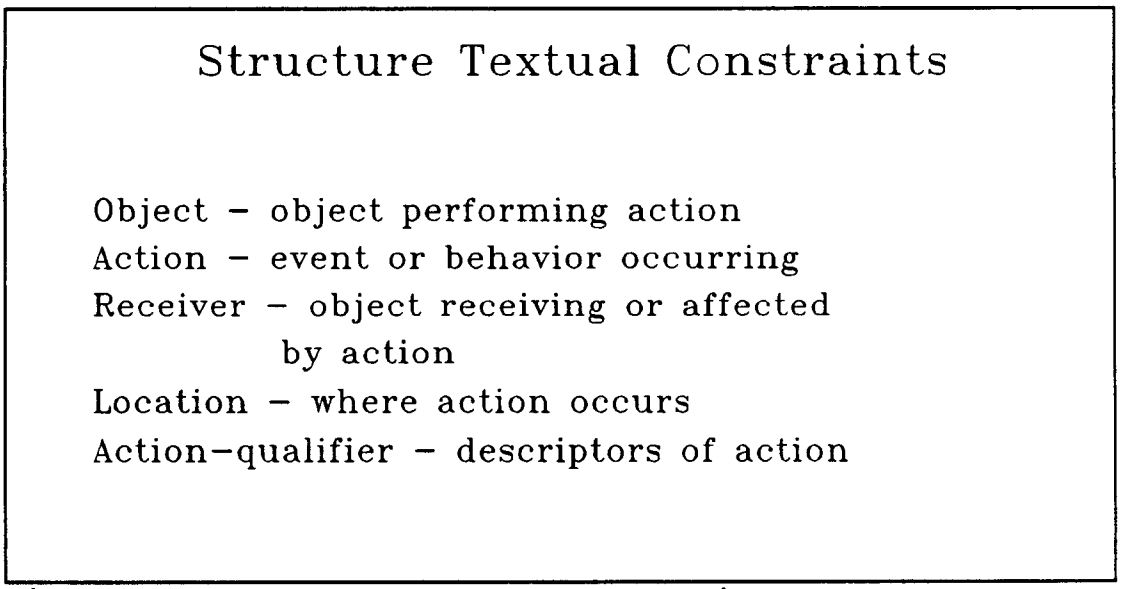

Figure 20 Structured Textual Constraints

with or without objects, compliments, or adverbial modifiers. Examples are "flipping frame raises out of water" or "plate slides into gripper slot". These forms are represented by decomposing the predicate statements into predefined syntactic structures, see Figure 20. The basis for this structure was developed from that prescribed in [Rich, 83], and also observed by [Lai, 87]. The object argument refers to what item is initiating or performing the action. The action argument denotes what action was performed or achieved, see [Libardi, 88] for a subset of typical function actions. The receiver argument denotes the recipient of the action. The action is further described in terms of an action location, where the action occurs, and some action qualifiers, adverbial modifiers of the action. This representation scheme provides a formalized structure for the constraints and allows for a more refined examination of the constraint by its action, receiver, or location values, rather than by just examining a textual string.

The second subclass of textual constraints are the simple constraints. These constraints are used when the relation or value expressed by the constraint is not in terms of a predicate form. Examples are be "flipping frame will have tendency to be heavy in back" or "machine power could be pneumatic, electric, or manual". These constraints are represented by noting the dependent feature and independent feature(s) of the constraint as well as the relation or value expressed. These are the most basic of the constraints represented, since they are modeled primarily as text strings.

The last textual constraint subclass is that of special. The special constraints are used for specialized constraint roles. For example the form constraints, those used to create and modify design object form, are represented using a specialized language consisting of a

set of arguments for the name, shape, parents, and components of the design objects. Other specialized languages are used for the status constraints, the function constraints, and the production constraints. The use of these textual language specializations is further discussed in section 5.2.3.c of implementation section. This use of specialized languages for specific constraint roles allows greater flexibility in representing and creating the textual constraints.

4.3.4 Constraint Causality

Another consideration of the constraint representation that is closely linked to the language of the constraints is the causality of the constraint relations. The representation assumes that all the constraints expressed in the design as causal, since they were originally stated that way by the designer. Causality refers to the dependence of the constraint relations [Chung, 89]. This causality in the representation implies a forced dependency in the constraint expressions, for example A = B + C, where A is the dependent variable and B & C are the independent variables. Another example is "block-a is below block-b", where the location of block-a is dependent on the location of block-b.

This assumption of causality greatly simplifies the constraint representation, by allowing the solving of constraints on an individual basis as opposed to solving sets of constraints simultaneously. If the constraints were considered non-causal or unidirectional, then complex parametric or variational constraint solvers would need to be implemented. Although the current representation could support parametric or variational constraint solvers, they are currently not implemented. Instead constraints are solved individually, since for the purposes of the design history, variational and parametric design solvers are

currently not required.

The three aspects of the constraint information (source, role, and language) are used to model the constraints in the design history tool. The constraint source denotes where the constraint originated from. The constraint role signifies what aspect of the design objects is being affected. The constraint language is used to express the values and relations of the constraints. Therefore, a specific constraint is represented by selecting the three subclasses of each of the appropriate constraint aspects (i.e., given source, numeric parameter role, equality language), creating the constraint, and filling in the necessary arguments. An implementation of the design process representation for the design history tool is presented in the next chapter.

5. IMPLEMENTATION

Once the formal representation for the design process was established, its implementation commenced. A complete printout of both the knowledge base and supporting source code used for the implementation can be found in Appendix III. Because of its power and flexibility, HyperClass was chosen as the object-oriented system upon which the design process representation is implemented. The following sections relate this implementation, explaining the underlying principles involved.

Before explaining the implementation of the design history tool, it is important to have a basic understanding of HyperClass and of object-oriented programming in general. The next section is devoted to explaining the basics of HyperClass and some of its more important facilities.

5.1 HyperClass Basics

HyperClass is an object-oriented programming environment, which was developed for the purposes of building, maintaining, and using knowledge base systems. It is implemented in Common Lisp and includes three subsystems: Strobe, MetaClass, and Impulse-86 [1]. Strobe is an object-oriented programming language that provides all the facilities of an object-oriented system. MetaClass was developed as an interface to Strobe which allows users to directly model their knowledge bases without being concerned with the intricacies of lower level programming. The interface consists of various object editors that can be used to build or modify the

---

[1] Strobe, MetaClass, and Impulse-86 are trademarks of Schlumberger Technologies Incorporated. See [Smith, 1987,1988] and [Schoen, 1989].

knowledge base. Impulse-86 is an interface-building tool. It provides a general and extensible substrate upon which to construct a wide variety of interactive user interfaces for developing, maintaining, and using knowledge-based systems. Impulse-86 is implemented in Strobe, and MetaClass is built from Impulse-86.

An object, which should not be confused with a design object in the design history model, is a fundamental element in HyperClass. A basic frame for an example

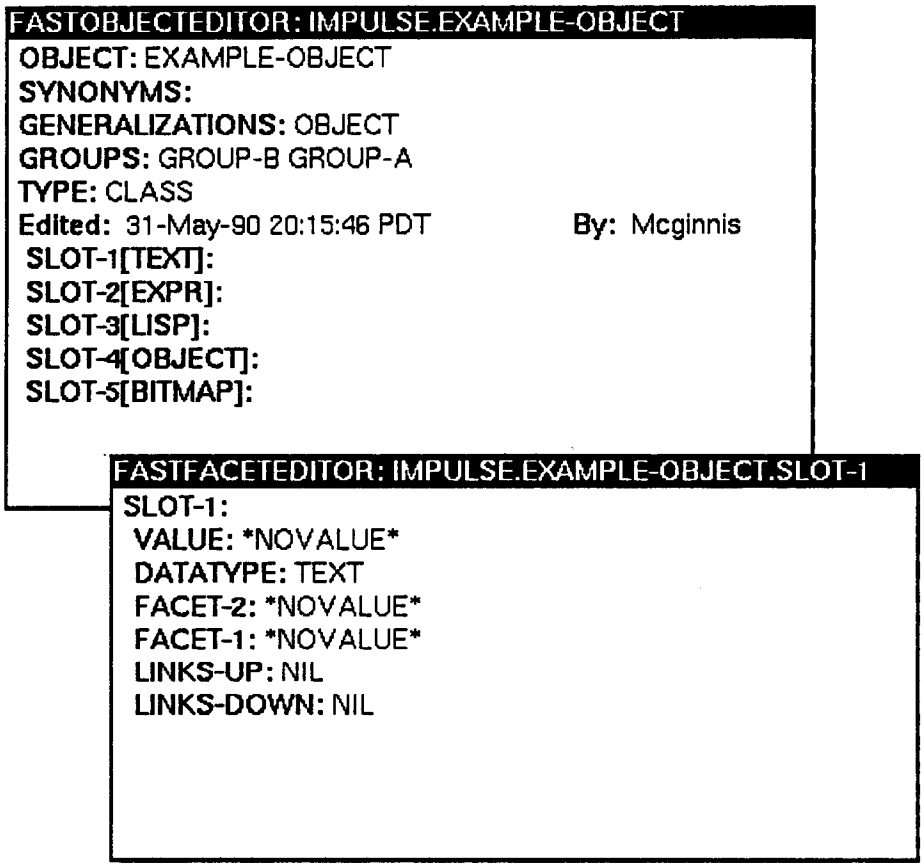

Figure 21 HyperClass Example Frame

object is shown in Figure 21. An object is a type of data structure that combines data and procedures in a single entity. Objects are referenced by name in HyperClass, with each object having a unique name distinguishable from

all others. In this example, the object name is EXAMPLE-OBJECT. Each object has properties that describe it. These are called slots, which are encapsulated in an object frame. As shown in EXAMPLE-OBJECT, SLOT-1 is one of the slots used to describe the object. Every slot has a value, which may be English text, a numerical quantity, a lisp procedure, pointers to other objects, or a bitmap. Each slot may also have facets in which further information can be located. The facets are used to describe the slot and its value. Each object may contain procedures or methods that constitute the actions that can be performed to exhibit some desired behavior or action. The procedures within the object and slot are invoked via their operation name.

HyperClass supports a modelling technique that allows users to directly represent the fundamental concepts of a particular domain. Within HyperClass, a distinction is made between the types of objects to be created: class or instance. A class object defines the properties and behaviors common to a set of objects. An instance object describes an element of a class object. Each class object may possess subclass and (sub)subclass objects. Instance objects, however, have no descendants, since they only denote a particular, unique individual.

Each fundamental concept in a domain can thus be represented by a class object. The taxonomical relationships that exist in a domain can be represented by the relationship between the class and its instance objects. For example, the "decision" concept of design history representation can be represented as a class object, called DECISION. This class object represents the generic decision which has a common set characteristics shared by all decisions. Each particular decision made by the designer is then represented as an instance of the DECISION class object, with each instance inheriting the

properties and procedures of its parent class. This inheritance mechanism allows many instances to share common properties and procedures at one identifiable location. The inheritance of an object is specified by the **generalizations** slot, see Figure 21, which contains the parent of that particular object. An object can be an instance of more than one class object. The **generalizations** slot in an object lists all of the class objects of which this object is an instance or subclass of.

The procedures associated with an object's slots are distinguished by their names and are invoked through the procedure of message passing. By sending a "message" to a slot or facet, a specific behavior or action can be performed. Since procedures are inherited in the same manner as properties, it is easy to build the required characteristics for a knowledge-based system.

Since one of the requirements of the design history tool is to provide a graphic interface, some means of displaying objects graphically is essential. This is accomplished by integrating a solid modelling package, Vantage[1], into HyperClass. This integration is described in [Charon, 1989]. The resulting HyperClass - Vantage hybrid is capable of creating solid models from the HyperClass knowledge base and then displaying those images with the HyperClass window system. Using HyperClass, Vantage, and their encapsulated components, the design history representation is constructed.

5.2 Design Process Knowledge Base

As explained earlier, the design history representation consists of three primary elements: design

---

[1] Vantage is a trademark of Carnegie-Mellon University [Balakumar, 1988].

objects, constraints, and design decisions. Using HyperClass, these concepts were directly implemented as class objects within a knowledge base. The creation and interaction of these three objects form the building blocks for design history knowledge base. A basic template for the design history knowledge base can be

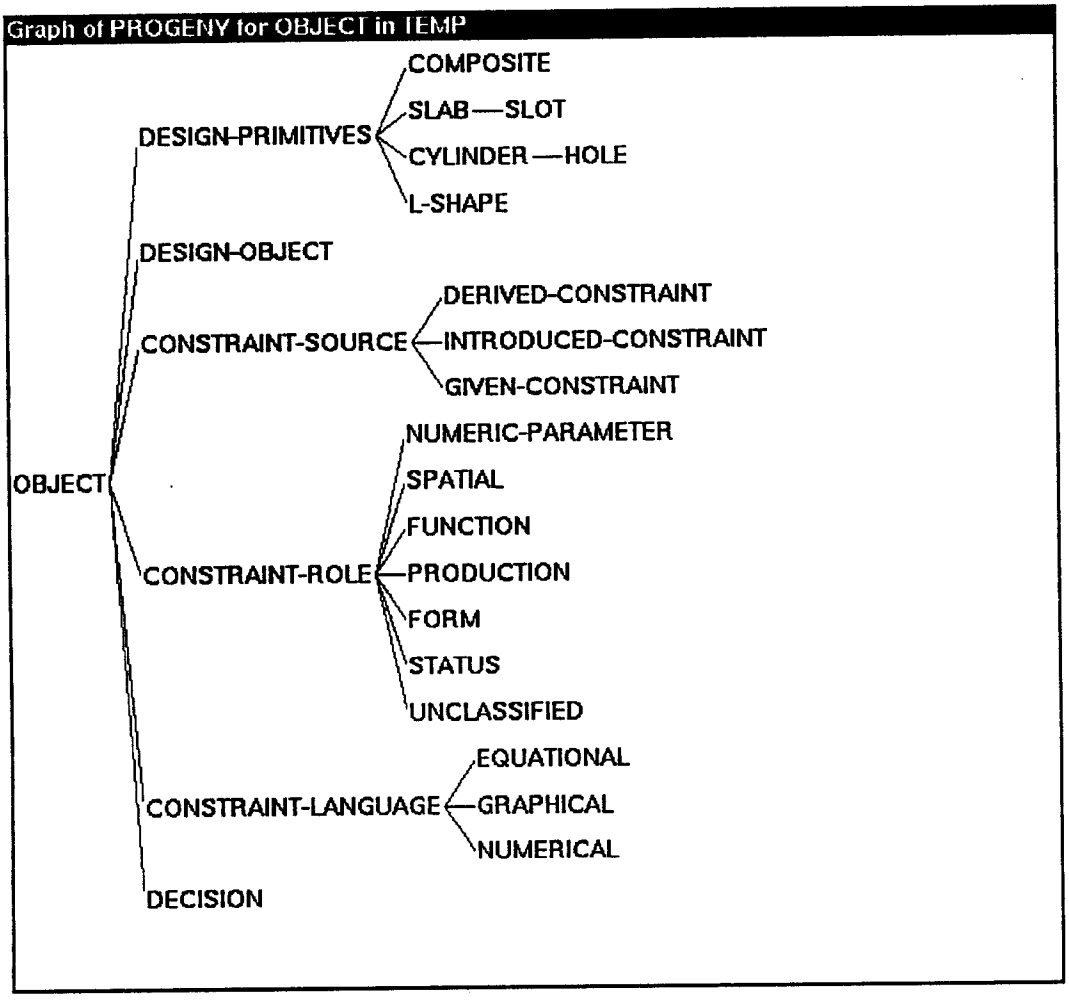

Figure 22 Design History Template

found in Figure 22. From the class objects shown in this knowledge base template, histories of the design process can be generated.

It is important to note here that because of the direct modelling capabilities associated with object

oriented programing, the previously discussed representation can be directly mapped into the implementation. In this way, basic representational concepts become class objects, important characteristics of those concepts become the slots of the objects, and specific instances of those concepts (i.e., table-top, decision-34, constraint-10) become instance objects with their appropriate slots specified. Because of this direct mapping, many of the details concerning the implementation have already been addressed in the representation discussion. At the cost of being redundant, these details will be restated, so that a complete description of the implementation can be presented without constant referrals to the representation previously discussed.

There are six primary class objects; **design-primitives**, **design-objects**, **constraint-source**, **constraint-role**, **constraint-language**, and **decision**. Some of these classes have sub-classes that represent specializations within that class. For example **design-primitives** can be decomposed into one of several simple or complex shapes, or can be composite in nature. The **constraint-language** class has also been decomposed into its more specific sub-classes to allow for different behaviors among the constraints. Specific instances of design objects, constraints, and decisions are generated from these classes and sub-classes. For example, a **table-top** design object is created from the class **design-object** and the sub-class **slab**. In this way, **table-top** inherits all the properties associated with both the **design-object** and **slab**. A particular constraint instance has the parents or generalizations of a **constraint-source** subclass, a **constraint-role** subclass, and a **constraint-language** subclass, inheriting properties from all three ancestors. Decision instances are generated solely from the **decision** class object.

5.2.1 Design Object Implementation

As previously stated, design objects represent the physical artifacts of the design. They provide a framework or basic structure in which to assign and order the constraints. A constraint specifying a value would be of little use unless that constraint was associated with the specific design object parameter it was specifying. The design objects are generated as instances of the class object **design-object** and subclasses of the **design-primitives**. In doing this, the specific design object instance inherits the general attributes associated with all design objects from the **design-object** class and inherits the specific attributes of the **design-primitives** subclass that was chosen. These properties are represented as slots in the design objects, as discussed below.

5.2.1.a Design Object

The **design-object** class object has as its descendants

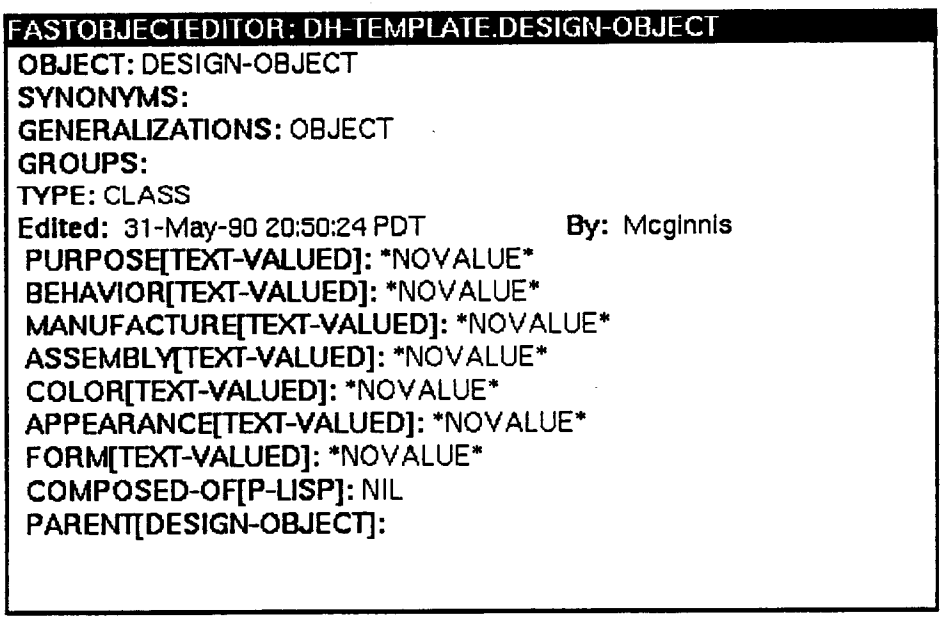

Figure 23 Design Object Class Frame

all the design object instances of the design. That is to say, each design object instance has the generalization of the **design-object** class object. This class object contains attribute slots that can be inherited by all the design objects, see Figure 23. The attribute slots include **purpose**, **behavior**, **manufacture**, **assembly**, **color**, **appearance**, and **form**. New attribute slots are created for the design objects as needed by the user.

Other slots of interest in the **design-object** class object include component slots, **composed-of**, and **parent**. The component slots are the slots created as pointers to the component objects of that design object instance. Component slots are created only for instances of design objects as they are needed. For example, a table object may have the components of a top and four legs. Each of these components are represented in the **table** design object by the slots **top**, **leg1**, **leg2**, **leg3**, and **leg4**. The slots in turn contain pointers to the design objects of top, leg1, leg2, leg3, and leg4, see Figure 24. The **composed-of** slot is a specialized method slot that automatically collects all the component slots of a particular design object instance and places this component list in the composed-of slot. The **parent** slot is a pointer used to indicate what assembly or part has this design object as a component. For example, the parent slot of the **table** design object, mentioned above, contains a pointer to an **office** design object, of which the table is part.

This use of component, **composed-of**, and **parent** slots helps maintain the hierarchial structure of the design objects. The hierarchial structure refers to the part-of relationship that exist within a design. For example, the legs are part of the table, and the table in turn is part of an office. This hierarchial structure is identical to the breakdown of the design objects into assemblies and

FASTOBJECTEDITOR: THESIS-TABLE.TABLE

OBJECT: TABLE
SYNONYMS:
GENERALIZATIONS: COMPOSITE DESIGN-OBJECT
GROUPS: DESIGN
TYPE: INDIVIDUAL

X-TRANSLATION[EXPR]: 0
Y-TRANSLATION[EXPR]: 0
Z-TRANSLATION[EXPR]: 0
X-ROTATION[EXPR]: 0
Y-ROTATION[EXPR]: 0
Z-ROTATION[EXPR]: 0
TOP[DESIGN-OBJECT]: TOP
LEG-1[DESIGN-OBJECT]: LEG-1
LEG-2[DESIGN-OBJECT]: LEG-2
LEG-3[DESIGN-OBJECT]: LEG-3
LEG-4[DESIGN-OBJECT]: LEG-4
PARENT[DESIGN-OBJECT]: OFFICE
MANUFACTURE[TEXT-VALUED]: (*) ("STEP-1: made from wood" "STEP-2: made by hand")
COLOR[TEXT-VALUED]: (*) "lighter than dark blue "
LENGTH[NUMERIC]: (*) "<6"
DEPTH[NUMERIC]: (*) 6
HEIGHT[NUMERIC]: (*) 12
FORM[TEXT-VALUED]: (*) "TABLE with shape COMPOSITE "
APPEARANCE[TEXT-VALUED]: "look nice "
CSG-NODE[OBJECT]: NODE-107
SCENE[OBJECT]: SCENE-108
REPRESENTATION[BITMAP]:

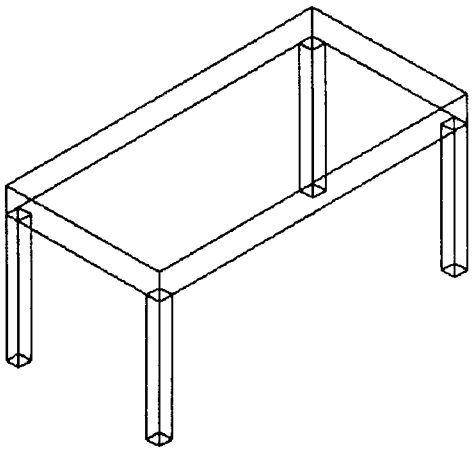

Figure 24 Table Object Instance Frame

components, except that components may be further broken down into their composite shapes or design features.

5.1.2.b Design Primitives

The **design-primitives** class object and its subclasses are used to contain information that is relevant to the

configuration of the design object. The design-primitives class object is decomposed into subclasses according to the basic design shapes that are used by the designer. These subclasses include **composite** objects, basic design shapes, such as **slab**, **slot**, **cylinder**, or **hole**, and more complex design shapes, such as **L-shape**, see Figure 18. The composite objects are those objects that have not yet been assigned a shape or whose shape is a conglomeration of shapes as with an assembly object. These subclasses can be expanded by adding new design shapes as required. The slots of these subclasses contain information on the particular topology and shape represented by that subclass. This information is denoted in the slots of the design primitives, see Figure 25. The slots shown are concerned with either the topology of the object, the location of the object, or are specialized methods used to create the csg (Constructive Solid Geometry) representation of the object.

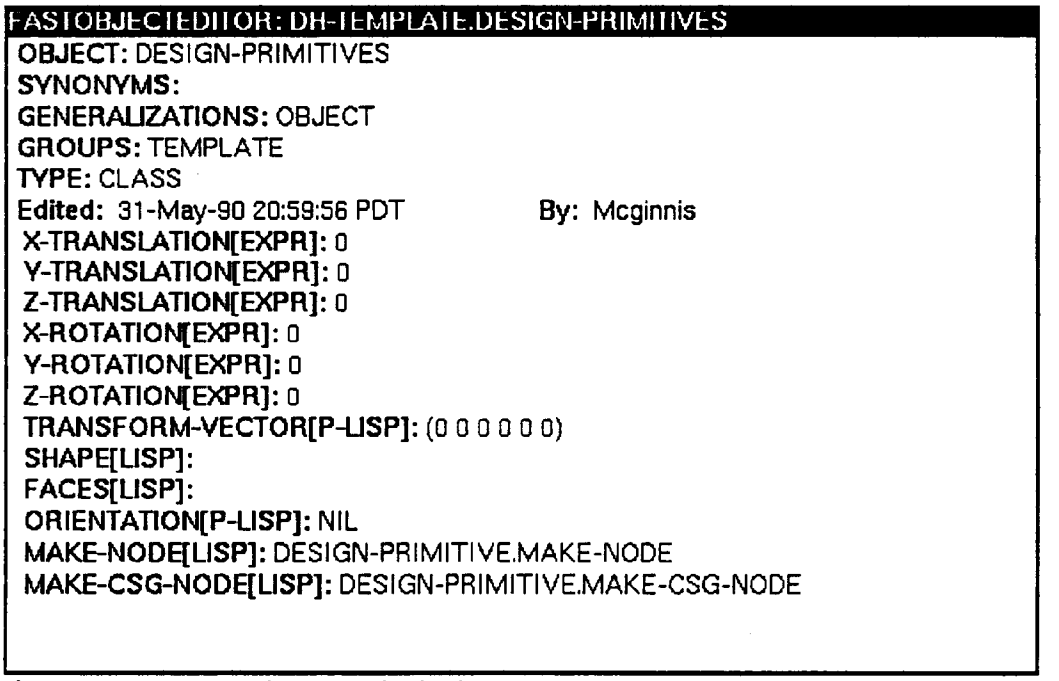

Figure 25 Design Primitives Class Frame

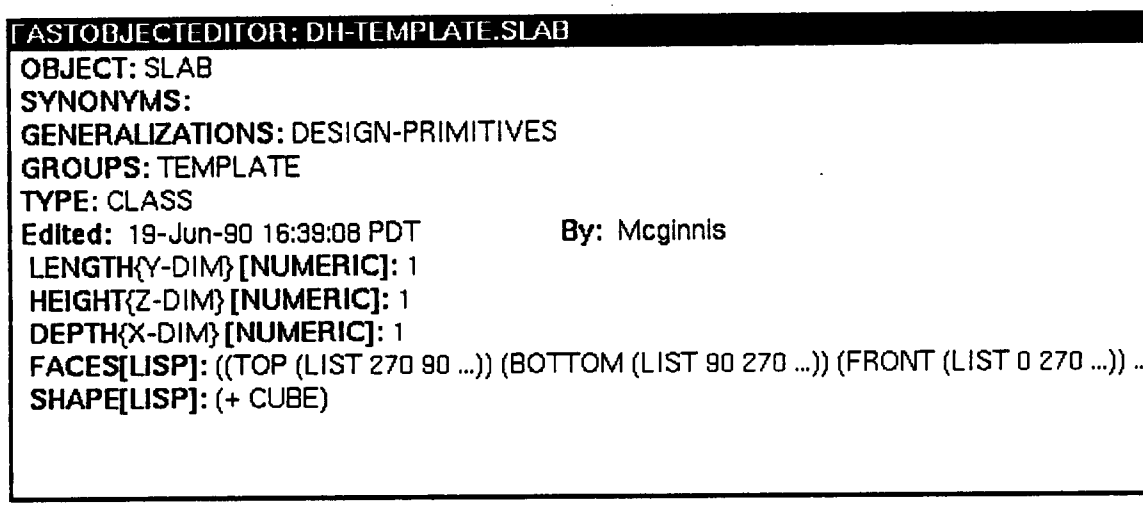

Figure 26 Slab Class Frame

The location and topology slots include the x,y,z rotation and translation, the **transform-vector**, **shape**, **faces**, and **orientation** slots. The x,y,z rotation and translation slots are used to place the object in the global reference frame. The values of these slots are specified by solving the spatial constraints assigned to that particular object. When these spatial constraints are fired, they change the values of the x,y,z, translation and rotation slots to enforce the relationship defined by the constraints (e.g., coplanar, co-axial, etc.). The **transform-vector** slot is a specialized design-primitives method, created for programming efficiency, that collects the values of the translation and rotation slots and displays that list as its value, see Figure 25.

The **shape** slot contains the basic csg shape of the object along with a specifier that tells whether the shape is a positive or negative volume. For example, the shape slot of **slab** contains the value (+ cube), signifying a positive cubical volume as the shape of the object, see Figure 26. The **faces** slot describes the surfaces of the design-primitive and associated plane equation for those

surfaces. It contains a list of the surfaces of the design primitive and a description of the planes upon which those surfaces lie. For example, the cube design primitive, has as one item in the **faces** slot, the value (TOP (LIST 270 90 0 (/ HEIGHT 2)), see Figure 26. This value can be evaluated to return a the cosine angles of the face normal vector and the distance from that face to the local origin. This information is used by the spatial constraints to designate which faces of the object are to be constrained and subsequently what translations and rotations will need to occur in order to satisfy that constraint.

The **orientation** slot is a specialized design primitive method that generates a textual description of how a particular design object instance constrained spatially. This method goes through x,y,z translation and rotation slots and determines which spatial constraints are currently affecting the orientation of the object. A textual phrase is then generated from each spatial constraint and verbosely displayed, such as "(leg-1 top) coplanar to (table-top bottom)". This allows a user to quickly examine what specific spatial constraints affect the orientation of an object without having to examine the graphical representation.

The specialized methods used to create the csg representation are **make-node** and **make-csg-node**, see Figure 25. The **make-node** method is used to generate the csg tree for the design object. It determines what components, if any, are present within an object and builds the csg tree for that object. The **make-csg-node** method is used to generate the leaf nodes of that csg tree. A **make-csg-node** method can be specialized for any design primitive that may be needed by the designer, such as the L-shape mentioned earlier.

Associated with the different shape subclasses of the

design-primitives class object, are the specific dimensional parameters of that particular shape. For the **slab** design primitive subclass, see Figure 26, the parameters required are the **length**, **depth**, and **height**. These are directly related to the x, y, and z dimensions of that shape that are required to build the csg representation.

A frame of a typical design object can be found in Figure 24. Here each slot represents a specific attribute of the object. A **representation** slot exists which contains a bitmap of the graphical image of the object. This representation was generated using the specialized csg representation methods discussed above. New attribute slots can be created as needed for the design objects, as there is no predetermined limit on design object attributes.

5.1.2.c Attribute Slots

The values for the slots of design objects are specified by associating a constraint with a particular attribute. Within each slot of a design object, there exist facets or sub-slots associated with that slot. Each

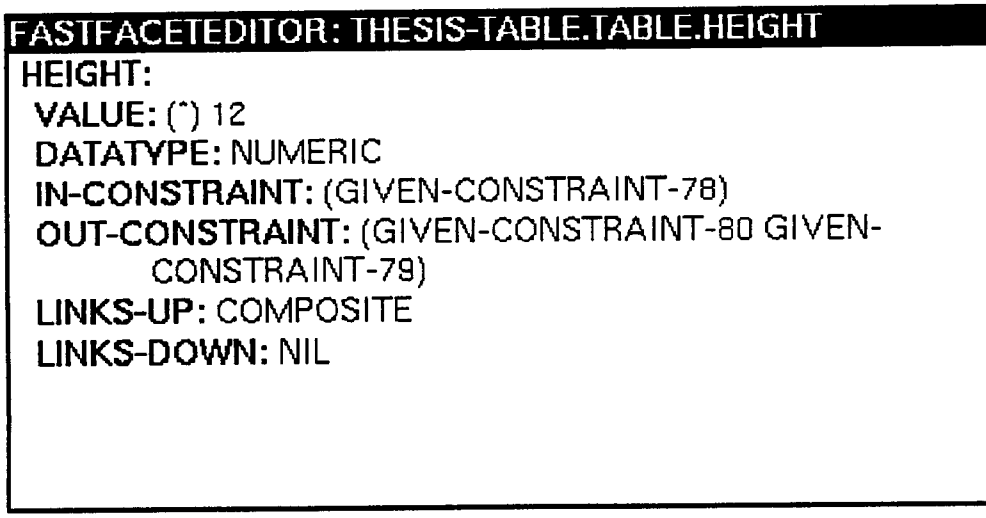

Figure 27 Table Height Attribute with Facets

parameter slot of a design object has an **in-constraint** and an **out-constraint** facet, see Figure 27. These facets are used to determine which constraint specifies the slot value, the in-constraints, and which constraints use this slot value, the out-constraints, see section 4.2.1 for further discussion. This use of **in-constraint** and **out-constraint** facets relates the temporal values of design object attributes as well as providing a means of specifying the constraint dependencies.

5.2.2 Decision Implementation

The design decisions are the processes by which new derived constraints are added to the knowledge base. Decision instances are created from the **decision** class

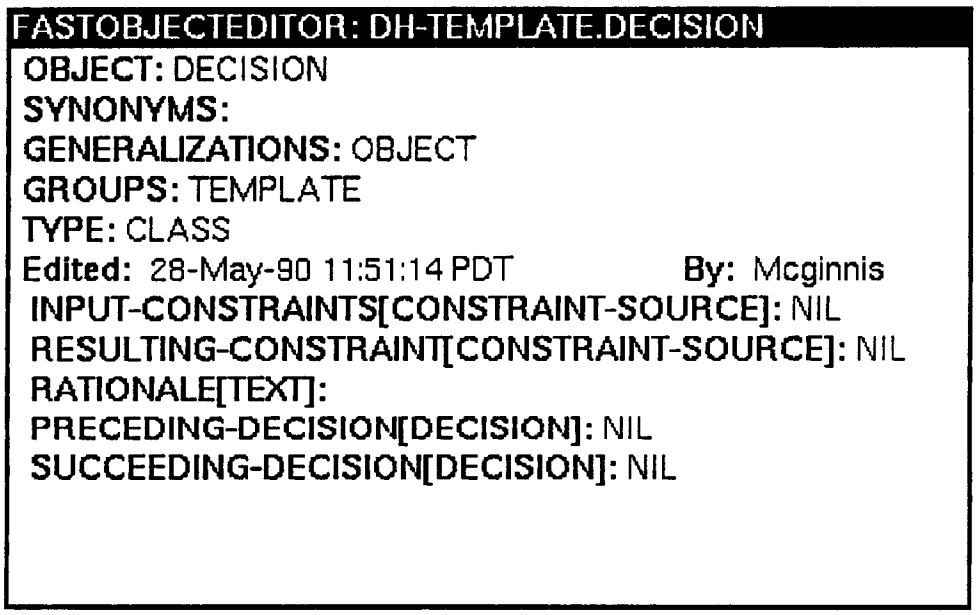

Figure 28 Decision Class Frame

object, see Figure 28. The decision representation does not require any computational or manipulative capabilities. The three primary slots of are the **input-constraint**, **rationale** and **resulting-constraint** slots. The input constraints are that subset of the existing active

constraints that were used or considered during a particular decision-making process. The active constraints are those constraints that define the current state of the design. The rationale slot is used to contain a text string describing the reasoning process behind the decision. The resulting constraint slot is used as a pointer to the new derived constraints, which were generated by that particular decision. By noting these input and resulting constraints, the propagation of the constraints can be followed.

The other slots of the decision object are the **succeeding-decision** and **preceding-decision** slots. These slots point to the previous and following decisions in the overall design process. This is the way that the temporal sequence of the design decisions is represented.

There are two types of constraint dependencies in the design representation.

1. Direct dependencies, which are gained by inspecting the **in-constraint** and **out-constraint** facets of the design object parameter slots.

2. Indirect dependencies, which are noted by examining the propagation of the constraint through the input and resulting constraint slots of the decisions.

It is important to note these two types of constraint dependencies, since they affect the design differently. For example, suppose the value of constraint, C1, was directly dependent on the value of another constraint, C2, via the **in-constraint** and **out-constraint** facets of the design object parameters. A change in C2 would require a change in the value of constraint C1. If on the other hand, constraint C1 was indirectly dependent on C2 through the input and resulting constraint slots of a decision, a change in C2 may not necessarily warrant a change in constraint C1. This separation allows a user to determine what affects changing the value of any particular

constraint will have on the overall design.

5.2.3 Constraint Implementation

The constraints in the knowledge base contain the basic information describing the state of the design. New information is added to the knowledge base in the form of constraints. Constraints are either generated by a decision process as is the case with derived constraints, or they are brought into the design as given or introduced constraints. All the constraints are causal, as mentioned previously, in that there is a dependent variable whose value is simply instantiated or is determined from one or more independent variables. The values specified by the constraints are tied to the design object parameters by the in- and out-constraint facets of the design object parameter slots.

Again, because an object-oriented programing environment was used, an almost direct mapping of the constraint representation into the implementation was achieved. Each of the three aspects of constraint information discussed previously is implemented as a fundamental class object from which the constraint instances are descended. These objects are **constraint-source**, **constraint-role**, and **constraint-language**. All constraint instances created within the knowledge base must have three primary ancestors or generalizations. Each primary generalization must be one of the subclasses or descendants of a fundamental constraint class object (i.e., source, role, language). Each fundamental class and its subclasses contain specialized slots and functions that enable them to behave or perform in a distinct manner. This specialization of and distinction between the different aspects of constraint information allows for the modelling of many different types of constraints. Therefore to understand how any particular constraint is

modelled, an understanding of each of the three constraint aspects must be obtained.

5.2.3.a Constraint Source

The source of a constraint identifies where the constraint originated. The three subclasses of constraint source are **given-constraint**, **introduced-constraint**, and **derived-constraint**. Each constraint instance must have as one of its primary generalizations or ancestors one of these three constraint sources.

The **constraint-source** class object contains **status**, **originating-decision**, **time-code**, and **actual-text** slots. The **status** slot denotes the status of the constraint and can be specified by one of four possible values accepted,

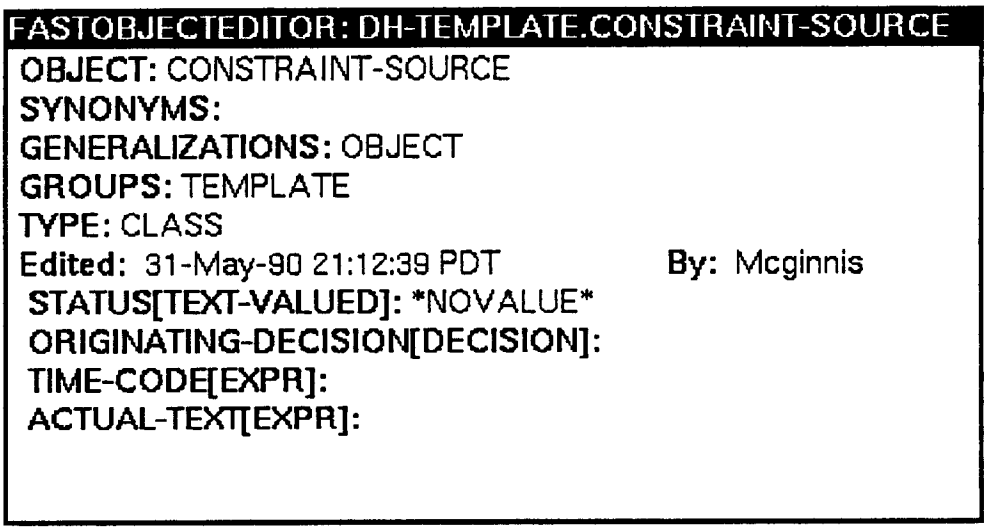

Figure 29 Constraint Source Class Frame

rejected, suspended, or nil, see Figure 29. A nil value indicates that the constraint was generated but was never subjected to an evaluation decision by the designer. The values of accept, reject, or suspend give the outcome of an evaluation decision. If a constraint is rejected by an evaluation decision, it is marked as inactive. This inactive status means that this constraint is no longer

considered in the active design space and subsequently it is not evaluated or solved for when trying to determine the value of a design object attribute (see section 4.2.1). The **originating-decision** slot contains a pointer to the decision from which the constraint originated.

The **time-code** slot contains a numeric time code denoting when during a design protocol the constraint was stated. A typical time code is 2035.23, which states that the constraint was stated 35 minutes and 23 seconds into the second video tape of the design protocol. The **actual-text** slot contains the actual text string stated by the designer in the protocol or it may contain a reference to one of the designer's original drawings. These slots are not an integral part of the basic constraint implementation, but are aids in this research for identifying the constraints in the design protocols from which they were extracted.

Given and introduced constraints have an additional **source** slot that denotes the origin of that constraint. The **source** slot contains a simple text string that explains where the constraint came from (i.e., design specifications, other designers, domain knowledge, design textbooks).

5.2.3.b Constraint Role

The constraint role determines what attributes of the design objects are affected by the constraint. The class object of **constraint-role** has been decomposed into subclasses according to the attribute affected by the constraint, see Figure 30, which is an expansion of part of Figure 22. The constraints are generated from only the leaf nodes subclasses. New subclasses can be easily added to this tree as required.

Related closely to the constraint role is the constraint language. Certain constraint roles are

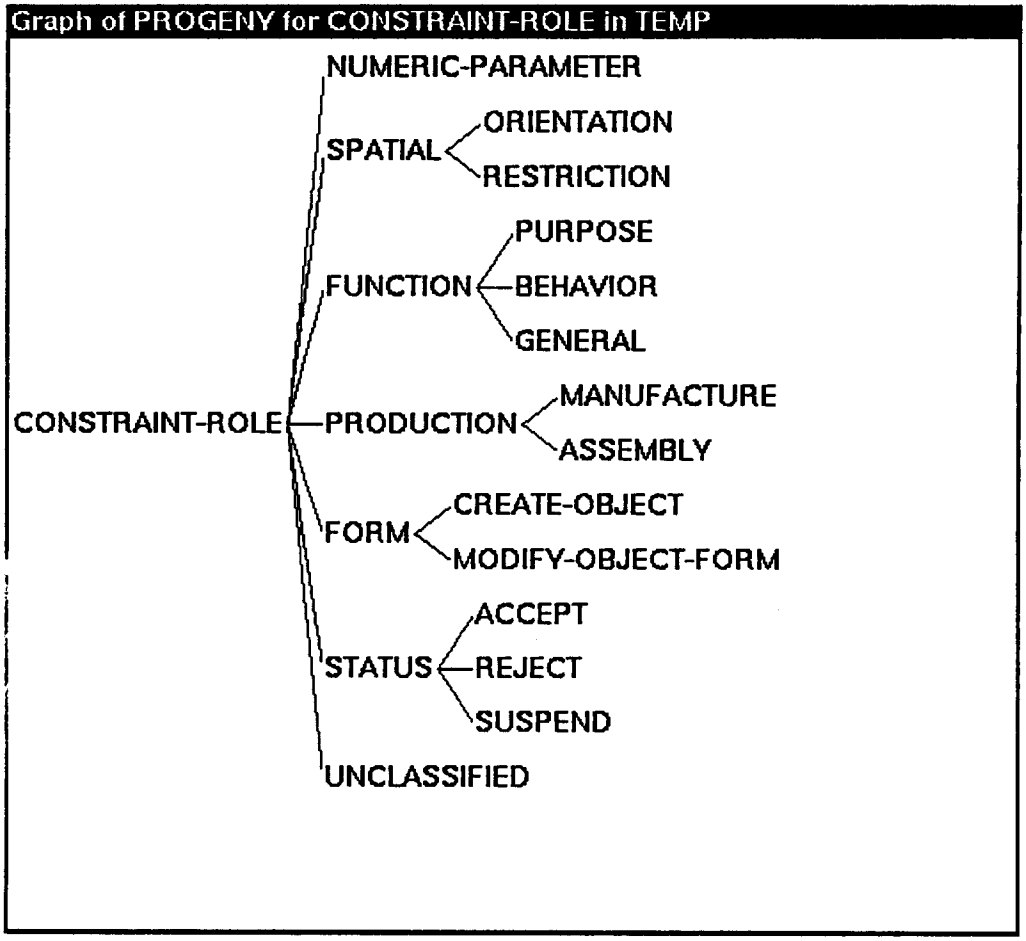

Figure 30 Constraint Role with Subclasses

expressed exclusively with a particular constraint language or constraint language subclass. Figure 31 presents a matrix of the standard constraint role - language possibilities.

Because of the close mating between constraint role

and language, a **default-language** slot has been added to the **constraint-role** class and subclasses. This slot tells which constraint language is to be used for the particular constraint role. In this way, the constraint language can be specialized for specific constraint roles.

The **numeric-parameter** role is groups together all of the numeric-valued design object attributes (i.e., length, weight, volume, stress, deflection). Because there exist a great multitude of possible numeric attributes and interactions between them, no attempt was made to decompose this role.

The **spatial** constraint role is used to define the spatial constraints on the design objects of the design.

Role – Language Relation

	Equality	Inequality	Quality	Conditional	Other	Spatial-Orientation	Spatial-Restriction	Structured	Simple	Special
Numeric Parameter	X	X	X	X	X					
Spatial						X	X			
Function								X		X
Production								X		X
Form										X
Status										X
Unclassified									X	

Figure 31 Role - Language Relation

It has been decomposed into the **orientation** and **restriction** subclasses. Constraints descended from the

orientation constraint role are used to describe the spatial orientation of the design objects. When solved, these constraints specify the x,y,z translations and rotations of the design objects.

The **restriction** constraints describe areas of the physical design space that cannot be occupied by the design objects. Since the design history tool did not require a restriction constraint solver, none was implemented, although a solver could be added if it is later deemed necessary.

The **function** constraint role represents the functional characteristics of the design objects. The three subclasses are **purpose**, **behavior**, and **general**. The **purpose** and **behavior** objects stipulate what functions the design object is to perform and how the object is to perform them, respectively. For any given design object, there may exist multiple purposes and behaviors. For example, the purpose of a battery contact is to hold the battery in place as well as to provide a electrically conductive path. For this reason, multiple **purpose** or **behavior** slots may exist within purpose and behavior constraints, respectively.

Within these purpose and behavior slots are pointers to **function-modules**, which describe a functional value or relationship. By using separate objects to contain basic functional information, design objects can share functionality. For example, if a specific function module describing the function of "supplying electric energy" is created, that module may be shared as a purpose of a battery, generator, or solar cell. Another benefit in isolating the functions is that they can be created and decomposed without regard to any specific design object.

The **general** function constraint role subclass describes the creation or modification of a function module without regard to any particular design object.

For example, the function of "supplying electric energy" stated above, can be decomposed into "generate energy" and "transmit energy" without any regard to what design object was generating or what object was transmitting.

Function modules are the basic building blocks of design object functionality. The function modules are generated from the descendants of the textual language subclass object **structured** and of the **function-module** class object. A typical function module instance can be

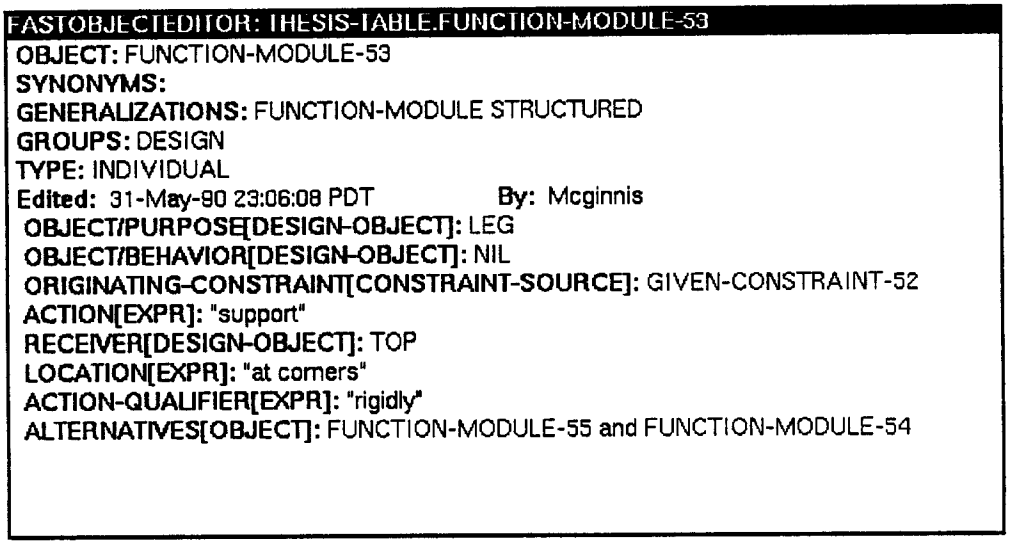

Figure 32 Function Module Instance Frame

found in Figure 32. Most of the slots shown are inherited from the **structured** generalization, which is discussed in further detail below under textual constraint language. The slots that are inherited from the **function-module** object and are particular to the function module instances are **object/purpose**, **object/behavior**, and **originating-constraint**. The **object/purpose** and **object/behavior** slots for a specific module point to objects that have this particular function module instance as their purpose or behavior. These slots act as back pointers to the design objects that refer to this particular function module. The **originating-constraint** slot points to the constraint

that originally created the function module. Function modules can be generated by purpose or behavior of design objects. They can also be created independently of any design object in the **general** role subclass.

The **production** constraints describe how a design object is to be manufactured and assembled. As a result, this role has been decomposed into **manufacture** and **assembly**. Similar to function constraints, production constraints contain multiple slots, called **step** slots. Each **step** slot contains a text string that describes a manufacture or assembly method to be used in the production of that particular design object. By having the option for multiple production steps, an assembly or manufacture method can be decomposed into a series of instructions. For example, a manufacture constraint of a table can contain the values "made by hand" and "built in house".

The **form** constraint role describes the creation and modification of design object instances. The two subclass of this are **create-object** and **modify-object-form**, which make a distinction between when an object is created and when an object is modified. These constraints change the design space by creating new objects or changing objects by specifying the name, shape, parent, or components of a new or existing objects.

Status constraints record the fact that a constraints has been evaluated in a decision and that the result of that decision was to **accept, reject**, or **suspend** the specified constraint. The only slot that resides in the status constraint is the **status** slot. This slot contains the value to be associated with the specific status role subclass, namely inactive, active, or suspended, respectively.

The last constraint role object is that of **unclassified**, which is used for constraint types that have

not yet been classified. These constraints cover design object attributes such as color, texture, appearance, safety, usability and so on. They are a catch all for any constraint that are not currently categorized.

5.2.3.c Constraint Language

The third aspect of information to be specified in the construction of a constraint is the **constraint-language.**

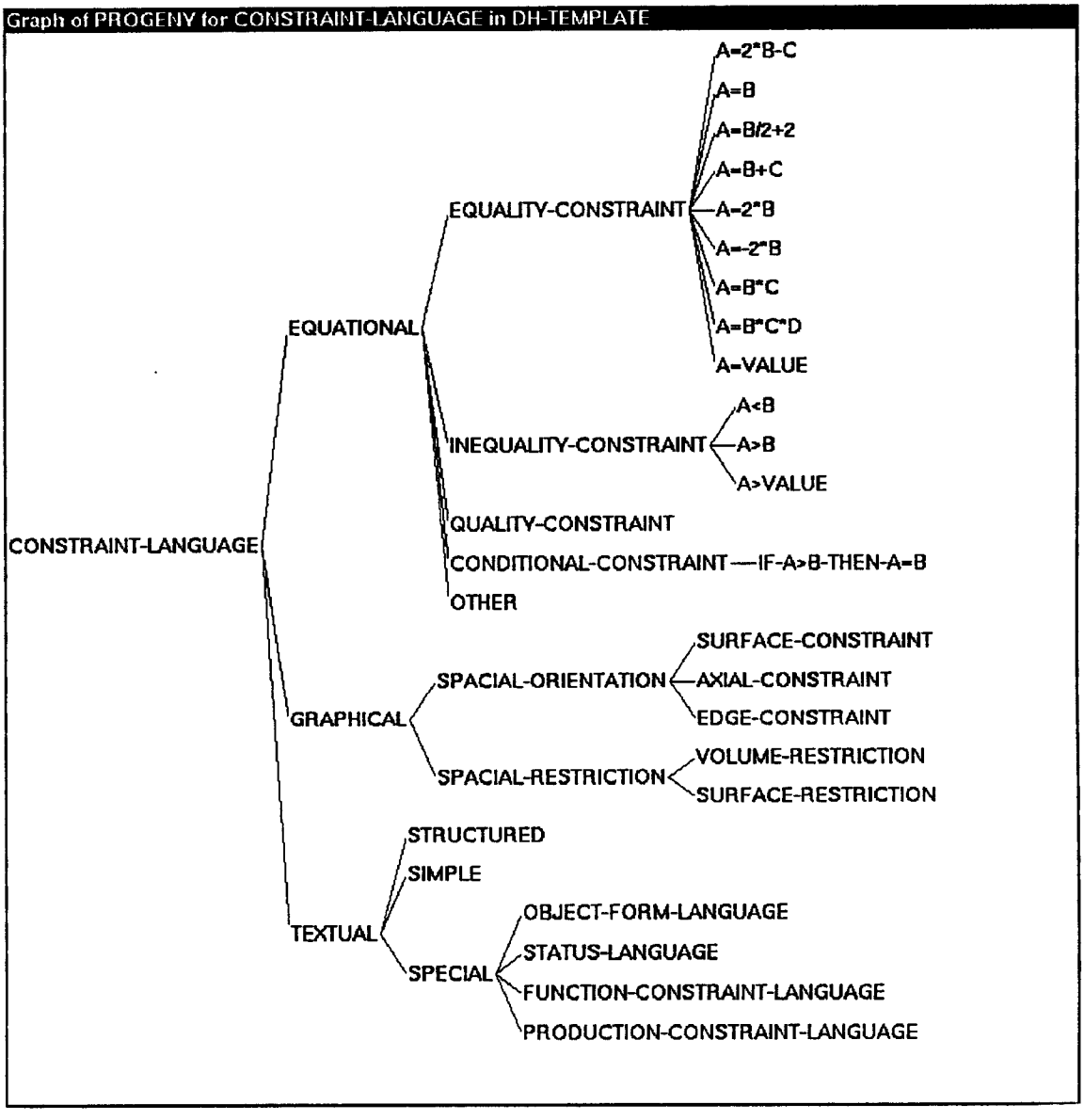

Figure 33 Constraint Language with Subclasses

The constraint language and its descendants contain the basic representational expressions of the constraint, see Figure 33. A specific constraint language is specified by choosing a leaf node subclass of the constraint-language tree. As with constraint roles, additional constraint languages can be added as necessary.

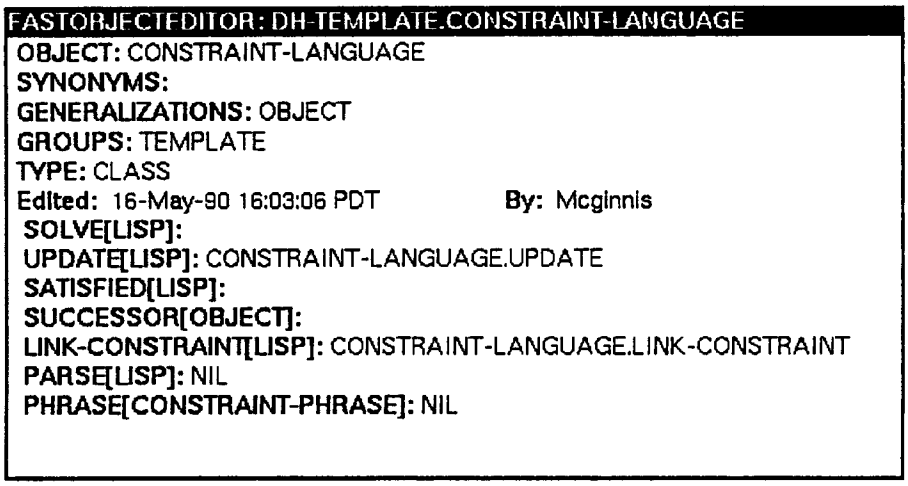

Figure 34 Constraint Language Class Frame

Associated with the constraint languages are the solvers of the constraints. The constraint solvers are needed in the design history only for the purposes of generating graphical images. Because of this requirement, only simple constraint solvers are used, as was discussed in section 4.2.1. These solvers enforce the constraint expressions stated by the constraints. For example, the expression "A=B+C", when solved, returns the sum of the argument values of B and C. The **constraint-language** class object contains a **solve** slot that is inherited by its subclasses and that indicates which specific solvers to use for each language subclass, see Figure 34. For example there is a specific solver for the equality constraints that works differently from the solver of the inequality constraints. In this way, different languages can be solved appropriately and new solvers can be added as required.

In object-oriented programming, in HyperClass, the names of LISP procedures can be given as the values of slots. These LISP procedures are called "methods", and they are executed by sending a "message" to the object slot. The constraint solver for each constraint is implemented by a LISP method , whose name is given in the **solve** slot of each constraint. The **solve** slots are inherited from the appropriate constraint language subclass.

Related to the solve slot are the **update** and **satisfied** slots. The method named in the **update** slot can be applied to update the constraints when design object attribute values are changed. For example, take the constraint **A=B+C**, where A, B, and C are all bound to particular design object attributes. If a change were to occur in the value of the attributes bound to B or C, then the constraint A=B+C can be resolved using that new attribute value. This updating of constraints ensures the consistency of the constraints. It can also be used for redesign purposes, when a user would like to see the effect of changing an attribute's value. This update facility is analogous to a truth maintenance system, where the constraints are the nodes of the network. The update facility is currently disconnected, because it was not required for the purpose of the design history tool. It can be reintegrated into the system by sending an update message to all new constraints as they are entered into the knowledge base. This constraint updating is the first step toward automatic redesign.

The method contained in the **satisfied** slot can be applied to determine if a particular constraint has been violated. This function sets the values of the constraint arguments to the current values of design object attributes to which they are bound. The constraint is then evaluated with these new values and a value of either

true or false is returned indicating whether the constraint is satisfied or not. This method differs from the **solve** method in that it does not solve for the dependent variable, but instead determines if the equation stated by the constraint still holds true.

Other slots residing in the language subclasses are **successor**, **link-constraint**, **parse** and **phrase**. The **successor** slot indicates which constraint (if any) succeeds the current constraint. Often when a constraint is made, it is simply a modification of a previous constraint. In these cases, rather than create a completely new constraint, the constraint is linked to the old constraint via the **linksup** and **linksdown** facets of the new constraint's slots. The successor slot of the old constraint is then set to indicate which new constraint has superseded it.

The **link-constraint** slot contains a specialized method that is invoked upon the creation of all new constraints. This method links the in- and out-constraint facets of the design object attributes to the constraints that use or specify those attributes. This method is invoked only once when the constraint is first introduced into the knowledge base.

The **parse** slot names a method for generating text phrases from the constraint instance. In Figure 35 for example, the constraint **A=B+C** in which **A** is bound to the table height, **B** is bound to top height, and **C** is bound to the top height, is parsed into "table height = leg height + top height". Similarly, a specific graphical constraint can be parsed into "table top-surface coplanar to computer bottom-surface". This parsing of constraints allows a designer to examine the constraints in a more easily understood manner. The **phrase** slot is the slot that contains the result of this constraint parsing, see Figure 35. In the event that the constraint cannot be easily

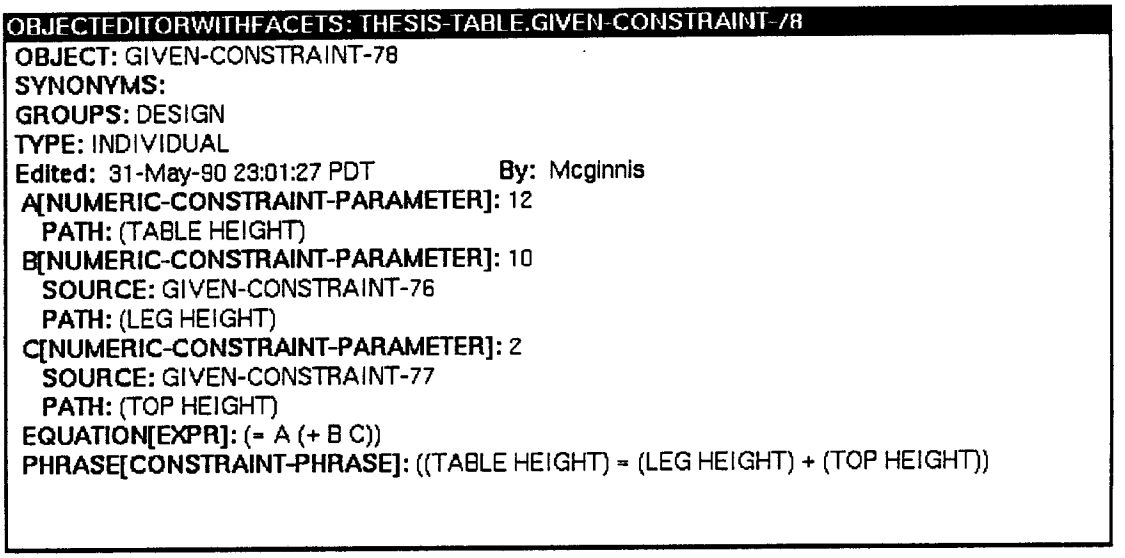

Figure 35 Equality Constraint Instance Frame

parsed, the phrase slot can be specified by directly imputing a text string.

All the constraint expressions are modelled in the same manner, with the major difference between them being the type of arguments and operations used. The constraints are implemented based on the syntactic structure mentioned previously, that is there are dependent and independent feature(s) connected by some relation or instantiation. A constraint expression will contain one or more constraint arguments (A,B,C), see Figure 35. These arguments are related to each other by some set of language operators (i.e., =, +, coplanar, rotates about,...etc.). The language operators employed depend on the constraint language specified. The argument slots of the constraint object each have associated with them facets or sub-slots. The facets specific to these constraint arguments are **path**, **role**, and **source**.

The **path** facet of a constraint argument is used to bind that argument to a particular design object parameter. For example, a path facet for an equational constraint is (gripper width) or (pedestal load), for a graphical constraint (flipping-frame front) or (plate x-

axis), and for a textual constraint (outer-frame behavior) or (pedestal appearance).

The **role** facet is used to distinguish the constraint argument as being dependent or independent. A role facet is set as one of two values: in or out. The in value signifies that an argument's value is brought into the constraint and is therefore independent. The out value states that an argument's value is calculated or generated within the constraint and is considered to be the dependent argument. The information in the **path** and **role** facets is used to link the constraints to the **in-constraint** and **out-constraint** facets of the design object attributes that are being used or specified.

The **source** facet of a constraint argument (see Figure 35) points to the original constraint from which the argument retrieved its value. The source facet appears and is used only by the independent argument of the constraint expression. Where the path facet points to the design object attribute bound to the constraint argument (A,B,C), the source indicates what particular constraint was specifying that attribute at that time. For example, in Figure 35, argument **B** is bound the path facet to the (leg height). At the time this constraint was constructed, the value of (leg height) was 10. This value of 10 for (leg height) was specified by a constraint in the **in-constraint** facet of that attribute, and the constraint that specified that value was **given-constraint-76**. In this way, as the value of the (leg height) attribute changes along with the in-constraints specifying it, the constraint argument using that specific value can determine from which specific in-constraint it was specified. This use of the source facet in the independent arguments of the constraint directly indicates what constraint was involved in that argument's value (i.e.,, B = 10).

Using a standard method for the constraint implementation provided a consistent and reliable means of entering and creating the constraints. Although the arguments of all constraints were modelled in a similar manner, the argument types, language operators used, constraint solvers, and constraint parsers are different for each constraint language subclass. The specialization of the constraints by constraint language allowed for a variety of constraint expressions to be modelled. It also enabled the implementation to assimilate new constraint languages or update old ones as needed. The following is a discussion of the constraint languages used, their argument types, the operators available, and the solving techniques used if any.

The **equational** language is used to express constraints that specify or relate a numeric quantity to a design

Typical Constraint Equations

Constraint Language	Expressions	Lisp Equation Used
Equality	A = Value	(= A Value)
	A = B + C	(= A (+ B C))
Inequality	A > Value	(> A Value)
	A < 2 * B	(< A (* 2 B))
Qualitative	A >> Value	none
	A ~ Value	none
Condition	IF A > B THEN A = B	(if (> A B) (= A B))
Other	A = result of FEA	none
	A = table lookup	none

Figure 36 Typical Equations

object parameter. Therefore, they are used exclusively by the constraint role **numeric-parameter**, which has "equational" as its default language. The arguments of the equational constraints should always be bound to the numeric valued parameters of the design objects, it does not make sense to specify the color slot of a design object as being "<6". The equational language arguments have both the role and path facets mentioned previously.

All the equational language descendants contain an **equation** slot that is used to calculate the dependent argument's value. This slot contains a lisp expression that is evaluated along with the constraint arguments and appropriate values to produce the constraint's output value. A list of typical equations for the different equational languages can be found in Figure 36. These equations can be selected from preexisting constraint class objects, **A=B+C**, **A<2*B**, or can be specified by the user when building new constraint classes as needed. For example, if a particular equation regarding the stress of an I-beam was not already present in the constraint class set, it is created by the user and then automatically added to the existing constraint set. The equational language is currently subdivided into the **equality**, **inequality**, **qualitative**, **conditional**, and **other** subclasses.

The **equality** constraints specify exact values for design object parameters. The main criterion for this equational constraint subclass is that it specifies an exact value as opposed to a general value or range of values. The primary operator used by the equality constraints is the equality (=). As with all equational constraints, other algebraic operators can also be used in association with the primary operator (i.e., +, -, /, *). One of the more common and most basic of the equality constraints is **A=VALUE**, in which a design object parameter

is instantiated or set to a specific numeric value. More complicated constraints can exist, and the user is only limited by the extent of computer code needed to solve the constraint equation. As mentioned earlier, the constraint subclasses are not finite, and new constraints can be easily added as necessary.

The **inequality** constraints are very similar to the equality constraints, with the biggest difference being the primary relational operators used. The operators used by the inequality constraints are the inequality operators (i.e., <, >, <>). The other main difference is the solver used for this constraint subclass. Where the solver for equality constraints returns the value calculated from the constraint equation, the inequality constraints return an expression that is derived from the constraint equation. For example, the inequality constraint **A<B+C** has an equation slot valued with the lisp expression "(< A (+ B C))". Assuming B=4 and C=5 the inequality constraint solver returns the expression "<9" as the solved value of this constraint. This solver is somewhat rudimentary in its treatment of inequalities, but it is adequate for the purposes of the current design history. If this implementation was to be used for a design tool, a more advanced solver could be used to solve sets equations simultaneously. Such a solver could return an exact value instead of the current inequality expression (i.e., "<4").

The **qualitative** constraints are the most abstract of the equational language constraints. These constraints relate the value of design object parameter to some subjective phrase, such as "as close as possible", "short", "want the maximum". Although the current constraint implementation does not incorporate a constraint optimizer, the use of a qualitative constraint subclass gives the capacity for this future enhancement. The solver for the qualitative constraints is similar to

the one for inequality constraints in that it returns an expression that expresses the essence of the constraint. For example, the constraint **A>>VALUE** states that argument **A** should be as large as possible in relation to **VALUE**. If **VALUE** was set to 10, the constraint returns the expression ">>10" when solved for.

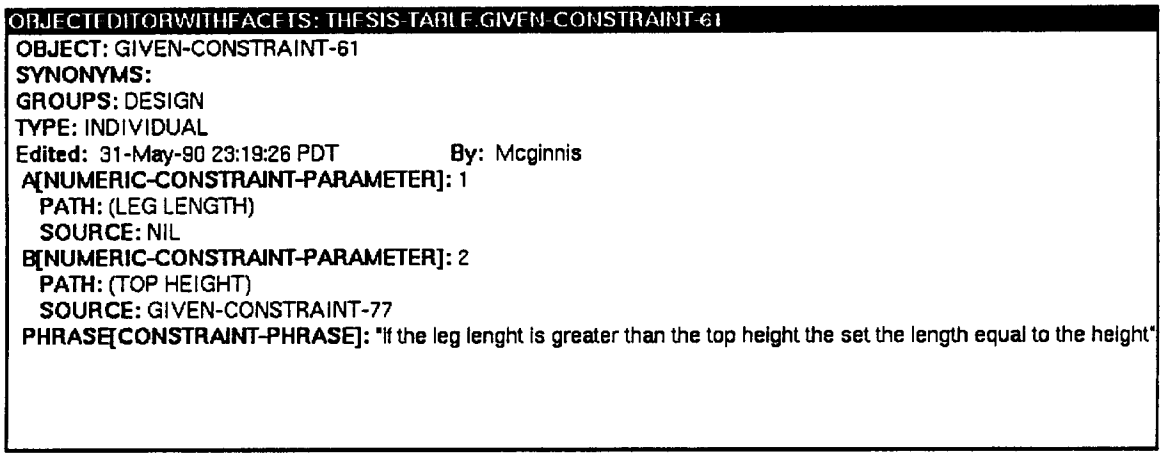

Figure 37 Conditional Constraint Instance Frame

The **conditional** constraints are used when the value of the constrained parameter is determined by some conditional (if-then) relationship. For example, a conditional constraint is "if the flipping frame hits the front of the tank then shorten the lever arm". This constraint is dependent on the behavior of the flipping frame and affects the length of the lever arm. These constraints are different from the other equational constraint subclasses in that the value of the dependent argument is contingent on a condition. These constraints are expressed in terms of if-then statements. Ideally the constraint solver should be able solve whatever conditional statement is residing in the equation slot of that constraint. Unfortunately not all conditions can be easily calculated, such as the flipping frame behavior stated above. Therefore the actual value returned by the solved constraint is the value stated by the original designer and the actual conditional statement resides in

the **phrase** slot of the constraint, see Figure 37. In this way the constraint behaves as an equality constraint when solved for but actually contain the conditional statement made by the designer.

The last equational constraint subclass currently recognized in the implementation is the **other** class. This subclass deals with those constraints whose resulting value cannot be coded into a coherent or easily solved for expression. Examples of these constraints include attribute values that were extracted from graph or table lookups or values that were derived from an analytical or numerical integration or differentiation. Like the conditional constraints, these constraints behave as equality constraints because they specify an exact value, but their true constraint expression resides in the **phrase** slot.

This set of subclasses for the **equational** constraint language is not exhaustive. New types of equational constraints will be added as they are identified. The subclasses presented here are only representative of the constraints extracted from the protocol data. For example, an additional constraint subclass might be finite element analysis, in which the **equation** slot of the constraint designates some FEA code or program to run in order to evaluate that constraint. All of the constraint languages are openended in this way, and this enables the implementation to expand as is needed.

Graphical, the next basic constraint language, is used to express spatial relationships or restrictions. Consequently there are two different subclasses for this language: **spatial-constraint** and **restriction-constraint.** The spatial relations are dependent on the topology of the objects to be related. For example, a 2-dimensional topology would involve of points, lines, and curves. Therefore the expressions needed to define these would

include parallel lines, line through a point, line tangent to a curve, and so on. The topology used by the designer was 3-dimensional, so design objects were constructed from solid geometries (CSG). This topology involves surfaces, axis, and edges. Some expressions needed for this topology include parallel and perpendicular surfaces, co-axial axis, same edges, or co-linear edges (edges that are collinear but can rotate relative to each other). Therefore the spatial constraint subclasses are divided according to the topology referred to by the constraint **surface-constraint**, **axial-constraint**, **edge-constraint**, and **combined.**

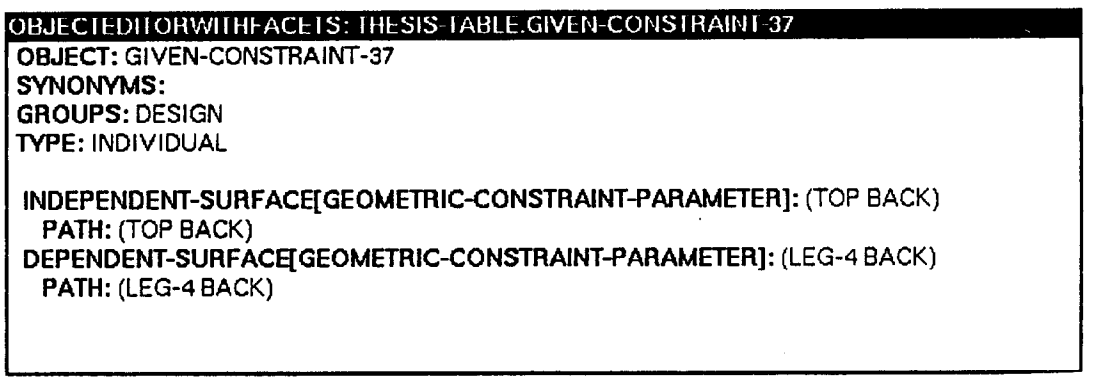

Figure 38 Spatial Orientation Instance Frame

The individual implementation of these subclasses is very similar and therefore will be explained as a single group. Each constraint has dependent and independent topologies, that are surface, edge, or axis, see Figure 38. These constraint slots are bound to the design object topologies to be related. The constraint is then enforced or solved for by sending a message to the constraint to assert itself. This assert method then sets the orientation of the dependent topology and subsequent design object in accordance with the location of the independent topology and the prescribed relation.

The spatial constraints are solved for by examining the design objects and determining the dependencies of

constraints affecting the x,y,z translations and rotations. This results in an ordered list of constraints that are solved for or asserted, in order to produce the correct orientation of the objects. The ordering of these constraints by the computer is important to the final orientation result. The assert method is used since these constraints are solved in an ordered set, as opposed to the regular constraints which are solved for on an individual basis. This allows spatial constraints to be stated in any chronological order as long as the constraint dependencies make sense (i.e., block-a's location is dependent on block-b's location, which is dependent on block-a's location,.... and so on).

The other subclass for the graphical language is the **restriction-constraint** subclass. These constraints are used to express limitations or restrictions on where the design objects can be located. These restrictions are expressed in terms of volumes not to enter, such as "leave the side areas of table free" or "do not enter tank in these areas (from specification drawing)". Other expressions affect surfaces, like "only 1/4" of the plate periphery can be handled" or "do not touch 1 1/2" from table back edge". The restriction constraints are therefore subdivided into **volume-restriction** and **surface-restriction**.

The **volume-restriction** constraints refer to volumes not be entered or violated. Therefore these expressions simply state a volume that is not to be occupied (e.g., "table side areas"). These constraints have an **area** slot that contains a description of an area or volume that is not to be entered.

The **surface-restriction** constraints are represented in terms of a surface **reference**, a **distance**, and **direction**, see Figure 39. The reference is the object surface from which the restriction originates (i.e., "the plate

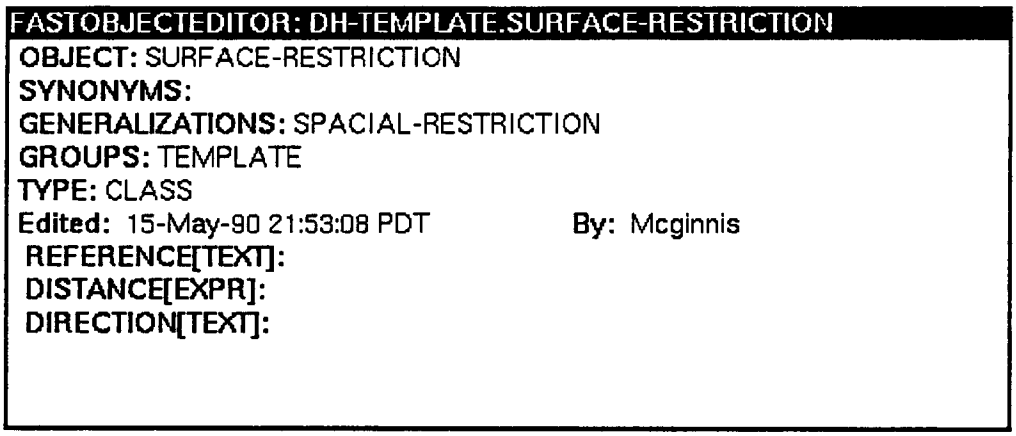

Figure 39 Surface Restriction Class Frame

periphery" or "the table back edge"). The distance is the measure from that reference, and the direction is a qualifier on that distance. The restriction constraints are currently not solved for or asserted, but a restriction constraint checker can be later added if desired.

The last basic constraint language, **textual**, expresses constraints that do not directly relate to the numerical or geometric properties of the design object. There are currently no solve methods for these constraints. The solved values returned by these constraints are the results of the **parse** method discussed previously. These constraints have been subdivided based on the representation scheme used: **structured**, **simple**, and **special**. The implementations for the **structured** and **simple** constraints are mapped directly from the representational forms discussed earlier. The only difference is the addition of the **parse** slot, which when used creates phrases from the constraint.

The **structured** constraints model expressions that are stated in the form of a declarative sentence. This language contains slots for the object, action, receiver, location, and action-qualifier arguments discussed earlier. The object and receiver slots are specified by

pointers to design objects; the remaining slots are

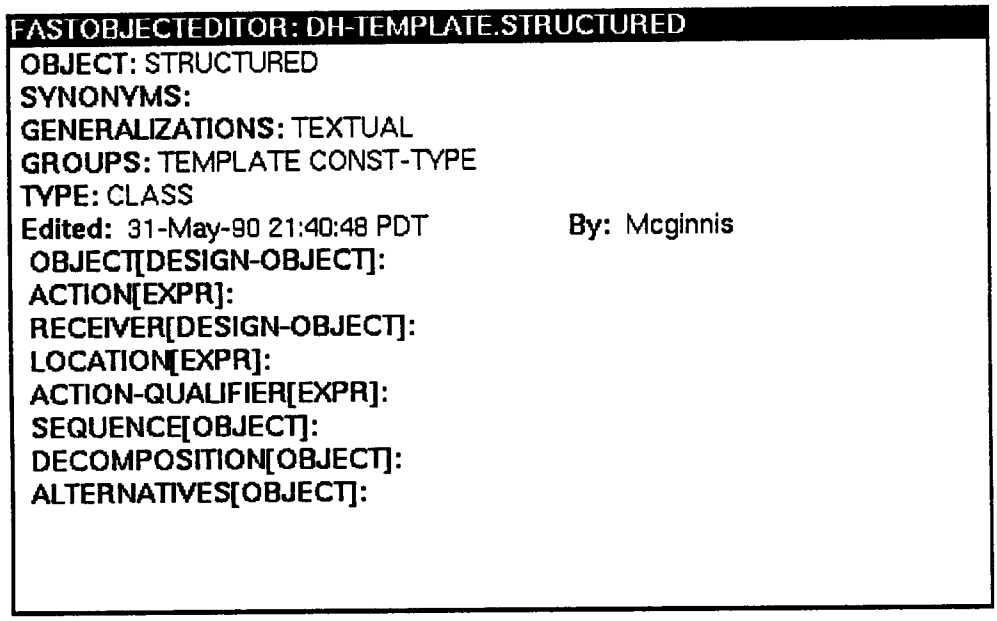

Figure 40 Structured Language Class Frame

specified by text strings, see Figure 40. Additional slots contained in the **structured** language object include the **sequence**, **decomposition**, and **alternative** slots. These slots can contain a series of other structured language constraints. These series slots are used if the stated structure is further broken down into either an ordered sequence, decomposed elements, or alternative options, respectively.

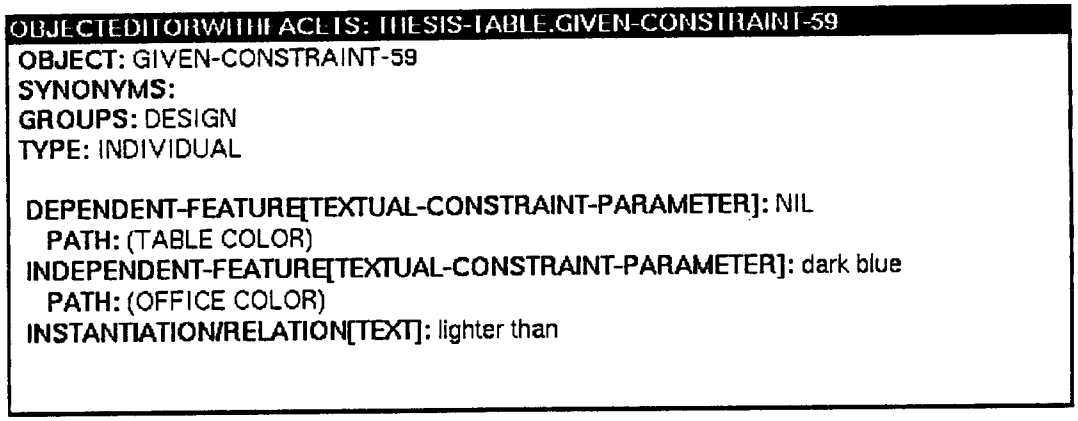

Figure 41 Simple Language Instance Frame

The **simple** language is used for constraint expressions that are not expressed in as structured declarative form. This language contains a **dependent-feature**, **independent-feature(s)**, and **relation/instantiation** slot, see Figure 41. The feature slots are specified by pointers to design object parameters. The relation or instantiation slot is specified by a text string.

The **special** constraint subclasses currently includes the **object-form-language**, **status-language**, and **function-constraint-language**. These subclasses are used for specific constraint roles and enable the language to become specialized for certain constraint roles. The **object-form-language** is used to create constraints affecting the basic form and configuration of the design objects. These constraints have slots for the name, shape, parent, and components of a design object, see Figure 42. When solved for, these constraints either create a new design object or modify an existing one.

The **status-language** constraints, used exclusively by the **status** constraint role, select a design object parameter and its related constraint for a status change.

FASTOBJECTEDITOR: THESIS-TABLE.GIVEN-CONSTRAINT-74
OBJECT: GIVEN-CONSTRAINT-74
SYNONYMS:
GENERALIZATIONS: GIVEN-CONSTRAINT CREATE-OBJECT OBJECT-FORM-LANGUAGE
GROUPS: DESIGN
TYPE: INDIVIDUAL

NAME[EXPR]: TABLE
SHAPE[DESIGN-PRIMITIVES]: COMPOSITE
PARENT[DESIGN-OBJECT]: OFFICE
COMPONENTS[DESIGN-OBJECT]: TOP, LEG-1, LEG-2, LEG-3, and LEG-4

Figure 42 Object Form Language Instance Frame

This constraint language contains the slots **affected-**

feature and **constraint-affected**. The affected feature slot refers to what design object parameter is to be indirectly affected by the status constraint. The constraint affected slot is used to point to the constraint that is to have its status slot value changed. The constraint solver simply sets the value of the **status** slot in the constraint named in the **constraint-affected** slot (i.e., active, inactive, suspended). This can subsequently affect the value of the design object parameter that was set by that particular constraint. Since the parameter value is determined by finding the first constraint with a non "inactive" valued status from the list of constraints residing in the **in-constraint** facet of that parameter.

The last special constraint subclass is the **function-constraint-language**. The main reason for separating this constraint language out from the others was the need for a special user interface when creating the function constraints. This interface enables the user to specify the purpose or behavior of an object by specifying the appropriate **function-module**. If the function module does not currently exist or needs to be changed, the user is given the opportunity to do this. This allows for both the selection and modification of the function modules to occur together, which makes for cleaner, more efficient constraint creation.

5.2.4 Constraint Implementation Summary

Using the decendents of the three primary class objects, **constraint-source**, **constraint-role**, and **constraint-language**, a constraint instance is generated. By using these constraint instances to specify the attribute values of the design objects, the state of the design is represented. The use of the decision instances to generate and record the introduction of new constraints

into the design space captures the temporal process of the design. By keeping track of the decisions, constraints, and design objects of design, a history of the design is maintained.

6. CONCLUSIONS

The goal of this research was to develop and implement a representation for mechanical designs that not only represented the final state of a design but also the temporal intermediate states as well. The representation was developed from a design process model based on protocol data taken of mechanical engineers solving original design problems. The representation was implemented in HyperClass, an object oriented programming environment. The implemented representation succeeded in fulfilling the requirements identified from the constraint analysis and has been used to represent a subset of the constraints extracted from the design protocols.

6.1 Conclusion of Constraint Analysis

The focus of the constraint analysis was concerned with the development and propagation of constraints and features in the mechanical design process. This was accomplished by studying the constraints that affected the design of one part throughout the entire design protocol. By identifying and codifying these constraints, a better understanding of the necessary requirements for representing them was acquired.

A constraint classification was developed and evaluated that consisted of a constraint source, level of abstraction, and constraint structure. The structure developed for the constraints was based on feature relationships. These feature relationships were of the form:

[dependent feature]-[instantiation]
or
[dependent feature]-[relation]-[independent feature(s)]

From these basic structures, 10 different structure types

were generated. These structures were used to successfully model all the constraints identified in the design of the part examined. These structures were also an aid in identifying two of the important aspects of the constraint information, namely the constraint role and the constraint language.

The analysis performed resulted in substantial insight into constraint and feature generation and propagation in the process of mechanical design. By examining the constraint classification and structures used, the requirements of a mechanical design constraint representation were established.

6.2 Representation Conclusions

A design process representation has been developed that is capable of documenting the initial, intermediate, and final states of a design as well as the design processes that connects them. By modelling the design process as a series of design decisions, the representation captures the intent of the original designer. The design decisions result in new constraints which in turn instantiate or modify the values or relations between the design object attributes of the design.

The representation is composed of three fundamental concepts that interact together to form a history of the design process: the design objects, constraints, and decisions. The design objects, which represent the physical artifacts of the design, are comprised of a set of design object attributes. These attributes embody the properties, parameters, and characteristics of the design object that are of concern to the designer, such as length, weight, behavior, and appearance. The values and relationships among these attributes are specified by the constraints of the design.

The representation for constraints is composed of three aspects of the constraint information: constraint source, constraint role, and constraint language. The constraint source refers to the origin of the constraint, whether it was given, introduced, or derived. The role of the constraint determines what types of design object attributes are affected by constraints, such as the function, production, or numeric parameter. The language of the constraint is concerned with what form the constraint was expressed in, whether it was equational, graphical, or textual. Using these three aspects, all constraints observed in the protocol can be represented.

The design decisions are the processes by which new derived constraints are generated within the design space. These are represented by noting the inputs and results of the decision. By chronicling the decisions of the design, a history of that design can be recorded. The representation is capable of maintaining not only the process of the design but also the intermediate states of the design as well.

6.3 Current Capabilities of Implementation

The design process representation was implemented in HyperClass, an object oriented programming environment. The implementation is openended and allows for new constraint types or subclasses to be added as needed. The architecture used enables modifications and enhancement to the system to occur without difficulty. The current constraint implementation can be used to represent all the constraints found in the protocols examined. Not all the constraints modelled have computational meaning; some constraints simply state the value or relation expressed by the designer in textual terms.

The constraint implementation is sufficient enough to

generate accurate graphical images of the design objects. These graphical images are 2-dimensional projections of the 3-dimensional design. The images are incorporated in a browsing interface tool, which uses the images for the querying of the design.

The implementation is the basic foundation of the design history tool. This tool will be used to record, represent, and playback designs for the purposes of improved design communication. The design history may potentially aid in the process of redesign in that it will relate the designer's original intent of the design.

6.4 Future Recommendations and Suggestions

Although the current implementation is capable of representing all the constraints extracted from the protocol data, not all the constraints represented have computational meaning. Only a subset of the equational and graphical language constraints have solvers associated with them. To improve the implementation, constraint solvers can be created or incorporated for a broader range of the constraints. The current equational and graphical solvers can also be upgraded if a variational design tool is desired.

The representation used for modelling the decision object is rudimentary, in that it treats the decision as black block. This representation of the decision notes only what goes in and what comes out but does not concern itself with the actual workings of the decision process. To give more meaning to the decisions represented in the system, a more detailed decision representation needs to be developed. This new decision representation should be capable of relating the designer's decision rationale in more detail and with more structure. A structured decision representation could lead the way for an expert system, which would perform automatic redesign.

Another improvement to the implementation would be a refinement of the design recorder. Although the current design recorder is sufficient for purposes of entering protocol data, it is not user friendly. It allows no easy means of quickly inspecting the results of the decisions or constraints being entered. The pop-up menus of the recorder disappear after a selection made and it is easier for the user to become confused about where he is in the recorder. A more user-friendly interface that constantly showed the user where in the recording process he was and displayed the effects of decisions and constraints being generated, would greatly enhance the usability of the implementation.

The design process representation and implementation presented and discussed in this thesis is an attempt to formally model the process of mechanical design. The representation, based on constraints, covers a broad range of different constraint types. Rather than try to fully represent one particular type of constraint, the representation lays a foundation from which to model many different constraint types. It is hoped that this foundation can be used as the basis for future research projects in the area of mechanical design.

7. BIBLIOGRAPHY

Balakumar, P., J.C. Robert, R. Hoffman, K. Ikeuchi, T. Kanade, "VANTAGE: A Frame-Based Geometric Modeling System", Carnegie-Mellon University, 1988.

Brown, D.C., "Using Design History Systems for Technology Transfer", Proceedings of the MIT-JSME Workshop on Cooperative Product Development, November 1989.

Charon, R., "Development of a Graphical Interface to the Design History Tool", Masters Thesis, Department of Computer Science, Oregon State University, November 1989.

Chung, J.C.H., Schussel, M.D., "Comparison of Variational and Parametric Design", 1989.

Conklin, J., Begeman, M. "gIBIS: a hypertext tool for Exploratory policy Discussion", Proceedings of the Conference on Computer Supported Cooperative Work, Sept 1988, ACM, pp 140-152.

Cunningham, J.J. and Dixon, J.R., "Designing With Features: the Origin of Features", International Computers in Engineering Conference, San Francisco, CA, August 1988., p. 237.

Dixon, J.R., Duffey, M.R., Irani, K.L., Meunier, K.L., and Orelup, M.F., "A Proposed Taxonomy of Mechanical Design Problems", International Computers in Engineering Conference, San Francisco, CA, August 1988, p. 41.

Eastman, C., Bond, A., Chase, S., "A Formal Approach for Product Model Information", 1989.

Esterline, A., Arnold M., Riley D.R., Erdman A.G., "Representations for the Knowledge Revealed in Conceptual Design Protocols", Proceeding of the NSF Engineering Review, Jan 1990, pp 123-135.

Hwang, B., and Ullman, D.G., "The Design Capture System: Capturing Back-of-the-Envelope Sketches", International Conference on Engineering Design, Dubrovnik, August 1990.

Jones, J.C., Design Methods, Seeds of Human Futures, New York:John Wiley and Sons, 1980.

Kuffner, T.A., "Mechanical Design History Content: The Information Request of Design Engineers", Masters Thesis, Department Mechanical Engineering, Oregon State University, December, 1989.

Kuffner T., and Ullman, D.G. "How Designers Understand Existing Designs" To be Submitted to the 1990 Conference on Mechanical Design Theory and Methodology, Chicago, Sept 1990.

Lai, K., Wilson, W.R.D., "FDL - A Language for Function Description and Rationalization in Mechanical Design", 1987.

Lakin, Fred et al., "The Electronic Design Notebook - Performing Medium and Processing Medium", Research Report, Stanford University With NASA Ames Research Center, 1989.

Lakin,F. Wanbaugh,J. Leifer,L. Cannon,D. Sivard,C. "The Electronic Design Notebook: Performing Medium and Processing Medium", Visual Computer, pp.1-13, 1989.

Leifer, L., "Conservation of Design Knowledge During Design, Development and Validation of the Space Infrared Telescope Facility Tertiary Mirror Assembly", Proceedings of the NASA Ames Research Center Artificial Intelligence Forum, 18 Nov, 1987, pp 366-374.

Libardi,E.C.,Jr. Dixon,J.R. Simmons,M.K. "Computer Environments for the Design of Mechanical Assemblies: A Research Review", Engineering With Computers, 3, pp.121-136, 1988.

McCall, R.J. "MIKROPLIS: a Hypertext System for Design", Design Studies, Vol 10, Number 4, Oct 1989, pp 228-238.

McGinnis, B. and D. Ullman, "The Evolution of Commitments in the Design of a Component", International Conference on Engineering Design, Harrogate UK, Aug 1989, pp. 467-495.

Pahl, G., and Beitz, W., Engineering Design: a systematic approach, The Design Council, London, 1988.

Quiesser, A. H., "The Sketch Pad", Measurement and Manufacturing Systems Laboratory, Hewlett-Packard Laboratories, March 1989.

R.G. Smith, and P.J. Carando, "Structured Object Programming In Strobe", Research Note SYS-86-26, Schlumberger-Doll Research, October 1987.

R.G. Smith, M.F. Kleyn, and E. Schoen, "Impulse Reference Guide", Research Note SYS-47-41, Schlumberger Technologies Corporation, January 1988.

Rich, E., Artificial Intelligence, McGraw-Hill, Inc., New York, 1983.

Rinderle,J.R. Suh,N.P. "Measures of Functional Coupling in Design", Journal of Engineering for Industry, 104, pp.383-388, 1982.

Schoen, E., Smith, R.G., and Atkinson, A., "The Generic Window System", Research Note, Schlumberger Technologies, August 1988.

Schoen, E., R.G. Smith, and P.J. Carando, "Ruling with Class (Programming with Rules in Class)", Research Note, Schlumberger Laboratory for Computer Science, February 1989.

Shah, J.J., and Wilson, P.R., "Analysis of Knowledge Abstraction, Representation and Interaction Requirements for Computer Aided Engineerng", Journal of Design Studies, accepted for publication.

Stauffer, L. and D.G. Ullman, "A Comparison of the Results of Empirical Studies into the Mechanical Design Process", Design Studies, Vol 9, No 2, Butterworths Ltd., April 1988, pp. 107-114.

Stauffer, L., D.G. Ullman, and T.G. Dietterich, "Protocol Analysis of Mechanical Engineering Design," Proceedings of the 1987 International Conference on Engineering Design, WDK 13, Boston, MA, August 1987, pp. 68-73.

Stauffer, L.l "An Empirical Study on the Process of Mechanical Design", Doctoral Dissertation, Department of Mechanical Engineering, Oregon State University, Sept 1987.

Tikerpuu, J., "General Feature-Based Frame Representation for Describing Mechanical Engineering Design, Developed from Empirical Data", Masters Thesis, Department of Mechanical Engineering, Oregon State University, March 1989.

Tikerpuu, J. and D.G. Ullman, "Data Representations for Mechanical Design Based on Empirical Data", International Computers in Engineering Conference, San Francisco, Aug 1988, pp. 245-254.

Ullman, D.G., Dietterich, T.G., and Stauffer, L., "A Model of the Mechanical Design Process Based on Empirical Data", Artificial Intelligence in Engineering, Design, and Manufacturing. Vol 2 (1) 33-52. 1988.

Ullman, D.G., Dietterich, T.G., and Stauffer, L.A., "A Model of the Mechanical Design Process Based on Empirical Data: A Summary", Proceedings of the AI in Engineering Conference, Stanford University, Aug., 1988.

Ullman, D.G., Stauffer, L.A., and Diettterich, T.G., "Toward Expert CAD", ASME Computers in Mechanical Engineering, 1987, Vol. 6, No. 3, pp. 21-29.

Unger, M.B. and Ray, S.R., "Feature - Based Process Planning in the AMFR", International Computers in Engineering Conference, San Francisco, CA, August 1988, p. 563.

VDI 2221, "Systematic Approach to the Design of Technical Systems and Products", VDI-Verlag GmbH, D-4000 Dusseldorf, 1987

APPENDICES

Appendix I. Original Problem Statement

Our manufacturing company needs a machine to coat thin, aluminum "plates" for use in a mechanical lung machine. A thin chemical layer will be cast on the surface of a water bath for coating the plates. The machine needs to dip both sides of the plates into this chemical bath. We need you to design three of these machines. There is a large machine shop on the premises for building these machines in-house.

Specifically, the process for coating these plates will be as follows:

A worker loads the machine with a .063 x 10 x 10 inch aluminum plate (see Figure 1). Since the worker needs to load and unload these plates all day from a standing position, fatigue should be kept to a minimum.

The worker visually insures that the surface of the water is clean and then uses a syringe to eject a premeasured amount of chemical in solvent solution on the surface of the water. The chemical solution spreads as an oil slick over the surface. When the solvent evaporates (just a few seconds) the 500 Angstrom thick chemical layer is ready to be applied to the surface of the plate. The chemical is nontoxic and safe to handle.

The chemical is applied to the surface of the plate by gently lowering the plate onto the water where surface tension will cause bonding instantly. The plate should not go in more than half of its thickness. Once the plate is coated, it is moved away from the surface and the process is finished for that side of the plate. The excess chemical left on the surface of the water is cleared from the bath manually by the worker (the layer is very thin and sticky).

The process must be repeated to coat the other side of the plate. After coating both sides, the plate is then

presented to the worker for unloading. The entire process should take a maximum of 40 seconds.

There are a few constraints on the problem, namely:

(1) The plates can only be edge handled. Only the edge 1/4" around the periphery of the plate can be touched by either the worker or the machine at any time (see Figure 1).
(2) The water must be kept clean as any impurities can affect the integrity of the chemical. This is especially true of organic materials.
(3) Parts of the machine that hold the plate can enter the water, outside the periphery of the plate, to a depth of up to 1/2" as shown in Figure 1.
(4) It is anticipated that the machine will mount on the table surface in the areas shown in figure 2. The machine can not extend within 1.5" of the edge of the table.
(5) The water bath level is automatically maintained 0.5" below the surface of the table, plus or minus 0.01".

Appendix II. Features from Protocol

II.A. All features Observed

aluminum block
aluminum block location
back cross member
back cross member location
back cross member purpose
back pivot point
back pivot point center line
back pivot point elevation
back pivot point location
back pivot point manufacture
bearing location
bearing manufacture
bearing material
bearing purpose
bolt location
bolt type
bolt installation
bushing installation
bushing material
bushing purpose
bushing
bushing location
chemical
chemical bonding
chemical layer
chemical toxicity
cross member
cross member purpose
detentes
detentes purpose
diagonal cross member
diagonal cross member
diagonal cross member location doubler
location
doubler manufacture
doubler purpose
doubler quantity
doubler size
doubler thickness
doubler
finger
flipping frame
flipping frame back cross member thickness
flipping frame back cross member purpose
flipping frame back cross member
flipping frame back cross member location
flipping frame cross member purpose
flipping frame cross members
flipping frame depth
flipping frame downward travel
flipping frame elevation
flipping frame front cross member location
flipping frame front cross member
flipping frame inner arm location
flipping frame location
flipping frame material
flipping frame movement
flipping frame operation
flipping frame orientation
flipping frame outer arm
flipping frame outer arm height
flipping frame outer arm location
flipping frame outer arm material
flipping frame outer arm purpose
flipping frame outer arm shape
flipping frame outer arm thickness
flipping frame racking
flipping frame rear portion
flipping frame rotation
flipping frame shape
flipping frame stability
flipping frame upward travel
flipping frame vertical travel
flipping frame weight
flipping frame width
framework
framework downward travel
framework material
framework orientation
framework side travel
framework stability
framework upward travel
front cross member
front cross member cross section
front cross member location
front cross member purpose
front pivot point
front pivot point arc
front pivot point center line
front pivot point diameter
front pivot point elevation
front pivot point elevation
front pivot point hole diameter
front pivot point location
front pivot point operation
front pivot point purpose
gripper
gripper location
gripper material
gripper operation
gripper shape
gripper shape purpose
gripper thickness
gripper tolerances
gripper width
guide rods
guides
guides operation
hangers
hangers purpose
knob
knob diameter
knob location
knob operation
knob purpose
knob shape
knob type
lever arm
lever arm bearing area
lever arm cross section

lever arm front extension length
lever arm length
lever arm location
lever arm longevity
lever arm moment
lever arm operation
lever arm purpose
lever arm rear
lever arm rear doublers
lever arm rear extension
lever arm rear extension length
lever arm rear outside doubler thickness
lever arm rear outside doubler size
lever arm shape
lever arm spring hole
lever arm vertical travel
lever arm width
lever arms orientation
machine
machine appearance
machine area
machine cleaning
machine design
machine installation
machine loading
machine longevity
machine manufacture
machine materials
machine mounting
machine mounting area
machine movement
machine operation
machine operation time
machine power
machine quantity
machine racking
machine rear
machine safety
machine stresses
machine surface
machine tubing material
machine tubing size
machine weight
operator
operator area
operator comfort
operator effort
operator fatigue
operator gripping length
operator hand
operator position
outer frame
outer frame inside width
outer frame material
outer frame orientation
outer frame racking
outer frame shape
outer frame torque
outer frame twisting
outer frame twisting movement
pedestal
pedestal base
pedestal base location
pedestal height
pedestal location
pedestal moment
pedestal width
pivot point hole quantity
pivot point holes
plate
plate coating
plate edge
plate loading
plate material
plate orientation
plate restriction
plate side
plate size
plate surface
plate thickness
plate weight
restricted area
restrictions
safety factor
shoulder bolts
shoulder bolts loading
shoulder bolts manufacture
shoulder bolts thread
sink location
sink rear edge
spring
spring hook
spring hook location
spring location
spring purpose
spring relaxed/stretched length
spring size
spring support holes diameter
spring support hole manufacture
tank free area
tank front
tank front edge
tank front edge location
tank location
tank rear edge
tank restricted area
tank safety border
tank sides
tank width
thrust collar
water
water bath
spring support plate
spring support plate quantity
spring support plate size
spring support plate thickness spring type
stop
stop elevation
stop location
stop manufacture
stop purpose
surface of table
table
table area
table back edge location
table depth
table edge
table edge area
table edge restriction
table ends

table front edge
table height
table length
table location
table molding
table rear edge
table sides
table surface
tank
tank area
tank border
tank center line
tank depth
tank edge
tank edge restrictionwater bath level
water bath size
water depth limit
water depth limitation
water depth restriction
water level
water purity
water surface
water surface cleaning
water tank depth
water tank width

Appendix II.B. Feature Attributes

Form Features

Geometry:
area
border
center line
cross section
depth
diameter
edge
elevation
extension
height
hole
layer
length
level
restriction
shape
size
surface
thickness
width

Spatial:
location
orientation
position

Production:
installation
manufacture
mounting
quantity
threads
tolerances
type

Material:
material

Misc:
appearance
comfort
design
effort
general
longevity
purity
safety
toxicity

Function Features

Purpose:
moment
purpose
racking
stability
stresses
torque
twisting

Behavior:
bonding
cleaning
coating
fatigue
loading
movement
operation
power
rotation
time
travel

Appendix III. Implementation Code

III.A. Knowldege Base Printout

```
;;; -*- Mode: Lisp; Syntax: Common-lisp; Package:SLB-CL;
Base: 10 -*-

;;;; Copyright (c) 1990 Schlumberger Technologies, Inc..
All rights reserved.

;;;; Dump of knowledge-base DH-TEMPLATE
;;;; Made by Lucid Common Lisp 3.0.2
;;;; on Sun-4 edison under Unix release unknown
;;;; at NIL.
;;;; Filed on
/home/edison/mcginnis/dh-template/dh-template.kb.
;;;; Written on 21-Jun-90 21:14:12 PDT by Mcginnis.

(in-package :strobe)

;;; Declare knowledge base DH-TEMPLATE

(DEFKB DH-TEMPLATE
       95
       NIL
       ((PACKAGE . "SLB-CL") (WRITE-DATE . 2855016852)
                             (KB-STRUCT-FNS
                               (ACTIVE-DEPENDENTS
DH-TEMPLATE/ACTIVE-DEPENDENTS
                                                  NIL)
                               (BREAKDOWN
DH-TEMPLATE/BREAKDOWN
                                           NIL)
                               (ORIGINAL-SOURCE
DH-TEMPLATE/ORIGINAL-SOURCE
                                              NIL)
                               (ALL-DEPENDENTS
DH-TEMPLATE/ALL-DEPENDENTS
                                             NIL))
                             (SHOWDATATYPESWHENEDITING .
T)
                             (EDITOR . "Mcginnis")
                             (AUTHOR . MCGINNIS)))

;;; Declare the objects in knowledge base DH-TEMPLATE


;;; Declare object ROOT

(DEFOBJECT ROOT
           CLASS
           NIL
```

```
            SYSTEM
            ((DOCUMENTATION (DATATYPE . TEXT)
                            (DOCUMENTATION
                             . "This slot contains
documentation for the object."))
              (EDITOR-TO-USE (VALUE FASTOBJECTEDITOR
                                    PLAYBACKEDITOR)))
            ("Achen" "19-Jul-89 10:01:33 CDT"))

;;; Declare object DATATYPE

(DEFOBJECT DATATYPE
            CLASS
            ROOT
            SYSTEM
            ((DATUM-ADD (DATATYPE . LISP)
                        (VALUE . SYS/MADDVALUE))
              (DATUM-EDIT (DATATYPE . LISP)
                          (VALUE . SYS/EDITEXPR))
              (DATUM-GET (DATATYPE . LISP)
                         (VALUE . SYS/MGETVALUE))
              (DATUM-PUT (DATATYPE . LISP)
                         (VALUE . SYS/MPUTVALUE))
              (DATUM-PRINT (DATATYPE . LISP)
                           (VALUE . SYS/PRINTEXPR))
              (DATUM-REMOVE (DATATYPE . LISP)
                            (VALUE . SYS/MREMOVEVALUE))))

;;; Declare object BITMAP

(DEFOBJECT BITMAP
            CLASS
            DATATYPE
            SYSTEM
            ((DATUM-EDIT (VALUE . SYS/EDITBM))
              (DATUM-PRINT (VALUE . SYS/PRINTBITMAP))))

;;; Declare object EXPR

(DEFOBJECT EXPR
            CLASS
            DATATYPE
            SYSTEM
            ((DATUM-EDIT (VALUE . SYS/EDITEXPR))
              (DATUM-PRINT (VALUE . SYS/PRINTEXPR))))

;;; Declare object NUMERIC

(DEFOBJECT NUMERIC
            CLASS
            EXPR
            TEMPLATE
```

```
            ((DATUM-GET (VALUE . NUMERIC.DATUM-GET))
              (DATUM-PRINT (VALUE . NUMERIC.DATUM-PRINT)))

            ("Mcginnis" "3-May-90 20:12:05 PDT"))

;;; Declare object TEXT-VALUED

(DEFOBJECT TEXT-VALUED
            CLASS
            EXPR
            TEMPLATE
            ((DATUM-PRINT (VALUE .
TEXT-VALUED.DATUM-PRINT))
              (DATUM-GET (VALUE . TEXT-VALUED.DATUM-GET)))

            ("Mcginnis" "7-May-90 13:01:37 PDT"))

;;; Declare object CONSTRAINT-PHRASE

(DEFOBJECT CONSTRAINT-PHRASE
            CLASS
            EXPR
            TEMPLATE
            ((DATUM-GET (VALUE .
CONSTRAINT-PHRASE.DATUM-GET))
              (DATUM-PRINT (VALUE .
CONSTRAINT-PHRASE.DATUM-PRINT)))
            ("Mcginnis" "15-May-90 22:28:57 PDT"))

;;; Declare object LISP

(DEFOBJECT LISP
            CLASS
            DATATYPE
            SYSTEM
            ((DATUM-EDIT (VALUE . SYS/EDITLISP))
              (DATUM-PRINT (VALUE . SYS/PRINTLISP))
              (DATUM-BOTTOMUPADDITIVEINHERITANCE (DATATYPE
. LISP)
                                                  (VALUE
                                                   .
SYS/LISP/DATUM-BOTTOMUPADDITIVEINHERITANCE))
              (DATUM-BOTTOMUPADDITIVEINHERITORS (DATATYPE .
LISP)
                                                 (VALUE
                                                  .
SYS/LISP/DATUM-BOTTOMUPADDITIVEINHERITORS))
              (DATUM-TOPDOWNADDITIVEINHERITANCE (DATATYPE .
LISP)
                                                 (VALUE
                                                  .
SYS/LISP/DATUM-TOPDOWNADDITIVEINHERITANCE))
```

```
            (DATUM-TOPDOWNADDITIVEINHERITORS (DATATYPE .
LISP)
                                                (VALUE
                                                 .
SYS/LISP/DATUM-TOPDOWNADDITIVEINHERITORS))
            (BOTTOMUPADDITIVEINHERITANCE (DATATYPE .
LISP)
                                                 (VALUE
                                                  .
SYS/DATUM-BOTTOMUPADDITIVEINHERITANCE))
            (BOTTOMUPADDITIVEINHERITORS (DATATYPE . LISP)
                                                (VALUE
                                                 .
SYS/DATUM-BOTTOMUPADDITIVEINHERITORS))
            (TOPDOWNADDITIVEINHERITANCE (DATATYPE . LISP)
                                                (VALUE
                                                 .
SYS/DATUM-TOPDOWNADDITIVEINHERITANCE))
            (TOPDOWNADDITIVEINHERITORS (DATATYPE . LISP)
                                               (VALUE
                                                .
SYS/DATUM-TOPDOWNADDITIVEINHERITORS))))

;;; Declare object P-LISP

(DEFOBJECT P-LISP
           CLASS
           LISP
           TEMPLATE
           ((DATUM-PRINT (VALUE . SYS/PRINTLISP-P-LISP)))
           ("Mcginnis" "2-Aug-89 9:47:29 CDT"))

;;; Declare object OBJECT

(DEFOBJECT OBJECT
           CLASS
           DATATYPE
           SYSTEM
           ((DATUM-EDIT (VALUE . SYS/EDITOBJECT))
             (DATUM-PRINT (VALUE . SYS/PRINTOBJECT))
             (CREATE (DATATYPE . LISP))
             (OBJECT-DELETION-PROCEDURES (DATATYPE .
LISP))
             (OBJECT-CREATION-PROCEDURES (DATATYPE .
LISP)))
           ("Mcginnis" "27-Nov-89 15:16:53 PST"))

;;; Declare object DECISION
```

```
(DEFOBJECT DECISION
           CLASS
           OBJECT
           TEMPLATE
           ((INPUT-CONSTRAINTS (DATATYPE .
CONSTRAINT-SOURCE)
                               (VALUE))
             (RESULTING-CONSTRAINT (DATATYPE .
CONSTRAINT-SOURCE)
                                   (VALUE))
             (PRECEDING-DECISION (DATATYPE . DECISION)
                                 (VALUE))
             (SUCCEEDING-DECISION (DATATYPE . DECISION)
                                  (VALUE))
             (OBJECT-DELETION-PROCEDURES
               (VALUE . DECISION.DELETION-PROCEDURES))
             (RATIONALE (DATATYPE . TEXT)))
           ("Mcginnis" "28-May-90 11:51:14 PDT"))

;;; Declare object CONSTRAINT-SOURCE

(DEFOBJECT CONSTRAINT-SOURCE
           CLASS
           OBJECT
           TEMPLATE
           ((CREATE (VALUE)) (TIME-CODE (DATATYPE . EXPR)

                                        (ROLE .
*NOVALUE*))
                             (ACTUAL-TEXT (DATATYPE .
EXPR))
                             (STATUS (DATATYPE .
TEXT-VALUED)
                                     (IN-CONSTRAINT .
*NOVALUE*))
                             (ORIGINATING-DECISION
(DATATYPE . DECISION)))
           ("Mcginnis" "28-May-90 11:49:33 PDT"))

;;; Declare object GIVEN-CONSTRAINT

(DEFOBJECT GIVEN-CONSTRAINT
           CLASS
           CONSTRAINT-SOURCE
           TEMPLATE
           ((SOURCE (DATATYPE . TEXT)
                    (ROLE . *NOVALUE*)))
           ("Mcginnis" "15-May-90 21:40:05 PDT"))

;;; Declare object INTRODUCED-CONSTRAINT
```

```
(DEFOBJECT INTRODUCED-CONSTRAINT
           CLASS
           CONSTRAINT-SOURCE
           TEMPLATE
           ((SOURCE (DATATYPE . TEXT)
                    (ROLE . *NOVALUE*)))
           ("Mcginnis" "15-May-90 21:40:28 PDT"))

;;; Declare object DERIVED-CONSTRAINT

(DEFOBJECT DERIVED-CONSTRAINT
           CLASS
           CONSTRAINT-SOURCE
           TEMPLATE
           ((PRECEEDING-CONSTRAINT (DATATYPE . P-LISP)
                                   (VALUE
                                    .
DERIVED-CONSTRAINT.PRECEEDING-CONSTRAINT)))
           ("Mcginnis" "28-May-90 11:49:57 PDT"))

;;; Declare object CONSTRAINT-ROLE

(DEFOBJECT CONSTRAINT-ROLE
           CLASS
           OBJECT
           TEMPLATE
           ((OBJECT-DELETION-PROCEDURES
              (VALUE .
CONSTRAINT-ROLE.OBJECT-DELETION-PROCEDURES))
            (DEFAULT-LANGUAGE (DATATYPE .
CONSTRAINT-LANGUAGE)
                              (VALUE . TEXTUAL)))
           ("Mcginnis" "15-May-90 23:53:09 PDT"))

;;; Declare object SPATIAL

(DEFOBJECT SPATIAL
           CLASS
           CONSTRAINT-ROLE
           TEMPLATE
           NIL
           ("Mcginnis" "27-May-90 14:23:10 PDT"))

;;; Declare object ORIENTATION

(DEFOBJECT ORIENTATION
           CLASS
           SPATIAL
           TEMPLATE
           ((DEFAULT-LANGUAGE (VALUE .
SPATIAL-ORIENTATION)))
           ("Mcginnis" "15-May-90 21:54:24 PDT"))
```

```
;;; Declare object RESTRICTION

(DEFOBJECT RESTRICTION
           CLASS
           SPATIAL
           TEMPLATE
           ((DEFAULT-LANGUAGE (VALUE .
SPATIAL-RESTRICTION)))
           ("Mcginnis" "15-May-90 21:54:53 PDT"))

;;; Declare object FORM

(DEFOBJECT FORM
           CLASS
           CONSTRAINT-ROLE
           TEMPLATE
           ((DEFAULT-LANGUAGE (VALUE .
OBJECT-FORM-LANGUAGE))
             (SOLVE (DATATYPE . LISP)))
           ("Mcginnis" "27-May-90 14:26:18 PDT"))

;;; Declare object CREATE-OBJECT

(DEFOBJECT CREATE-OBJECT
           CLASS
           FORM
           TEMPLATE
           ((SOLVE (VALUE . CREATE-OBJECT.SOLVE)))
           ("Mcginnis" "1-May-90 22:23:11 PDT"))

;;; Declare object MODIFY-OBJECT-FORM

(DEFOBJECT MODIFY-OBJECT-FORM
           CLASS
           FORM
           TEMPLATE
           ((SOLVE (VALUE . MODIFY-OBJECT-FORM.SOLVE)))
           ("Mcginnis" "2-May-90 1:17:33 PDT"))

;;; Declare object UNCLASSIFIED

(DEFOBJECT UNCLASSIFIED
           CLASS
           CONSTRAINT-ROLE
           TEMPLATE
           ((DEFAULT-LANGUAGE (VALUE . SIMPLE)))
           ("Mcginnis" "27-May-90 14:26:50 PDT"))

;;; Declare object PARAMETER

(DEFOBJECT PARAMETER
```

```
            CLASS
            CONSTRAINT-ROLE
            TEMPLATE
            ((DEFAULT-LANGUAGE (VALUE . EQUATIONAL)))
            ("Mcginnis" "27-May-90 14:24:13 PDT"))

;;; Declare object FUNCTION

(DEFOBJECT FUNCTION
            CLASS
            CONSTRAINT-ROLE
            TEMPLATE
            ((PARSE (DATATYPE . LISP)
                    (VALUE . FUNCTION.PARSE)))
            ("Mcginnis" "7-May-90 13:36:30 PDT"))

;;; Declare object PURPOSE

(DEFOBJECT PURPOSE
            CLASS
            FUNCTION
            TEMPLATE
            ((DEFAULT-LANGUAGE (VALUE .
FUNCTION-CONSTRAINT-LANGUAGE))
              (PURPOSE-1 (DATATYPE . FUNCTION-MODULE)
                         (ROLE . FUNCTION-PARAMETER))
              (PURPOSE-2 (DATATYPE . FUNCTION-MODULE)
                         (ROLE . FUNCTION-PARAMETER))
              (PURPOSE-3 (DATATYPE . FUNCTION-MODULE)
                         (ROLE . FUNCTION-PARAMETER))
              (PURPOSE-4 (DATATYPE . FUNCTION-MODULE)
                         (ROLE . FUNCTION-PARAMETER))
              (PURPOSE-5 (DATATYPE . FUNCTION-MODULE)
                         (ROLE . FUNCTION-PARAMETER))
              (PURPOSE-6 (DATATYPE . FUNCTION-MODULE)
                         (ROLE . FUNCTION-PARAMETER))
              (PURPOSE-7 (DATATYPE . FUNCTION-MODULE)
                         (ROLE . FUNCTION-PARAMETER))
              (PURPOSE-8 (DATATYPE . FUNCTION-MODULE)
                         (ROLE . FUNCTION-PARAMETER))
              (PURPOSE-9 (DATATYPE . FUNCTION-MODULE)
                         (ROLE . FUNCTION-PARAMETER))
              (PURPOSE-10 (DATATYPE . FUNCTION-MODULE)
                          (ROLE . *NOVALUE*)))
            ("Mcginnis" "6-May-90 21:32:49 PDT"))

;;; Declare object BEHAVIOR

(DEFOBJECT BEHAVIOR
            CLASS
            FUNCTION
            TEMPLATE
```

```
            ((BEHAVIOR-1 (DATATYPE . FUNCTION-MODULE)
                         (ROLE . FUNCTION-PARAMETER))
              (BEHAVIOR-2 (DATATYPE . FUNCTION-MODULE)
                          (ROLE . FUNCTION-PARAMETER))
              (BEHAVIOR-3 (DATATYPE . FUNCTION-MODULE)
                          (ROLE . FUNCTION-PARAMETER))
              (BEHAVIOR-4 (DATATYPE . FUNCTION-MODULE)
                          (ROLE . FUNCTION-PARAMETER))
              (BEHAVIOR-5 (DATATYPE . FUNCTION-MODULE)
                          (ROLE . FUNCTION-PARAMETER))
              (BEHAVIOR-6 (DATATYPE . FUNCTION-MODULE)
                          (ROLE . FUNCTION-PARAMETER))
              (BEHAVIOR-7 (DATATYPE . FUNCTION-MODULE)
                          (ROLE . FUNCTION-PARAMETER))
              (BEHAVIOR-8 (DATATYPE . FUNCTION-MODULE)
                          (ROLE . FUNCTION-PARAMETER))
              (BEHAVIOR-9 (DATATYPE . FUNCTION-MODULE)
                          (ROLE . FUNCTION-PARAMETER))
              (BEHAVIOR-10 (DATATYPE . FUNCTION-MODULE)
                           (ROLE . FUNCTION-PARAMETER))
              (DEFAULT-LANGUAGE (VALUE .
FUNCTION-CONSTRAINT-LANGUAGE)))
            ("Mcginnis" "6-May-90 15:29:37 PDT"))

;;; Declare object GENERAL

(DEFOBJECT GENERAL
            CLASS
            FUNCTION
            TEMPLATE
            ((FUNCTION-MODULE (DATATYPE . FUNCTION-MODULE))

              (DEFAULT-LANGUAGE (VALUE .
FUNCTION-CONSTRAINT-LANGUAGE)))
            ("Mcginnis" "6-May-90 22:45:13 PDT"))

;;; Declare object PRODUCTION

(DEFOBJECT PRODUCTION
            CLASS
            CONSTRAINT-ROLE
            TEMPLATE
            ((DEFAULT-LANGUAGE (VALUE .
PRODUCTION-CONSTRAINT-LANGUAGE))
              (STEP-1 (DATATYPE . TEXT)
                      (ROLE . PRODUCTION-PARAMETER))
              (STEP-2 (DATATYPE . TEXT)
                      (ROLE . PRODUCTION-PARAMETER))
              (STEP-3 (DATATYPE . TEXT)
                      (ROLE . PRODUCTION-PARAMETER))
              (STEP-4 (DATATYPE . TEXT)
                      (ROLE . PRODUCTION-PARAMETER))
```

```
            (STEP-5 (DATATYPE . TEXT)
                    (ROLE . PRODUCTION-PARAMETER))
            (STEP-6 (DATATYPE . TEXT)
                    (ROLE . PRODUCTION-PARAMETER))
            (STEP-7 (DATATYPE . TEXT)
                    (ROLE . PRODUCTION-PARAMETER))
            (STEP-8 (DATATYPE . TEXT)
                    (ROLE . PRODUCTION-PARAMETER))
            (STEP-9 (DATATYPE . TEXT)
                    (ROLE . PRODUCTION-PARAMETER))
            (STEP-10 (DATATYPE . TEXT)
                     (ROLE . PRODUCTION-PARAMETER)))
          ("Mcginnis" "28-May-90 12:39:43 PDT"))

;;; Declare object ASSEMBLY

(DEFOBJECT ASSEMBLY CLASS PRODUCTION TEMPLATE)

;;; Declare object MANUFACTURE

(DEFOBJECT MANUFACTURE CLASS PRODUCTION TEMPLATE)

;;; Declare object STATUS

(DEFOBJECT STATUS
           CLASS
           CONSTRAINT-ROLE
           TEMPLATE
           ((DEFAULT-LANGUAGE (VALUE . STATUS-LANGUAGE))
             (STATUS (DATATYPE . EXPR)))
           ("Mcginnis" "7-May-90 14:10:08 PDT"))

;;; Declare object ACCEPT

(DEFOBJECT ACCEPT
           CLASS
           STATUS
           TEMPLATE
           ((STATUS (VALUE . ACTIVE)))
           ("Mcginnis" "2-May-90 22:52:11 PDT"))

;;; Declare object REJECT

(DEFOBJECT REJECT
           CLASS
           STATUS
           TEMPLATE
           ((STATUS (VALUE . IN-ACTIVE))))

;;; Declare object SUSPEND

(DEFOBJECT SUSPEND
```

```
            CLASS
            STATUS
            TEMPLATE
            ((STATUS (VALUE . SUSPENDED))))

;;; Declare object CONSTRAINT-LANGUAGE

(DEFOBJECT CONSTRAINT-LANGUAGE
            CLASS
            OBJECT
            TEMPLATE
            ((UPDATE (DATATYPE . LISP)
                     (VALUE . CONSTRAINT-LANGUAGE.UPDATE))

              (SOLVE (DATATYPE . LISP))
              (SUCCESSOR)
              (PARSE (DATATYPE . LISP) (VALUE))
              (PHRASE (DATATYPE . CONSTRAINT-PHRASE)
                      (ROLE . *NOVALUE*))
              (LINK-CONSTRAINT (DATATYPE . LISP)
                               (VALUE .
CONSTRAINT-LANGUAGE.LINK-CONSTRAINT))
              (SATISFIED (DATATYPE . LISP)))
            ("Mcginnis" "16-May-90 16:03:06 PDT"))

;;; Declare object EQUATIONAL

(DEFOBJECT EQUATIONAL
            CLASS
            CONSTRAINT-LANGUAGE
            TEMPLATE
            ((EQUATION (DATATYPE . EXPR))
              (SATISFIED (VALUE . EQUATIONAL.SATISFIED)))
            ("Mcginnis" "16-May-90 16:07:06 PDT"))

;;; Declare object EQUALITY-CONSTRAINT

(DEFOBJECT EQUALITY-CONSTRAINT
            CLASS
            EQUATIONAL
            (TEMPLATE CONST-TYPE)
            ((SOLVE (VALUE . EQUALITY-CONSTRAINT.SOLVE))
              (UPDATE (VALUE . EQUALITY-CONSTRAINT.UPDATE))

              (PARSE (VALUE . EQUALITY-CONSTRAINT.PARSE)))

            ("Mcginnis" "15-May-90 21:57:07 PDT"))

;;; Declare object A=2*B-C

(DEFOBJECT A=2*B-C
            CLASS
```

```
            EQUALITY-CONSTRAINT
            TEMPLATE
            ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
                (PATH . *NOVALUE*)
                (ROLE . OUT))
              (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                 (SOURCE . *NOVALUE*)
                 (PATH . *NOVALUE*)
                 (ROLE . IN))
              (C (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                 (SOURCE . *NOVALUE*)
                 (PATH . *NOVALUE*)
                 (ROLE . IN))
              (EQUATION (VALUE = A (* 2 (- B C)))))
            ("Mcginnis" "15-May-90 22:05:52 PDT"))

;;; Declare object A=B

(DEFOBJECT A=B
            CLASS
            EQUALITY-CONSTRAINT
            TEMPLATE
            ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
                (PATH . *NOVALUE*)
                (ROLE . OUT))
              (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                 (SOURCE . *NOVALUE*)
                 (PATH . *NOVALUE*)
                 (ROLE . IN))
              (EQUATION (VALUE = A B)))
            ("Mcginnis" "15-May-90 23:08:43 PDT"))

;;; Declare object A=B/2+2

(DEFOBJECT A=B/2+2
            CLASS
            EQUALITY-CONSTRAINT
            TEMPLATE
            ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
                (PATH . *NOVALUE*)
                (ROLE . OUT))
              (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                 (SOURCE . *NOVALUE*)
                 (PATH . *NOVALUE*)
                 (ROLE . IN))
              (EQUATION (VALUE = A (+ 2 (/ B 2)))))
            ("Mcginnis" "15-May-90 22:10:19 PDT"))
```

```
;;; Declare object A=B+C

(DEFOBJECT A=B+C
           CLASS
           EQUALITY-CONSTRAINT
           TEMPLATE
           ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
               (PATH . *NOVALUE*)
               (ROLE . OUT))
            (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

               (SOURCE . *NOVALUE*)
               (PATH . *NOVALUE*)
               (ROLE . IN))
            (C (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

               (SOURCE . *NOVALUE*)
               (PATH . *NOVALUE*)
               (ROLE . IN))
            (EQUATION (VALUE = A (+ B C))))
           ("Mcginnis" "15-May-90 22:10:54 PDT"))

;;; Declare object A=2*B

(DEFOBJECT A=2*B
           CLASS
           EQUALITY-CONSTRAINT
           TEMPLATE
           ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
               (PATH . *NOVALUE*)
               (ROLE . OUT))
            (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

               (SOURCE . *NOVALUE*)
               (PATH . *NOVALUE*)
               (ROLE . IN))
            (EQUATION (VALUE = A (* 2 B))))
           ("Mcginnis" "15-May-90 22:11:15 PDT"))

;;; Declare object A=-2*B

(DEFOBJECT A=-2*B
           CLASS
           EQUALITY-CONSTRAINT
           TEMPLATE
           ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
               (PATH . *NOVALUE*)
               (ROLE . OUT))
            (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

               (SOURCE . *NOVALUE*)
               (PATH . *NOVALUE*)
```

```
                (ROLE . IN))
              (EQUATION (VALUE = A (* 2 B))))
            ("Mcginnis" "15-May-90 22:11:39 PDT"))

;;; Declare object A=B*C

(DEFOBJECT A=B*C
           CLASS
           EQUALITY-CONSTRAINT
           TEMPLATE
           ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
               (PATH . *NOVALUE*)
               (ROLE . OUT))
             (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                (SOURCE . *NOVALUE*)
                (PATH . *NOVALUE*)
                (ROLE . IN))
             (C (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                (SOURCE . *NOVALUE*)
                (PATH . *NOVALUE*)
                (ROLE . IN))
              (EQUATION (VALUE = A (* B C))))
            ("Mcginnis" "15-May-90 22:11:57 PDT"))

;;; Declare object A=B*C*D

(DEFOBJECT A=B*C*D
           CLASS
           EQUALITY-CONSTRAINT
           TEMPLATE
           ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
               (PATH . *NOVALUE*)
               (ROLE . OUT))
             (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                (SOURCE . *NOVALUE*)
                (PATH . *NOVALUE*)
                (ROLE . IN))
             (C (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                (SOURCE . *NOVALUE*)
                (PATH . *NOVALUE*)
                (ROLE . IN))
             (D (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                (SOURCE . *NOVALUE*)
                (PATH . *NOVALUE*)
                (ROLE . IN))
              (EQUATION (VALUE = A (* B C D))))
            ("Mcginnis" "15-May-90 22:12:20 PDT"))
```

```
;;; Declare object A=VALUE

(DEFOBJECT A=VALUE
           CLASS
           EQUALITY-CONSTRAINT
           TEMPLATE
           ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
               (PATH . *NOVALUE*)
               (ROLE . OUT))
             (VAL (DATATYPE . EXPR)
                  (SOURCE . *NOVALUE*)
                  (ROLE . IN))
             (EQUATION (VALUE = A VAL)))
           ("Mcginnis" "15-May-90 22:47:23 PDT"))

;;; Declare object INEQUALITY-CONSTRAINT

(DEFOBJECT INEQUALITY-CONSTRAINT
           CLASS
           EQUATIONAL
           (TEMPLATE CONST-TYPE)
           ((SOLVE (VALUE . INEQUALITY-CONSTRAINT.SOLVE))

             (PARSE (VALUE . EQUALITY-CONSTRAINT.PARSE)))

           ("Mcginnis" "15-May-90 22:44:52 PDT"))

;;; Declare object A<B

(DEFOBJECT A<B
           CLASS
           INEQUALITY-CONSTRAINT
           TEMPLATE
           ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
               (ROLE . OUT)
               (PATH)
               (SOURCE))
             (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                (ROLE . IN)
                (PATH)
                (SOURCE))
             (EQUATION (VALUE < A B))))

;;; Declare object A>B

(DEFOBJECT A>B
           CLASS
           INEQUALITY-CONSTRAINT
           TEMPLATE
           ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
```

```
                (ROLE . OUT)
                (PATH)
                (SOURCE))
              (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                 (ROLE . IN)
                 (PATH)
                 (SOURCE))
              (EQUATION (VALUE > A B))))

;;; Declare object A>VALUE

(DEFOBJECT A>VALUE
           CLASS
           INEQUALITY-CONSTRAINT
           TEMPLATE
           ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
               (SOURCE)
               (PATH)
               (ROLE . OUT))
            (EQUATION (VALUE > A VALUE))
            (VALUE (DATATYPE . EXPR)
                   (SOURCE . *NOVALUE*)
                   (ROLE . IN)))
           ("Mcginnis" "9-May-90 15:11:00 PDT"))

;;; Declare object QUALITY-CONSTRAINT

(DEFOBJECT QUALITY-CONSTRAINT
           CLASS
           EQUATIONAL
           (TEMPLATE CONST-TYPE)
           ((DEPENDENT-FEATURE (DATATYPE .
NUMERIC-CONSTRAINT-PARAMETER)
                               (ROLE . OUT)
                               (PATH))
            (INDEPENDENT-FEATURE (DATATYPE .
NUMERIC-CONSTRAINT-PARAMETER)
                                 (ROLE . IN)
                                 (PATH))
            (QUALITY-EXPR (DATATYPE . TEXT)
                          (ROLE . IN)
                          (PATH))
            (PARSE (VALUE . QUALITY-CONSTRAINT.PARSE)))
           ("Mcginnis" "20-Jun-90 2:47:08 PDT"))

;;; Declare object CONDITIONAL-CONSTRAINT

(DEFOBJECT CONDITIONAL-CONSTRAINT
           CLASS
           EQUATIONAL
           TEMPLATE
```

```
            NIL
            ("Mcginnis" "16-May-90 0:09:53 PDT"))

;;; Declare object IF-A>B-THEN-A=B

(DEFOBJECT IF-A>B-THEN-A=B
            CLASS
            CONDITIONAL-CONSTRAINT
            TEMPLATE
            ((A (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)
                (ROLE . OUT)
                (PATH)
                (SOURCE))
              (B (DATATYPE . NUMERIC-CONSTRAINT-PARAMETER)

                 (ROLE . IN)
                 (PATH)
                 (SOURCE))
              (EQUATION (VALUE IF (> A B) (= A B)))))

;;; Declare object OTHER

(DEFOBJECT OTHER CLASS EQUATIONAL TEMPLATE)

;;; Declare object GRAPHICAL

(DEFOBJECT GRAPHICAL
            CLASS
            CONSTRAINT-LANGUAGE
            TEMPLATE
            ((ASSERT (DATATYPE . LISP)))
            ("Mcginnis" "30-Apr-90 15:30:49 PDT"))

;;; Declare object SPATIAL-ORIENTATION

(DEFOBJECT SPATIAL-ORIENTATION
            CLASS
            GRAPHICAL
            TEMPLATE
            ((RELATION (DATATYPE . EXPR))
              (PARSE (VALUE . SPACIAL-ORIENTATION.PARSE)))

            ("Mcginnis" "21-Jun-90 21:01:37 PDT"))

;;; Declare object SURFACE-CONSTRAINT

(DEFOBJECT SURFACE-CONSTRAINT
            CLASS
            SPATIAL-ORIENTATION
            TEMPLATE
            ((DEPENDENT-SURFACE (DATATYPE .
GEOMETRIC-CONSTRAINT-PARAMETER)
```

```
                                    (ROLE . OUT)
                                    (PATH . *NOVALUE*))
              (INDEPENDENT-SURFACE (DATATYPE .
GEOMETRIC-CONSTRAINT-PARAMETER)
                                    (ROLE . IN)
                                    (PATH . *NOVALUE*)))
            ("Mcginnis" "15-May-90 21:44:25 PDT"))

;;; Declare object PARALLEL-SURFACE-CONSTRAINT

(DEFOBJECT PARALLEL-SURFACE-CONSTRAINT
            CLASS
            SURFACE-CONSTRAINT
            TEMPLATE
            ((ASSERT (VALUE .
PARALLEL-SURFACE-CONSTRAINT.ASSERT))
              (LINK-CONSTRAINT
                (VALUE .
PARALLEL-SURFACE-CONSTRAINT.LINK-CONSTRAINT))
              (RELATION (VALUE . "parallel to")))
            ("Mcginnis" "15-May-90 22:58:50 PDT"))

;;; Declare object PERPENDICULAR-SURFACE-CONSTRAINT

(DEFOBJECT PERPENDICULAR-SURFACE-CONSTRAINT
            CLASS
            SURFACE-CONSTRAINT
            TEMPLATE
            ((ASSERT (VALUE .
PERPENDICULAR-SURFACE-CONSTRAINT.ASSERT))
              (RELATION (VALUE . "perpendicular to"))
              (LINK-CONSTRAINT
                (VALUE .
PERPENDICULAR-SURFACE-CONSTRAINT.LINK-CONSTRAINT)))
            ("Mcginnis" "15-May-90 23:54:35 PDT"))

;;; Declare object COPLANAR-SURFACE-CONSTRAINT

(DEFOBJECT COPLANAR-SURFACE-CONSTRAINT
            CLASS
            SURFACE-CONSTRAINT
            TEMPLATE
            ((ASSERT (VALUE .
COPLANAR-SURFACE-CONSTRAINT.ASSERT))
              (LINK-CONSTRAINT
                (VALUE .
COPLANAR-SURFACE-CONSTRAINT.LINK-CONSTRAINT))
              (RELATION (VALUE . "coplanar to")))
            ("Mcginnis" "15-May-90 23:00:00 PDT"))

;;; Declare object DISTANCE-BETWEEN-SURFACE-CONSTRAINT
```

```
(DEFOBJECT DISTANCE-BETWEEN-SURFACE-CONSTRAINT
           CLASS
           SURFACE-CONSTRAINT
           TEMPLATE
           ((ASSERT (VALUE .
DISTANCE-BETWEEN-SURFACE-CONSTRAINT.ASSERT))
            (DISTANCE (DATATYPE .
NUMERIC-CONSTRAINT-PARAMETER)
                      (PATH . *NOVALUE*))
            (INITIAL-DIST-VALUE (DATATYPE . EXPR)
                                (ROLE . STARTER-IN))
            (LINK-CONSTRAINT
              (VALUE .
COPLANAR-SURFACE-CONSTRAINT.LINK-CONSTRAINT))
            (CREATE (VALUE .
DISTANCE-BETWEEN-SURFACE-CONSTRAINT.CREATE))
            (RELATION (VALUE . "set distance from")))
           ("Mcginnis" "15-May-90 23:00:49 PDT"))

;;; Declare object OPPOSING-COPLANAR-SURFACE-CONSTRAINT

(DEFOBJECT OPPOSING-COPLANAR-SURFACE-CONSTRAINT
           CLASS
           SURFACE-CONSTRAINT
           TEMPLATE
           ((ASSERT (*INHERIT* .
COPLANAR-SURFACE-CONSTRAINT)
                    (DATATYPE . LISP)
                    (VALUE .
OPPOSING-COPLANAR-SURFACE-CONSTRAINT.ASSERT))
            (LINK-CONSTRAINT (*INHERIT* .
COPLANAR-SURFACE-CONSTRAINT)
                             (DATATYPE . LISP)
                             (VALUE
                              .
COPLANAR-SURFACE-CONSTRAINT.LINK-CONSTRAINT))
            (RELATION (*INHERIT* .
COPLANAR-SURFACE-CONSTRAINT)
                      (DATATYPE . EXPR)
                      (VALUE . "coplanar to")))
           ("Mcginnis" "20-Jun-90 0:17:27 PDT"))

;;; Declare object AXIAL-CONSTRAINT

(DEFOBJECT AXIAL-CONSTRAINT
           CLASS
           SPATIAL-ORIENTATION
           TEMPLATE
           NIL
           ("Mcginnis" "15-May-90 21:44:51 PDT"))

;;; Declare object EDGE-CONSTRAINT
```

```
(DEFOBJECT EDGE-CONSTRAINT
           CLASS
           SPATIAL-ORIENTATION
           TEMPLATE
           NIL
           ("Mcginnis" "15-May-90 21:45:18 PDT"))

;;; Declare object SPATIAL-RESTRICTION

(DEFOBJECT SPATIAL-RESTRICTION
           CLASS
           GRAPHICAL
           TEMPLATE
           ((RESTRICTED-OBJECT (DATATYPE . DESIGN-OBJECT)

                               (ROLE . OUT)
                               (PATH . *NOVALUE*)))
           ("Mcginnis" "21-Jun-90 21:02:02 PDT"))

;;; Declare object VOLUME-RESTRICTION

(DEFOBJECT VOLUME-RESTRICTION
           CLASS
           SPATIAL-RESTRICTION
           TEMPLATE
           ((RESTRICTED-AREA (DATATYPE . TEXT)
                             (ROLE . *NOVALUE*)))
           ("Mcginnis" "15-May-90 21:53:36 PDT"))

;;; Declare object SURFACE-RESTRICTION

(DEFOBJECT SURFACE-RESTRICTION
           CLASS
           SPATIAL-RESTRICTION
           TEMPLATE
           ((REFERENCE (DATATYPE . TEXT)
                       (ROLE . *NOVALUE*))
             (DISTANCE (DATATYPE . EXPR)
                       (ROLE . *NOVALUE*))
             (DIRECTION (DATATYPE . TEXT)
                        (ROLE . *NOVALUE*)))
           ("Mcginnis" "15-May-90 21:53:08 PDT"))

;;; Declare object TEXTUAL

(DEFOBJECT TEXTUAL
           CLASS
           CONSTRAINT-LANGUAGE
           TEMPLATE
           ((PARSE) (SOLVE (VALUE . TEXTUAL.SOLVE))
                    (OBJECT (DATATYPE . DESIGN-OBJECT)
```

```
                              (PATH . *NOVALUE*)
                              (ROLE . OUT)))
            ("Mcginnis" "6-May-90 15:15:17 PDT"))

;;; Declare object STRUCTURED

(DEFOBJECT STRUCTURED
            CLASS
            TEXTUAL
            (TEMPLATE CONST-TYPE)
            ((ACTION (DATATYPE . EXPR)
                     (ROLE . *NOVALUE*))
              (RECEIVER (DATATYPE . DESIGN-OBJECT)
                        (ROLE . *NOVALUE*))
              (LOCATION (DATATYPE . EXPR)
                        (ROLE . *NOVALUE*))
              (PRE-CONDITION)
              (POST-CONDITION)
              (ACTION-QUALIFIER (DATATYPE . EXPR)
                                (ROLE . *NOVALUE*))
              (PARSE (VALUE . STRUCTURED.PARSE))
              (SEQUENCE (ROLE . SERIES))
              (DECOMPOSITION (ROLE . SERIES))
              (ALTERNATIVES (ROLE . SERIES)))
            ("Mcginnis" "7-May-90 13:30:51 PDT"))

;;; Declare object SIMPLE

(DEFOBJECT SIMPLE
            CLASS
            TEXTUAL
            TEMPLATE
            ((PARSE (VALUE . SIMPLE.PARSE))
              (DEPENDENT-FEATURE (DATATYPE .
TEXTUAL-CONSTRAINT-PARAMETER)
                                 (PATH)
                                 (ROLE . OUT))
              (INDEPENDENT-FEATURE (DATATYPE .
TEXTUAL-CONSTRAINT-PARAMETER)
                                   (PATH)
                                   (ROLE . IN))
              (INDEPENDENT-FEATURE2 (DATATYPE .
TEXTUAL-CONSTRAINT-PARAMETER)
                                    (PATH)
                                    (ROLE . IN))
              (INSTANTIATION/RELATION (DATATYPE . TEXT)
                                      (PATH)
                                      (ROLE . IN)))
            ("Mcginnis" "15-May-90 21:24:08 PDT"))

;;; Declare object SPECIAL
```

```
(DEFOBJECT SPECIAL
           CLASS
           TEXTUAL
           TEMPLATE
           NIL
           ("Mcginnis" "6-May-90 15:06:02 PDT"))

;;; Declare object OBJECT-FORM-LANGUAGE

(DEFOBJECT OBJECT-FORM-LANGUAGE
           CLASS
           SPECIAL
           TEMPLATE
           ((NAME (DATATYPE . EXPR)
                  (ROLE . *NOVALUE*))
             (SHAPE (DATATYPE . DESIGN-PRIMITIVES)
                    (ROLE . *NOVALUE*))
             (PARENT (DATATYPE . DESIGN-OBJECT)
                     (ROLE . *NOVALUE*))
             (COMPONENTS (DATATYPE . DESIGN-OBJECT)
                         (ROLE . *NOVALUE*))
             (PARSE (VALUE . OBJECT-FORM-LANGUAGE.PARSE)))

           ("Mcginnis" "7-May-90 21:12:43 PDT"))

;;; Declare object STATUS-LANGUAGE

(DEFOBJECT STATUS-LANGUAGE
           CLASS
           SPECIAL
           TEMPLATE
           ((CONSTRAINT-AFFECTED (DATATYPE .
CONSTRAINT-SOURCE)
                                 (ROLE . OUT)
                                 (PATH . *NOVALUE*))
             (FEATURE-AFFECTED (DATATYPE .
CONSTRAINT-PARAMETER)
                               (ROLE . *NOVALUE*)
                               (PATH . *NOVALUE*))
             (SOLVE (VALUE . STATUS-LANGUAGE.SOLVE)))
           ("Mcginnis" "2-May-90 23:35:32 PDT"))

;;; Declare object FUNCTION-CONSTRAINT-LANGUAGE

(DEFOBJECT FUNCTION-CONSTRAINT-LANGUAGE
           CLASS
           SPECIAL
           TEMPLATE
           NIL
           ("Mcginnis" "28-May-90 11:55:54 PDT"))

;;; Declare object PRODUCTION-CONSTRAINT-LANGUAGE
```

```
(DEFOBJECT PRODUCTION-CONSTRAINT-LANGUAGE
           CLASS
           SPECIAL
           TEMPLATE
           ((PARSE (VALUE .
PRODUCTION-CONSTRAINT-LANGUAGE.PARSE)))
           ("Mcginnis" "28-May-90 14:39:20 PDT"))

;;; Declare object CONSTRAINT-PARAMETER

(DEFOBJECT CONSTRAINT-PARAMETER
           CLASS
           OBJECT
           TEMPLATE
           NIL
           ("Mcginnis" "27-Apr-90 3:52:59 PDT"))

;;; Declare object GEOMETRIC-CONSTRAINT-PARAMETER

(DEFOBJECT GEOMETRIC-CONSTRAINT-PARAMETER
           CLASS
           CONSTRAINT-PARAMETER
           TEMPLATE
           ((DATUM-PRINT (VALUE .
GEOMETRIC-CONSTRAINT-PARAMETER.PRINT))
             (DATUM-GET (VALUE .
GEOMETRIC-CONSTRAINT-PARAMETER.GET)))
           ("Mcginnis" "18-Aug-89 9:53:43 CDT"))

;;; Declare object NUMERIC-CONSTRAINT-PARAMETER

(DEFOBJECT NUMERIC-CONSTRAINT-PARAMETER
           CLASS
           CONSTRAINT-PARAMETER
           TEMPLATE
           NIL
           ("Mcginnis" "27-Apr-90 4:01:00 PDT"))

;;; Declare object TEXTUAL-CONSTRAINT-PARAMETER

(DEFOBJECT TEXTUAL-CONSTRAINT-PARAMETER
           CLASS
           CONSTRAINT-PARAMETER
           TEMPLATE)

;;; Declare object DESIGN-OBJECT

(DEFOBJECT DESIGN-OBJECT
           CLASS
           OBJECT
           NIL
```

```
            ((COMPOSED-OF (DATATYPE . P-LISP)
                          (VALUE .
DESIGN-OBJECT.COMPOSED-OF))
             (FORM (DATATYPE . TEXT-VALUED)
                   (IN-CONSTRAINT . *NOVALUE*)
                   (OUT-CONSTRAINT . *NOVALUE*)
                   (ROLE . DUMMY-PARAMETER))
             (CREATE-PARAMETER (DATATYPE . LISP)
                               (VALUE .
DESIGN-OBJECT.CREATE-PARAMETER))
             (PURPOSE (DATATYPE . TEXT-VALUED)
                      (OUT-CONSTRAINT . *NOVALUE*)
                      (IN-CONSTRAINT . *NOVALUE*))
             (COLOR (DATATYPE . TEXT-VALUED)
                    (VALUE)
                    (IN-CONSTRAINT . *NOVALUE*)
                    (OUT-CONSTRAINT . *NOVALUE*)
                    (ROLE . PARAMETER))
             (ASSEMBLY (DATATYPE . TEXT-VALUED)
                       (VALUE)
                       (IN-CONSTRAINT . *NOVALUE*)
                       (OUT-CONSTRAINT . *NOVALUE*)
                       (ROLE . PARAMETER))
             (MANUFACTURE (DATATYPE . TEXT-VALUED)
                          (VALUE)
                          (IN-CONSTRAINT . *NOVALUE*)
                          (OUT-CONSTRAINT . *NOVALUE*)
                          (ROLE . PARAMETER))
             (BEHAVIOR (DATATYPE . TEXT-VALUED)
                       (VALUE)
                       (IN-CONSTRAINT . *NOVALUE*)
                       (OUT-CONSTRAINT . *NOVALUE*)
                       (ROLE . PARAMETER))
             (PARENT (DATATYPE . DESIGN-OBJECT)))
            ("Mcginnis" "27-May-90 14:24:39 PDT"))

;;; Declare object DESIGN-PRIMITIVES

(DEFOBJECT DESIGN-PRIMITIVES
           CLASS
           OBJECT
           TEMPLATE
           ((CREATE (VALUE . DESIGN-PRIMITIVES.CREATE))
             (OBJECT-DELETION-PROCEDURES (VALUE))
             (SHAPE (DATATYPE . LISP)
                    (ROLE . DUMMY-PARAMETER))
             (TRANSFORM-VECTOR (DATATYPE . P-LISP)
                               (VALUE .
DESIGN-PRIMITIVES.TRANSFORM-VECTOR))
             (X-TRANSLATION (DATATYPE . EXPR)
                            (VALUE . 0)
                            (ROLE . GEOMETRY)
```

```
                                (IN-CONSTRAINT . *NOVALUE*)
                                (OUT-CONSTRAINT . *NOVALUE*))

                    (Y-TRANSLATION (DATATYPE . EXPR)
                                   (VALUE . 0)
                                   (ROLE . GEOMETRY)
                                   (IN-CONSTRAINT)
                                   (OUT-CONSTRAINT))
                    (Z-TRANSLATION (DATATYPE . EXPR)
                                   (VALUE . 0)
                                   (ROLE . GEOMETRY)
                                   (IN-CONSTRAINT)
                                   (OUT-CONSTRAINT))
                    (X-ROTATION (DATATYPE . EXPR)
                                (VALUE . 0)
                                (ROLE . GEOMETRY)
                                (IN-CONSTRAINT)
                                (OUT-CONSTRAINT))
                    (Y-ROTATION (DATATYPE . EXPR)
                                (VALUE . 0)
                                (ROLE . GEOMETRY)
                                (IN-CONSTRAINT)
                                (OUT-CONSTRAINT))
                    (Z-ROTATION (DATATYPE . EXPR)
                                (VALUE . 0)
                                (ROLE . GEOMETRY)
                                (IN-CONSTRAINT)
                                (OUT-CONSTRAINT))
                    (FACES (DATATYPE . LISP))
                    (MAKE-NODE (DATATYPE . LISP)
                               (VALUE .
DESIGN-PRIMITIVE.MAKE-NODE))
                    (STATUS (VALUE))
                    (MAKE-CSG-NODE (DATATYPE . LISP)
                                   (VALUE .
DESIGN-PRIMITIVE.MAKE-CSG-NODE))
                    (ORIENTATION (DATATYPE . P-LISP)
                                 (VALUE .
DESIGN-PRIMITIVES.ORIENTATION)))
                ("Mcginnis" "15-May-90 23:23:23 PDT"))

;;; Declare object COMPOSITE

(DEFOBJECT COMPOSITE
           CLASS
           DESIGN-PRIMITIVES
           TEMPLATE
           ((MAKE-NODE (VALUE . COMPOSITE.MAKE-NODE))
(SHAPE (VALUE + NIL))

(LENGTH
```

```
(DATATYPE . NUMERIC)
(VALUE . 1)
(IN-CONSTRAINT
*NOVALUE*)
(OUT-CONSTRAINT
*NOVALUE*)
(ROLE . PARAMETER))
(HEIGHT
(DATATYPE . NUMERIC)
(VALUE . 1)
(IN-CONSTRAINT
*NOVALUE*)
(OUT-CONSTRAINT
*NOVALUE*)
(ROLE . PARAMETER))
(DEPTH
(DATATYPE . NUMERIC)
(VALUE . 1)
(IN-CONSTRAINT
*NOVALUE*)
(OUT-CONSTRAINT
*NOVALUE*)
(ROLE . PARAMETER))
(FACES
(VALUE
(TOP
```

```
 (LIST 270
       90
       0
       (/ HEIGHT 2)))
(BOTTOM
 (LIST 90
       270
       180
       (/ HEIGHT 2)))
(FRONT
 (LIST 0
       270
       90
       (/ DEPTH 2)))
(BACK
 (LIST 180
       90
       270
       (/ DEPTH 2)))
(RIGHT
 (LIST 90
       0
       270
       (/ LENGTH 2)))
(LEFT
 (LIST 270
```

```
        180

        90

        (/ LENGTH 2))))))
            ("Mcginnis" "19-Jun-90 14:18:37 PDT"))

;;; Declare object SLAB

(DEFOBJECT SLAB
            CLASS
            DESIGN-PRIMITIVES
            TEMPLATE
            (((LENGTH Y-DIM)  (DATATYPE . NUMERIC)
                              (VALUE . 1)
                              (OUT-CONSTRAINT . *NOVALUE*)
                              (ROLE . PARAMETER)
                              (IN-CONSTRAINT . *NOVALUE*))
              ((HEIGHT Z-DIM)  (DATATYPE . NUMERIC)
                               (VALUE . 1)
                               (OUT-CONSTRAINT . *NOVALUE*)

                               (ROLE . PARAMETER)
                               (IN-CONSTRAINT . *NOVALUE*))

              ((DEPTH X-DIM)  (DATATYPE . NUMERIC)
                              (VALUE . 1)
                              (OUT-CONSTRAINT . *NOVALUE*)
                              (IN-CONSTRAINT . *NOVALUE*)
                              (ROLE . PARAMETER))
              (FACES
                (VALUE (TOP (LIST 270 90 0 (/ HEIGHT 2)))
                       (BOTTOM (LIST 90 270 180 (/ HEIGHT
2)))
                       (FRONT (LIST 0 270 90 (/ DEPTH 2)))

                       (BACK (LIST 180 90 270 (/ DEPTH 2)))

                       (RIGHT (LIST 90 0 270 (/ LENGTH 2)))

                       (LEFT (LIST 270 180 90 (/ LENGTH
2)))))
              (SHAPE (VALUE + CUBE)))
            ("Mcginnis" "19-Jun-90 16:39:08 PDT"))

;;; Declare object SLOT

(DEFOBJECT SLOT
            CLASS
            SLAB
            TEMPLATE
            ((SHAPE (*INHERIT* . DESIGN-PRIMITIVES)
```

```
                    (DATATYPE . LISP)
                    (VALUE - CUBE)))
            ("Mcginnis" "14-Nov-89 23:07:42 PST"))

;;; Declare object CYLINDER

(DEFOBJECT CYLINDER
           CLASS
           DESIGN-PRIMITIVES
           TEMPLATE
           ((SHAPE (VALUE + CYLINDER))
             ((RADIUS LENGTH DEPTH) (DATATYPE . NUMERIC)
                                    (VALUE . 1)
                                    (OUT-CONSTRAINT .
*NOVALUE*)
                                    (IN-CONSTRAINT .
*NOVALUE*)
                                    (ROLE . PARAMETER))
             (HEIGHT (DATATYPE . NUMERIC)
                     (VALUE . 1)
                     (OUT-CONSTRAINT . *NOVALUE*)
                     (IN-CONSTRAINT . *NOVALUE*)
                     (ROLE . PARAMETER))
             (NUMBER-OF-APP-FACES (DATATYPE . EXPR)
                                  (VALUE . 10)
                                  (ROLE . PARAMETER))
             (FACES
               (VALUE (TOP (LIST 270 90 0 (/ HEIGHT 2)))
                      (BOTTOM (LIST 90 270 180 (/ HEIGHT
2))))))
           ("Mcginnis" "19-Jun-90 16:47:54 PDT"))

;;; Declare object HOLE

(DEFOBJECT HOLE
           CLASS
           CYLINDER
           TEMPLATE
           ((SHAPE (VALUE - CYLINDER)))
           ("Mcginnis" "14-Nov-89 20:29:42 PST"))

;;; Declare object POS-RIGHT-WEDGE

(DEFOBJECT POS-RIGHT-WEDGE
           CLASS
           DESIGN-PRIMITIVES
           TEMPLATE
           (((LENGTH Y-DIM) (DATATYPE . NUMERIC)
                            (VALUE . 1)
                            (ROLE . PARAMETER)
                            (IN-CONSTRAINT . *NOVALUE*)
                            (OUT-CONSTRAINT . *NOVALUE*))
```

```
                ((HEIGHT Z-DIM)  (DATATYPE . NUMERIC)
                                 (VALUE . 1)
                                 (OUT-CONSTRAINT)
                                 (IN-CONSTRAINT)
                                 (ROLE . PARAMETER))
                ((DEPTH X-DIM)  (DATATYPE . NUMERIC)
                                (VALUE . 1)
                                (OUT-CONSTRAINT)
                                (IN-CONSTRAINT)
                                (ROLE . PARAMETER))
                (SHAPE (VALUE + RT-ANG))
                (FACES
                  (VALUE (TOP (LIST 270 90 0 HEIGHT))
                         (BOTTOM (LIST 90 270 180 0))
                         (BACK (LIST 180 90 270 0))
                         (LEFT (LIST 270 180 90 0))
                         (SLANT (POS-RIGHT-WEDGE/SLANT-FACE
DEPTH LENGTH)))))
              ("Mcginnis" "19-Jun-90 16:37:14 PDT"))

;;; Declare object NEG-RIGHT-WEDGE

(DEFOBJECT NEG-RIGHT-WEDGE
              CLASS
              POS-RIGHT-WEDGE
              TEMPLATE
              ((SHAPE (VALUE - RT-ANG)))
              ("Mcginnis" "19-Jun-90 15:00:47 PDT"))

;;; Declare object L-SHAPE

(DEFOBJECT L-SHAPE
              CLASS
              DESIGN-PRIMITIVES
              TEMPLATE
              (((MAJOR-HEIGHT Z-DIM)  (DATATYPE . NUMERIC)
                                      (VALUE . 2)
                                      (OUT-CONSTRAINT)
                                      (IN-CONSTRAINT)
                                      (ROLE . PARAMETER))
                ((MAJOR-LENGTH Y-DIM)  (DATATYPE . NUMERIC)
                                       (VALUE . 2)
                                       (OUT-CONSTRAINT)
                                       (IN-CONSTRAINT)
                                       (ROLE . PARAMETER))
                ((DEPTH X-DIM)  (DATATYPE . NUMERIC)
                                (VALUE . 1)
                                (OUT-CONSTRAINT)
                                (IN-CONSTRAINT)
                                (ROLE . PARAMETER))
                (MINOR-HEIGHT (DATATYPE . NUMERIC)
```

```
                              (VALUE . 1)
                              (OUT-CONSTRAINT)
                              (IN-CONSTRAINT)
                              (ROLE . PARAMETER))
                    (MINOR-LENGTH (DATATYPE . NUMERIC)
                              (VALUE . 1)
                              (OUT-CONSTRAINT)
                              (IN-CONSTRAINT)
                              (ROLE . PARAMETER))
                    (FACES
                      (VALUE (OUTER-TOP (LIST 270 90 0
MAJOR-HEIGHT))
                             (INNER-TOP (LIST 270 90 0
MINOR-HEIGHT))
                             (BOTTOM (LIST 90 270 80 0))
                             (FRONT (LIST 0 90 270 (/ DEPTH 2)))

                             (BACK (LIST 180 270 90 (/ DEPTH 2)))

                             (OUTER-RIGHT (LIST 90 0 270
MAJOR-LENGTH))
                             (INNER-RIGHT (LIST 90 0 270
MINOR-LENGTH))
                             (LEFT (LIST 270 180 90 0))))
                    (MAKE-CSG-NODE (VALUE .
L-SHAPE.MAKE-CSG-NODE))
                    (SHAPE (VALUE + L-SHAPE)))
                  ("Mcginnis" "20-Jun-90 1:28:35 PDT"))

;;; Declare object FUNCTION-MODULE

(DEFOBJECT FUNCTION-MODULE
           CLASS
           OBJECT
           TEMPLATE
           ((DEFAULT-LANGUAGE (DATATYPE .
CONSTRAINT-LANGUAGE)
                              (VALUE . STRUCTURED))
             (ORIGINATING-CONSTRAINT (DATATYPE .
CONSTRAINT-SOURCE))
             (OBJECT/PURPOSE (DATATYPE . DESIGN-OBJECT))
             (OBJECT/BEHAVIOR (DATATYPE . DESIGN-OBJECT)))

           ("Mcginnis" "6-May-90 21:59:38 PDT"))

;;; Declare object TEXT

(DEFOBJECT TEXT
           CLASS
           DATATYPE
           SYSTEM
           ((DATUM-EDIT (VALUE . SYS/EDITTEXT))
```

```
            (DATUM-PRINT (VALUE . SYS/PRINTTEXT))))
;;; Declare object ROOT-38
(DEFOBJECT ROOT-38 INDIVIDUAL ROOT DESIGN)
;;; Declare object ROOT-83
(DEFOBJECT ROOT-83 INDIVIDUAL ROOT DESIGN)
;;; Declare object ROOT-85
(DEFOBJECT ROOT-85 INDIVIDUAL ROOT DESIGN)
```

III.B. Knowledge Base Support Files

III.B.1. Design-object.lisp

```
(in-package 'slb-cl)

(defun design-object/create/component (object component)
    (createslot object (objectname component)
'design-object)
    (putvalue object (objectname component) component)
    (putvalue component 'parent object)
    (putfacet object (objectname component) 'role
'component)
    (createfacet object (objectname component)
'in-constraint)
    (createfacet object (objectname component)
'out-constraint)
    component)

(defun design-object.create-parameter (object slot facet
name &optional type value)
  (declare (ignore slot facet))
  (if (and (null value)
       (equal type 'numeric))
      (setq value 0))
  (createslot object name type)
  (putvalue object name value)
  (createfacet object name 'in-constraint)
  (createfacet object name 'out-constraint)
  (putfacet object name 'role 'parameter)
  name)

(defun design-object.deletion-procedure (object)
  (mapcar 'deleteobject
       (mapcan #'(lambda (slot)
                 (if (facet? object slot 'in-constraint)
                   (getfacet? object slot 'in-constraint)))

        (listslots object))))

(defun design-object/create (object shape)
  (let ((obj (design-primitives.create shape nil nil
object)))
    (if (not (generalization? shape 'design-object))
    (addgeneralizations object 'design-object))
    obj))

;; This function collects and returns a list of the
components
;; for a particular object
(defun design-object.composed-of (object slot facet)
  (declare (ignore slot facet))
  (objectnames
```

```
   (mapcar #'(lambda (slot) (message* object slot))
   (listslots object 'role 'component))))

(defun design-object/make-object-generic (object)
  (changeobjecttype object 'class))

(defun design-object/active-objects (objects)
  (remove nil
      (mapcar #'(lambda (obj)
                 (if (equal 'active (message* obj
'configuration))
                     obj))
       objects)))

;; This function gives the immediate components of
;; a design object
(defun dh-template/breakdown (object)
  (mapcar #'(lambda (slot)
          (getvalue object slot))
   (remove 'self (slotnames (listslots object 'role
'component)))))

;;;-------------------------------------------------------
-------------------------


(defun design-primitives.deletion-procedures (object)
  (mapc #'(lambda (slot)
            (deleteobject (getvalue object slot)))
   (remove 'self (slotnames (listslots object 'role
'component))))
  (mapc #'(lambda (slot)
            (let ((const-list (append (getfacet? object
slot 'in-constraint)
                              (getfacet? object slot
'out-constraint))))
           (if const-list
               (mapc 'deleteobject const-list))))
   (listslots object 'role)))

;; This method returns the transform vector of
;; a design object
(defun design-primitives.transform-vector (object slot
facet)
  (declare (ignore slot facet))
  (mapcar #'(lambda (slot)
          (message* object slot 'get))
   (list  'x-translation 'y-translation 'z-translation
       'z-rotation 'y-rotation 'x-rotation)))

(defun design-primitive.included-objects (object slot
facet)
```

```
  (declare (ignore slot facet))
  (mapcar #'(lambda (component)
          (list component
            (getvalue? component 'csg-node)))
   (slotnames (listslots object 'role 'component))))

;; This method returns the geometric constraints that
currently
;; affect the design object
(defun design-primitives.orientation (object slot facet)
  (declare (ignore slot facet))
  (let ((constraints
     (remove-duplicates
      (remove nil
          (mapcar #'(lambda (slot)
                      (first
                     (getfacet? object slot
'in-constraint)))
           (listslots object 'role 'geometry))))))
    (mapcar #'(lambda (const)
            (message* const 'phrase 'get))
     constraints)))

(defun design-primitives.create (object slot facet
&optional name)
  (declare (ignore slot facet))
  (let ((obj (createobject name 'individual object
'design)))
    obj))

;; this function determines the cosine angles and length
for the
;; plane equation of a wedge slant face
(defun pos-right-wedge/slant-face (depth length)
  (let* ((slant (sqrt (+ (* depth depth)
                (* length length))))
     (alpha1 (acos (/ length slant)))
     (alpha2 (acos (/ depth slant))))
    (list
     (- 90 (cnvt-to-degrees alpha2))
     (- 360 (- 90 (cnvt-to-degrees alpha1)))
     90
     (* depth (sin alpha2)))))

;; This method builds the CSG tree for some of the
;; design-primitive subclasses
(defun design-primitive.make-node (object slot facet)
  (declare (ignore slot facet))
  (if (and (slot? object 'csg-node)
       (slotvaluep object 'csg-node))
      (message* object 'csg-node)
      (let* ((comp-list (append (list object)
```

```
                    (mapcar #'(lambda (slot)
                                (message* object slot))
                     (listslots object 'role 'component))))

          (node
           (if  (rest comp-list)
             (progn
               (setq base-node (message* (first comp-list)

                                         'make-csg-node))
               (mapc
                #'(lambda (obj)
                    (case (first (message* obj 'shape))
                      ('+ (setq action 'union))
                      ('- (setq action 'difference)))
                    (setq base-node
                      (csgnode* (gentemp "NODE-") action
(list base-node
                                                    (message* obj
'make-node)))))
                (rest comp-list))
               base-node)
             (message* (first comp-list)  'make-csg-node
))))
     (putvalue object 'csg-node node)
     (boun-rep* node)
     (putvalue object 'scene (scene* (gentemp "SCENE-")
(list node)))
     node)))

;; This method builds the CSG node for a particular shape

(defun design-primitive.make-csg-node (object slot facet)

  (declare (ignore slot facet))
  (if (and (slot? object 'csg-node)
           (slotvaluep object 'csg-node))
      (message* object 'csg-node)
      (let* ((node-name (intern (format nil "~s-NODE"
                                  (objectname object)) 'slb-cl))
         (shape (first (last (message* object 'shape))))
         (parameters (mapcar #'(lambda (slot)
                                 (message* object slot 'get))
                       (case shape
                         (cube '(depth length height))
                         (cylinder '(radius height
number-of-app-faces))
                         (rt-ang '(depth length height)))))
         (trans-vect (mk-motion-matrix* (message*  object
'transform-vector))))
     (csgnode* node-name shape parameters :trans
trans-vect))))
```

```
;; This method forms the basic CSG node for the L-SHAPE
design primitive
(defun l-shape.make-csg-node (object slot facet)
  (declare (ignore slot facet))
  (if (and (slot? object 'csg-node)
           (slotvaluep object 'csg-node))
      (message* object 'csg-node)
      (let* ((node-name (intern (format nil "~s-NODE"
                                (objectname object)) 'slb-cl))
         (h (message* object 'major-height 'get))
         (h1 (message* object 'minor-height 'get))
         (l (message* object 'major-length 'get))
         (l1 (message* object 'minor-length 'get))
         (d (message* object 'depth 'get))
         (trans-vect (message*  object 'transform-vector))

         (c1-local (multiply-array-matrix
              (list 0
                    (/ l 2)
                    (/ h1 2))
              (first (get-values
                   (mk-motion-matrix*
                    (list 0 0 0
                          (- (fourth trans-vect))
                          (- (fifth trans-vect))
                          (- (sixth trans-vect))))
                   'matrix-name))))
         (c2-local (multiply-array-matrix
              (list 0
                    (/ l1 2)
                    (/ h 2))
              (first (get-values
                   (mk-motion-matrix*
                    (list 0 0 0
                          (- (fourth trans-vect))
                          (- (fifth trans-vect))
                          (- (sixth trans-vect))))
                   'matrix-name))))
         (trans-1 (mk-motion-matrix*
                  (append (mapcar #'+ c1-local (butlast
trans-vect 3))
                       (list (fourth trans-vect)
                          (fifth trans-vect)
                          (sixth trans-vect)))))
         (trans-2 (mk-motion-matrix*
                  (append (mapcar #'+ c2-local (butlast
trans-vect 3))
                       (list (fourth trans-vect)
                          (fifth trans-vect)
                          (sixth trans-vect)))))
         (node1 (csgnode* (gentemp) 'cu (list d l h1)
```

```
:trans trans-1))
         (node2 (csgnode* (gentemp) 'cu (list d l1 h)
:trans trans-2)))
    (csgnode* node-name 'union (list node1 node2) :fast
t))))


;; This method builds a CSG tree for the composite design
primitive
(defun composite.make-node (object slot facet)
  (declare (ignore slot facet))
  (if (and (slot? object 'csg-node)
       (slotvaluep object  'csg-node))
      (message* object 'csg-node)
      (let* ((nodes (mapcar #'(lambda (component)
                        (message* component 'make-node))
                (slotnames (listslots object 'role
'component))))
         (object-node
          (progn
         (setq base-node (first nodes))
         (mapcar #'(lambda (node)
                     (setq base-node
                    (csgnode* (gentemp "NODE-")
                           'union
                           (list base-node
                              node))))
          (rest nodes))
         base-node)))
    (boun-rep* object-node)
    (putvalue object 'csg-node object-node)
    (putvalue object 'scene (scene* (gentemp "SCENE-")
nodes))
    object-node)))
```

III.B.2. General-Constraints.lisp

```
;; This files contains methods and functions for
constraint use
;; in general

(in-package 'slb-cl)

(defun derived-constraint.preceeding-constraint (object
slot facet)
  (declare (ignore slot facet))
  (let ((dep-obj (message* object (car (listslots object))
'path)))
    (mapcan #'(lambda (constraint)
            (if (equal dep-obj
                    (message* constraint
                        (car (listslots constraint))
'path))
              (list constraint)
             nil))
     (instances 'constraint-source))))

(defun derived-constraint/create()
   (createobject nil 'individual 'derived-constraint
'design))

(defun given-constraint/create ()
  (createobject nil 'individual 'given-constraint
'design))

(defun introduced-constraint/create ()
  (createobject nil 'individual 'introduced-constraint
'design))

;; This method is used to link the  constraints to the in
and out
;; constraint facest of the design object slots that are
used in
;; expressing the particular constraint
(defun constraint-language.link-constraint (object slot
facet)
  (declare (ignore slot facet))
  (mapcar  #'(lambda (slot)
           (let* ((role (getfacet? object slot 'role))
               (path (getfacet? object slot 'path))
               (s-object (first path))
               (s-slot (second path)))
             (if (and s-object s-slot)
              (case role
                (in (addfacet s-object s-slot
'out-constraint
                          (objectnames object)))
```

```
                (out (addfacet s-object s-slot
'in-constraint
                           (objectnames object)))))))
   (listslots object 'path)))

;; This method is used to set the source facets of the
arguments of a
;; constraint.  The source facet holds the constraint that
specified
;; that particular value shown in the argument
(defun set-source-constraints (constraint)
  (mapcar #'(lambda (slot)
          (let* ((path (getfacet? constraint slot 'path))

              (s-const (objectname
                     (first (getfacet? (first path)
                                       (second path)
                                       'in-constraint)))))
            (putfacet constraint slot 'source s-const)))
   (listslots* constraint 'source)))


(defun constraint-role.object-deletion-procedures
(constraint)
  (let ((decision  (getvalue? constraint
'origination-decision)))
    (if decision
    (deleteobject decision))
    (mapc #'(lambda (slot)
          (let ((path (getfacet? constraint slot 'path)))

            (if (and (object? (first path))
                  (slot? (first path) (second path)))
                (let ((role (getfacet? constraint slot
'role)))
                  (case role
                    (in (removefacet (first path)
                                     (second path)
                                     'out-constraint
                                     (objectname constraint)))
                    (out (removefacet (first path)
                                      (second path)
                                      'in-constraint
                                      (objectname
constraint)))))))))
     (listslots constraint 'path ))))

(defun dh-template/active-dependents (object)
  (constraint/active-dependencies object))

;; This function return the constraints that depend on
this constraint
```

```
;; for their current value
(defun constraint/active-dependencies (object)
  (let ((path (getfacet? object
                (first (slotname (listslots* object 'role
'out)))
                'path)))
    (mapcan  #'(lambda (const)
               (let ((obj-path
                   (getfacet const
                          (first (listslots* const 'role
'out))
                          'path)))
                 (if (equal const
                      (first (getfacet (first obj-path)
                                  (second obj-path)
                                  'in-constraint)))
                  (list const))))
     (getfacet? (first path) (second path)
'out-constraint))))

(defun constraint/all-dependencies (object)
  (let ((path (getfacet? object
                (first (slotname (listslots* object 'role
'out)))
                'path)))
    (getfacet? (first path) (second path)
'out-constraint)))

(defun constraint/original-sources (constraint)
  (mapcar #'(lambda (slot)
           (getfacet? constraint slot 'source))
   (listslots* constraint 'source)))

(defun constraint/get-active-dependents (constraint)
  (catch 'get-active-dependents-error
    (constraint/get-active-dependents-1 constraint (list
constraint))))

(defun constraint/get-active-dependents-1 (start
constraints)
  (let ((dependents (mapcar
#'constraint/active-dependencies constraints)))
    (dolist (dep dependents)
      (when (member start dep :test #'same-object)
    (throw 'get-active-dependents-error :fail)))
    (apply #'append
         constraints
         (mapcar #'(lambda (clist)
                  (constraint/get-active-dependents-1
start clist))
          dependents))))
```

```
(defun constraint-language.update (constraint slot facet)

  (declare (ignore constraint slot facet))
  nil)


(defun constraint-phrase.datum-print (constraint slot
facet &optional val (dsp *standard-output*))
  (declare (ignore facet))
  (let ((phrase (message* constraint 'parse)))
    (if phrase
    (prin1 phrase dsp)
    (prin1 (getvalue? constraint 'phrase) dsp))))

(defun constraint-phrase.datum-get (constraint slot facet
&optional path)
  (declare (ignore facet path))
  (let ((phrase (message* constraint 'parse)))
    (if phrase
    phrase
    (getvalue? constraint 'phrase))))
```

III.B.3. Graphical-Constraints.lisp

```
;; This file contains the methods and functions associated
with the
;; graphical constraints of the kb.  Also included is
geometric constraint
;; solver that collects and orders the constraints needed
to be solved for
;; a particular design object

(in-package 'slb-cl)

(defun geometric-constraint-parameter.get (object slot
facet &optional xpath)
  (declare (ignore facet))
  (getfacet? object slot 'path))

(defun geometric-constraint-parameter.print
    (object slot facet  &optional val (dsp
*standard-output*))
  (declare (ignore facet))
  (let* ((path (getfacet? object slot 'path)))
    (if path
    (prinl path dsp))))

(defun spacial-orientation.parse (constraint slot facet)
  (declare (ignore slot facet))
  (format nil "~s ~a ~s"
      (getfacet? constraint
              (first (listslots constraint 'role 'out))
              'path)
      (getvalue? constraint 'relation)
      (getfacet? constraint
              (first (listslots constraint 'role 'in))
              'path)))


;;;;============================================================
=========================
;;;; The following are the individual assert methods for
the orientation
;;;; constraints.  Each assert method performs
manipulations to the x,y,z
;;;; rotations and translations of the dependent object
specified in the
;;;; constraint.
;;;;============================================================
=========================

(defun parallel-surface-constraint.assert (object slot
facet)
  (declare (ignore slot facet))
```

```
  ;;; gather variables and perform rotation
  (let* ((indep-object (first (message* object
'independent-surface 'path)))
     (indep-face (second (message* object
'independent-surface 'path)))
     (indep-face-plane (find-local-plane-equation
indep-object indep-face))
     (indep-face-vector (list
                  (cos (deg-to-rad (first
indep-face-plane)))
                  (cos (deg-to-rad (second
indep-face-plane)))
                  (cos (deg-to-rad (third
indep-face-plane)))))
     (dep-object (first (message* object
'dependent-surface 'path)))
     (dep-face (second (message* object 'dependent-surface
'path)))
     (dep-face-plane (find-local-plane-equation dep-object
dep-face))
     (dep-face-vector (list
                (cos (deg-to-rad (first dep-face-plane)))

                (cos (deg-to-rad (second
dep-face-plane)))
                (cos (deg-to-rad (third
dep-face-plane)))))
     (rot-vector (norm-of-vector
               (cross-product dep-face-vector
                        indep-face-vector)))
     (angle-between (- (angle-between-vectors
indep-face-vector
                               dep-face-vector
                               rot-vector)))
     (delta-rx (cnvt-to-degrees (* angle-between  (first
rot-vector))))
     (delta-ry (cnvt-to-degrees (* angle-between  (second
rot-vector))))
     (delta-rz (cnvt-to-degrees (* angle-between  (third
rot-vector))))
     (rx (message* indep-object 'x-rotation 'get))
     (ry (message* indep-object 'y-rotation 'get))
     (rz (message* indep-object 'z-rotation 'get))
     (affected-axis
      (mapcar #'(lambda (value rot)
               (if (not (or (= value 180)
                            (= value 0)))
                 rot))
        (butlast (find-local-plane-equation indep-object
indep-face))
        '(x-rotation y-rotation z-rotation))))
    (if (member 'x-rotation affected-axis)
```

```
    (message* dep-object 'x-rotation 'put (+ delta-rx
rx)))
    (if (member 'y-rotation affected-axis)
    (message* dep-object 'y-rotation 'put (+ delta-ry
ry)))
    (if (member 'z-rotation affected-axis)
    (message* dep-object 'z-rotation 'put (+ delta-rz
rz)))))


(defun coplanar-surface-constraint.assert (object slot
facet)
  (declare (ignore slot facet))
  ;;; gather variables and perform rotation
  (let* ((indep-object (first (message* object
'independent-surface 'path)))
     (indep-face (second (message* object
'independent-surface 'path)))
     (indep-face-plane (find-local-plane-equation
indep-object indep-face))
     (indep-face-vector (list
                  (cos (deg-to-rad (first
indep-face-plane)))
                  (cos (deg-to-rad (second
indep-face-plane)))
                  (cos (deg-to-rad (third
indep-face-plane)))))
     (dep-object (first (message* object
'dependent-surface 'path)))
     (dep-face (second (message* object 'dependent-surface
'path)))
     (dep-face-plane (find-local-plane-equation dep-object
dep-face))
     (dep-face-vector (list
                (cos (deg-to-rad (first dep-face-plane)))

                (cos (deg-to-rad (second
dep-face-plane)))
                (cos (deg-to-rad (third
dep-face-plane)))))
     (rot-vector (norm-of-vector
               (cross-product dep-face-vector
                         indep-face-vector)))
     (angle-between (- (angle-between-vectors
indep-face-vector
                               dep-face-vector
                               rot-vector)))
     (delta-rx (cnvt-to-degrees (* angle-between  (first
rot-vector))))
     (delta-ry (cnvt-to-degrees (* angle-between  (second
rot-vector))))
     (delta-rz (cnvt-to-degrees (* angle-between  (third
```

```
rot-vector))))
     (rx (message* indep-object 'x-rotation 'get))
     (ry (message* indep-object 'y-rotation 'get))
     (rz (message* indep-object 'z-rotation 'get))
     (affected-axis
      (mapcar #'(lambda (value rot)
                (if (not (or (= value 180)
                             (= value 0)))
                 rot))
       (butlast (find-local-plane-equation indep-object
indep-face))
       '(x-rotation y-rotation z-rotation))))
    (if (member 'x-rotation affected-axis)
    (message* dep-object 'x-rotation 'put (+ delta-rx
rx)))
    (if (member 'y-rotation affected-axis)
    (message* dep-object 'y-rotation 'put (+ delta-ry
ry)))
    (if (member 'z-rotation affected-axis)
    (message* dep-object 'z-rotation 'put (+ delta-rz
rz)))
;;; perform translation of dep-object
    (let* ((indep-face-global-plane
        (find-global-plane-equation indep-object
indep-face))
       (indep-face-global-vector (list
                    (cos (deg-to-rad (first
indep-face-global-plane)))
                    (cos (deg-to-rad (second
indep-face-global-plane)))
                    (cos (deg-to-rad (third
indep-face-global-plane)))))
       (indep-face-normal-vector
        (norm-of-vector indep-face-global-vector))
       (indep-face-dist (abs (first (last
indep-face-plane))))
       (dist-path (getfacet? object 'distance 'path))
       (dep-face-dist (abs (first (last dep-face-plane))))

       (offset (if dist-path
                (message* (first dist-path)
                    (second dist-path) 'get)
                0))
       (indep-translation (butlast (message* indep-object
'transform-vector) 3))
       (dep-translation (butlast (message* dep-object
'transform-vector) 3))
       (face-dist (- (+ (distance-between-parallel-planes
indep-translation
                                          dep-translation

indep-face-normal-vector)
```

```
                    indep-face-dist
                    offset)
                 dep-face-dist)))
       (message* dep-object 'x-translation 'put
          (+ (* face-dist
                (first indep-face-normal-vector))
             (first dep-translation)))
       (message* dep-object 'y-translation 'put
          (+ (* face-dist
                (second indep-face-normal-vector))
             (second dep-translation)))
       (message* dep-object 'z-translation 'put
          (+ (* face-dist
                (third indep-face-normal-vector))
             (third dep-translation))))
    object))

(defun opposing-coplanar-surface-constraint.assert (object
slot facet)
  (declare (ignore slot facet))
  ;;; gather variables and perform rotation
  (let* ((indep-object (first (message* object
'independent-surface 'path)))
     (indep-face (second (message* object
'independent-surface 'path)))
     (indep-face-plane (find-local-plane-equation
indep-object indep-face))
     (indep-face-vector (list
                    (cos (deg-to-rad (first
indep-face-plane)))
                    (cos (deg-to-rad (second
indep-face-plane)))
                    (cos (deg-to-rad (third
indep-face-plane)))))
     (dep-object (first (message* object
'dependent-surface 'path)))
     (dep-face (second (message* object 'dependent-surface
'path)))
     (dep-face-plane (find-local-plane-equation dep-object
dep-face))
     (dep-face-vector (list
                  (cos (deg-to-rad (first dep-face-plane)))

                  (cos (deg-to-rad (second
dep-face-plane)))
                  (cos (deg-to-rad (third
dep-face-plane)))))
     (rot-vector (norm-of-vector
                 (cross-product dep-face-vector
                           indep-face-vector)))
     (angle-between (angle-between-vectors
indep-face-vector
```

```
                                dep-face-vector
                                rot-vector))
     (rot-angle (if (> angle-between pi)
                    angle-between
                    (- 180 angle-between)))
     (delta-rx (cnvt-to-degrees (* rot-angle  (first
rot-vector))))
     (delta-ry (cnvt-to-degrees (* rot-angle  (second
rot-vector))))
     (delta-rz (cnvt-to-degrees (* rot-angle (third
rot-vector))))
     (rx (message* indep-object 'x-rotation 'get))
     (ry (message* indep-object 'y-rotation 'get))
     (rz (message* indep-object 'z-rotation 'get))
     (affected-axis
      (mapcar #'(lambda (value rot)
                 (if (not (or (= value 180)
                              (= value 0)))
                   rot))
       (butlast (find-local-plane-equation indep-object
indep-face))
       '(x-rotation y-rotation z-rotation))))
    (break)
    (if (member 'x-rotation affected-axis)
    (message* dep-object 'x-rotation 'put (+ delta-rx
rx)))
    (if (member 'y-rotation affected-axis)
    (message* dep-object 'y-rotation 'put (+ delta-ry
ry)))
    (if (member 'z-rotation affected-axis)
    (message* dep-object 'z-rotation 'put (+ delta-rz
rz)))
;;; perform translation of dep-object
    (let* ((indep-face-global-plane
        (find-global-plane-equation indep-object
indep-face))
       (indep-face-global-vector (list
                           (cos (deg-to-rad (first
indep-face-global-plane)))
                           (cos (deg-to-rad (second
indep-face-global-plane)))
                           (cos (deg-to-rad (third
indep-face-global-plane)))))
       (indep-face-normal-vector
        (norm-of-vector indep-face-global-vector))
       (indep-face-dist (abs (first (last
indep-face-plane))))
       (dist-path (getfacet? object 'distance 'path))
       (dep-face-dist (abs (first (last dep-face-plane))))

       (offset (if dist-path
                 (message* (first dist-path)
```

```
                    (second dist-path) 'get)
                 0))
       (indep-translation (butlast (message* indep-object
'transform-vector) 3))
       (dep-translation (butlast (message* dep-object
'transform-vector) 3))
       (face-dist (- (- (+
(distance-between-parallel-planes indep-translation
                                                dep-translation

indep-face-normal-vector)
                     offset)
                  indep-face-dist
                  dep-face-dist))))
       (message* dep-object 'x-translation 'put
         (+ (* face-dist
               (first indep-face-normal-vector))
            (first dep-translation)))
      (message* dep-object 'y-translation 'put
         (+ (* face-dist
               (second indep-face-normal-vector))
            (second dep-translation)))
      (message* dep-object 'z-translation 'put
         (+ (* face-dist
               (third indep-face-normal-vector))
            (third dep-translation))))
    object))


;; This method creates a new 'A=VALUE constraint to
specify the offset
;; distance between two objects.  It is invoked only when
the constraint is
;; initially created.
(defun distance-between-surface-constraint.create (object
slot facet)
  (declare (ignore slot facet))
  (let* ((const-source (objectname (first (generalizations
object))))
     (orig-decision (getvalue? object
'originating-decision))
     (dep-object (first (message* object
'dependent-surface 'path)))
     (indep-path (message* object 'independent-surface
'path))
     (val (message* object 'initial-dist-value))
     (slot-name (intern (format nil "offset from ~s ~s"
                        (first indep-path)
                        (second indep-path)) 'slb-cl)))
    (if (not (slot? dep-object slot-name))
    (message* dep-object 'create-parameter nil slot-name
'numeric))
```

```
    (putfacet  object 'distance 'path (list dep-object
slot-name))
    (let ((new-const
       (case const-source
         (derived-constraint (derived-constraint/create))

         (given-constraint (given-constraint/create))
         (introduced-constraint
(introduced-constraint/create)))))
      (addgeneralizations new-const '(parameter a=value))

      (putvalue new-const 'val val)
      (putfacet new-const 'a 'path (list dep-object
slot-name))
      (putvalue new-const 'originating-decision
orig-decision)
      new-const)))

;;*********************************************************
*******************
;;  Link methods used to link constraints to design-object
slots              *
;;*********************************************************
*******************


(defun parallel-surface-constraint.link-constraint (object
slot facet)
  (declare (ignore slot facet))
  (let* ((indep-object (first (message* object
'independent-surface 'path)))
     (indep-face (second (message* object
'independent-surface 'path)))
     (indep-face-axis (second (assoc indep-face (message*
indep-object 'faces))))
     (dep-object (first (message* object
'dependent-surface 'path)))
         (changed-slots '(x-rotation y-rotation
z-rotation)))
    (mapcar #'(lambda (slot)
            (addfacet indep-object slot 'out-constraint
object)
            (addfacet  dep-object slot 'in-constraint
object))
     changed-slots)))

(defun coplanar-surface-constraint.link-constraint (object
slot facet)
  (declare (ignore slot facet))
  (let* ((indep-object (first (message* object
'independent-surface 'path)))
     (dep-object (first (message* object
```

```
'dependent-surface 'path)))
     (indep-face (second (message* object
'independent-surface 'path)))
     (indep-face-plane (find-local-plane-equation
indep-object indep-face))
     (affected-axis
      (mapcar #'(lambda (value trans rot)
                (if (or (= value 180)
                        (= value 0))
                  trans
                  rot))
        (butlast indep-face-plane)
        '(x-translation y-translation z-translation)
        '(x-rotation y-rotation z-rotation))))
    (mapcar #'(lambda (slot)
              (addfacet indep-object slot 'out-constraint
object)
              (addfacet  dep-object slot 'in-constraint
object))
     affected-axis)
    object))

;not sure that this linker links the right slots
(defun perpendicular-surface-constraint.link-constraint
(object slot facet)
  (declare (ignore slot facet))
  (let* ((indep-object (first (message* object
'independent-surface 'path)))
     (indep-face (second (message* object
'independent-surface 'path)))
     (indep-face-axis (second (assoc indep-face (message*
indep-object 'faces))))
     (dep-object (first (message* object
'dependent-surface 'path))))
    (case indep-face-axis
      (x (setq changed-slots '(y-rotation z-rotation)))
      (y (setq changed-slots '(x-rotation z-rotation)))
      (z (setq changed-slots '(x-rotation y-rotation))))
    (mapcar #'(lambda (slot)
              (addfacet indep-object slot 'out-constraint
object)
              (addfacet  dep-object slot 'in-constraint
object))
     changed-slots)))


;;*************************************************************
**********************
;;*             miscellaneous functions
                    *
;;*************************************************************
**********************
```

```
(defun find-normal-vector (object axis)
  (let* ((rx (message* object 'x-rotation))
     (ry (message* object 'y-rotation))
     (rz (message* object 'z-rotation))
     (mot-matrix (mk-motion-matrix* (list 0 0 0 rz ry
rx)))
     (rows (list 'first-row 'second-row 'third-row)))
    (case axis
      ('x (mapcar #'(lambda (row)
                (first (first (get-values mot-matrix
row))))
       rows))
      ('y (mapcar #'(lambda (row)
                (second (first (get-values mot-matrix
row))))
       rows))
      ('z (mapcar #'(lambda (row)
                (third (first (get-values mot-matrix
row))))
       rows)))))


(defun find-global-plane-equation (object face)
  (let* ((plane-equation (second (assoc face (message*
object 'faces))))
     (parameters (slotnames (listslots* object 'role
'parameter)))
     (par-values (mapcar #'(lambda (slot)
                       (message* object slot 'get))
                 parameters))
     (equation (progv parameters par-values
               (eval plane-equation)))
     (x-rot (message* object 'x-rotation 'get))
     (y-rot (message* object 'y-rotation 'get))
     (z-rot (message* object 'z-rotation 'get))
     (new-vect (multiply-array-matrix
             (list (cos (cnvt-to-radians (first
equation)))
                (cos (cnvt-to-radians (second equation)))

                (cos (cnvt-to-radians (third equation))))

             (first (get-values (mk-motion-matrix* (list 0
0 0
                                        (- z-rot)
                                        (- y-rot)
                                        (- x-rot)))
                           'matrix-name)))))
    (list (cnvt-to-degrees (acos (first new-vect)))
      (cnvt-to-degrees (acos (second new-vect)))
      (cnvt-to-degrees (acos (third new-vect)))
```

```
      (fourth equation))))

(defun find-local-plane-equation (object face)
  (let* ((plane-equation (second (assoc face (message*
object 'faces))))
     (parameters (slotnames (listslots* object 'role
'parameter)))
     (par-values (mapcar #'(lambda (slot)
                        (message* object slot 'get))
                 parameters))
     (equation (progv parameters par-values
                (eval plane-equation))))
    (list (first equation)
      (second equation)
      (third equation)
      (fourth equation))))


(defun distance-between-points (point1 point2)
  (let ((d1 (- (first point1) (first point2)))
    (d2 (- (second point1) (second point2)))
    (d3 (- (third point1) (third point2))))
  (sqrt (+ (* d1 d1)
       (* d2 d2)
       (* d3 d3)))))

(defun distance-between-parallel-planes (point1 point2
normal)
  (let ((vector (vector-between-points point1 point2)))
    (/ (dot-product vector normal)
       (vector-length normal))))

(defun vector-length (vector)
  (let ((x (first vector))
    (y (second vector))
    (z (third vector)))
    (sqrt (+ (* x x)
         (* y y)
         (* z z)))))

(defun vector-between-points (point1 point2)
  (list (- (first point2) (first point1))
    (- (second point2) (second point1))
        (- (third point2) (third point1))))



(defun cnvt-to-radians (angle)
  (/ (* angle pi) 180))

(defun cnvt-to-degrees (angle)
  (/ (* angle 180) pi))
```

```
;;;
*************************************************************
*****************
;;; *          geometric constraint sorter - solver
                    *
;;;
*************************************************************
*****************

(defun get-active-geometric-constraints (object)
  (reverse (reduce-list
        (mapcan #'geometric-constraints-needed
                (mapcar #'(lambda (slot)
                          (first (getfacet? object slot
'in-constraint)))
                 (listslots object 'role 'geometry))))))

(defun geometric-constraints-needed (constraint)
  (if  constraint
       (append (list (objectname constraint))
           (mapcan #'geometric-constraints-needed
               (mapcan #'(lambda (slot)
                         (let ((object (first
                                       (message* constraint
slot 'path))))

(get-active-geometric-constraints object)))
                 (listslots constraint 'role 'in))))))


(defun geometric-constraints-for-component (object)
  (append
   (get-active-geometric-constraints object)
   (geometric-constraints-for-component-1 object)))


(defun geometric-constraints-for-component-1 (object)
  (let ((components (mapcar #'(lambda (slot)
                        (message* object slot))
              (listslots object 'role 'component))))
    (append
     (mapcan #'get-active-geometric-constraints
           components)
     (mapcan #'geometric-constraints-for-component-1
           components))))


(defun solve-geometric-constraints (objects)
  (if (atom objects)
      (setq objects (list objects)))
  (mapcar #'(lambda (constraint)
```

```
          (message* constraint 'assert))
    (reduce-list
     (mapcan #'geometric-constraints-for-component
           objects))))


;; This function removes all the graphic representation
information from
;; the design objects as well as resets the values of the
x,y,z
;; translations and rotations.
(defun reset-geometry ()
  (init)
  (mapc  #'(lambda (slot)
         (mapc #'(lambda (object)
                 (if (slot? object slot nil t)
                     (deleteslot object slot)))
           (progeny* 'design-object)))
    (list 'geo-status 'scene 'csg-node 'representation
'relevant-v1v2edges
       'win-origin-x 'win-origin-y 'zoomf))
    (mapc #'(lambda (slot)
         (mapcar #'(lambda (object)
                  (if (slotvaluep object slot)
                   (putvalue object slot 0)))
           (progeny* 'design-object)))
     (list 'x-translation 'y-translation
       'z-translation 'x-rotation 'y-rotation
'z-rotation)))
```

III.B.4 Equational-Constraint.lisp

```
;; This file contains functions and methods for use by the
equational
;; constraints

(in-package 'slb-cl)

;; This function evaluates the equation of a constraint
and returns whether
;; the cnostraint is satisfied (t) or not (nil)
(defun equational.satisfied (constraint slot facet)
  (declare (ignore slot facet))
  (let* ((vars (slotnames (append  (listslots* constraint
'role))))
     (vals (mapcar #'(lambda (slot)
                      (let ((path (getfacet? constraint slot
'path)))
                    (if path
                        (message* (first path)
                                  (second path)
                                  'get)
                        (getvalue? constraint slot))))
         vars))
     (equation (getvalue? constraint 'equation)))
    (progv vars vals
      (eval equation))))


(defun equality-constraint.solve (constraint slot facet)
  (declare (ignore slot facet))
  (let* ((vars (slotnames (listslots* constraint 'role
'in)))
     (vals (mapcar #'(lambda (slot)
                      (getvalue? constraint slot))
         vars))
     (equation (first (rest (rest (getvalue constraint
'equation)))))
     (result (progv vars vals
             (eval equation))))
    (putvalue constraint (first (listslots* constraint
'role 'out)) result)
    result))

(defun equality-constraint.parse (constraint slot facet)
  (declare (ignore slot facet))
  (let* ((vars (slotnames (listslots* constraint 'role)))

     (var-path-alist
      (mapcar #'(lambda (slot)
                (list
                  (slotname slot)
```

```
                (if (getfacet? constraint slot 'path)
                    (getfacet? constraint slot 'path)
                    (getvalue? constraint slot))))
       vars))
     (equation (getvalue constraint 'equation)))
    (prefix-to-infix equation var-path-alist)))


(defun prefix-to-infix (expr alist &optional (outer-op
nil))
  (cond ((null expr) nil)
    ((atom expr)
     (if (cdr (assoc expr alist))
         (cdr (assoc expr alist))
         (list expr)))
    ((and (member outer-op '(* /))
          (not (member (second expr) '(* /))))
     (list
      (append (prefix-to-infix (second expr) alist (first
expr))
           (list (first expr))
           (prefix-to-infix (third expr) alist (first
expr)))))
    (t
     (append (prefix-to-infix (second expr) alist (first
expr))
          (list (first expr))
          (prefix-to-infix (third expr) alist (first
expr))))))


(defun inequality-constraint.solve (constraint slot facet)

  (declare (ignore slot facet))
   (let* ((vars (slotnames (listslots* constraint 'role)))

      (vals (mapcar #'(lambda (slot)
                   (getvalue? constraint slot))
         vars))
      (equation (getvalue constraint 'equation))
      (result (format nil "~s~s"
                (first equation)
                (progv vars vals
                  (eval (list '+
                         (first (rest (rest
equation))))))))))
     (putvalue constraint (first (listslots* constraint
'role 'out)) result)
    result))

(defun quality-constraint.parse (constraint slot facet)
  (declare (ignore slot facet))
```

```
  (format nil "~{~A ~}"
      (remove nil
           (list
            (getvalue? constraint 'quality-expr)
            (getfacet? constraint 'independent-feature
'path)))))


(defun equational-constraint/create/subclass (name
var-list type)
  (if (null (listp var-list))
      (list var-list))
  (let ((object (createobject name 'class type  'template

                    `(((,(first var-list))
                     (datatype .
numeric-constraint-parameter)
                     (role . out)
                     (path)
                     (source))))))
    (mapcar #'(lambda (var)
            (fillslots object
              `(((,var)
              (datatype . numeric-constraint-parameter)
              (role . in)
              (path)
              (source)))))
     (rest var-list))
    object))

;; This method automatically maintains and updates the
equality constraints
;; as new constriants are added to the kb. This is done by
creating new
;; instances of those constraints that will be affected by
the addition or
;; modification information into the kb.
(defun equality-constraint.update (constraint slot facet)

  (declare (ignore slot facet))
  (let ((dependents (constraint/get-active-dependents
constraint))
    (decision (getvalue? constraint
'originating-decision)))
    (if (equal dependents :fail)
    :fail
    (mapcar #'(lambda (const)
                (let* ((generalizations (generalizations
const))
                     (new-const
                      (case (objectname (first
generalizations))
```

```
                    (given-constraint
                     (given-constraint/create))
                    (introduced-constraint
                     (introduced-constraint/create))
                    (derived-constraint
                     (derived-constraint/create)))))
                (addgeneralizations new-const
                                    (list (second
generalizations)
                                          (third generalizations)))

                (uncache const new-const)
                (mapcar #'(lambda (slot)
                           (let* ((path (getfacet new-const
slot 'path))
                                  (value (message* (first
path)
                                                   (second path)
                                                   'get)))
                             (putvalue new-const slot
value)))
                 (listslots* new-const 'path))
                (putvalue new-const
                          (slotname (first
                                     (listslots* new-const 'role
'out)))
                          (message new-const 'solve))
                (if decision
                    (putvalue new-const
'originating-decison decision))
                (message* new-const 'link-constraint)
                (set-source-constraints new-const)
                new-const))
     (cdr (reduce-list dependents))))))
```

III.B.5. Textual-Constraint.lisp

```
;; This file contains methods and functions used by the
textual constraints
(in-package 'slb-cl)

(defun textual.solve (constraint slot facet)
  (declare (ignore slot facet))
  (objectname (first
           (getfacet? constraint
                (first (listslots* constraint 'role 'out))

                'path))))

(defun status-language.solve (constraint slot facet)
  (declare (ignore slot facet))
  (let ((constraint (getvalue constraint
'constraint-affected))
    (status (getvalue constraint 'status)))
    (putvalue constraint 'status status)))


(defun structured.parse (constraint slot facet)
  (declare (ignore slot facet))
  (let ((sequence (getvalue? constraint 'sequence))
    (decomposition (getvalue? constraint 'decomposition))

    (alternatives (getvalue? constraint 'alternatives)))
    (format nil "~{~A ~}"
        (remove nil
             (list (format nil "~{~A ~}"
                      (remove nil
                           (list
                            (getvalue? constraint 'object)

                            (getvalue? constraint 'action)

                            (getvalue? constraint
'receiver)
                            (getvalue? constraint
'location)
                            (getvalue? constraint
'action-qualifier))))
                (if sequence
                    (format nil "SEQUENCE: ~a"
(series-parse sequence)))
                (if decomposition
                    (format nil "DECOMPOSITION: ~a"
(series-parse decomposition)))
                (if alternatives
                    (format nil "ALTERNATIVES: ~a"
(series-parse alternatives)))))))))
```

```
(defun series-parse (series)
  (setq counter 0)
  (mapcar #'(lambda (constraint)
          (setq counter (+ counter 1))
          (format nil "~D. ~A"
            counter
            (message* constraint 'parse)))
   series))

(defun simple.parse (constraint slot facet)
  (declare (ignore slot facet))
  (format nil "~{~A ~}"
      (remove nil
           (list
            (getvalue? constraint 'instantiation/relation)

            (getvalue? constraint 'independent-feature)
            (getvalue? constraint
'independent-feature2)))))

(defun function.parse (constraint slot facet)
  (declare (ignore slot facet))
  (remove nil
      (mapcar #'(lambda (slot)
                (let ((fm (find-last-successor
                          (getvalue? constraint slot))))
                  (if fm
                      (message* fm 'parse))))
       (listslots constraint 'datatype
'function-module))))

(defun production-constraint-language.parse (constraint
slot facet)
  (declare (ignore slot facet))
  (remove nil
      (mapcar #'(lambda (slot)
                (format nil "~s: ~a"
                  (slotname slot)
                  (getvalue? constraint slot)))a
       (listslots constraint 'role
'production-parameter))))


(defun object-form-language.parse (constraint slot facet)

  (declare (ignore slot facet))
  (let ((name (getvalue? constraint 'name))
    (shape (getvalue? constraint 'shape))
```

```
    (status (getvalue? constraint 'status)))
    (format nil
        "~:[~;~s ~]~:[~; with shape ~s ~]~:[~; and status
~s ~]"
        name name shape shape status status)))


;; This method is used by a specialized expr datatype that
returns a
;; formulated phrase of the textual constraint expression

(defun text-valued.datum-get (object slot facet &optional
path)
  "SYS/GETFUNCTION responds to a GET message with
GETVALUE."
  ;; From version of 18-Sep-86 by jaw
  (declare (ignore facet))
  (let ((constraints (objectname (getfacet? object slot
'in-constraint))))
    (if constraints
    (let* ((val  (dolist (const constraints)
                 (if (not (equal 'in-active
                         (getvalue? const 'status)))
                  (return (message* const 'parse))))))
      val)
    (getvalue object slot))))


(defun text-valued.datum-print  (object slot facet
&optional path  (dsp *standard-output*))
  "SYS/GETFUNCTION responds to a GET message with
GETVALUE."
  ;; From version of 18-Sep-86 by jaw
  (declare (ignore facet))
  (let ((constraints (objectname (getfacet? object slot
'in-constraint))))
    (if constraints
    (let* ((val  (dolist (const constraints)
                 (if (not (equal 'in-active
                         (getvalue? const 'status)))
                  (return (message* const 'parse))))))
      (prin1 val dsp))
    (prin1 (getvalue object slot) dsp))))

;; This method creates a new design object within the
knowledge base
(defun create-object.solve (constraint slot facet)
  (declare (ignore slot facet))
  (let* ((obj-name (getvalue? constraint 'name))
     (obj-shape (getvalue? constraint 'shape))
     (parent (getvalue? constraint 'parent))
     (components (getvalue? constraint 'components))
```

```
          (object (design-object/create obj-name
obj-shape)))
    (putfacet constraint 'object 'path
          (list obj-name 'form))
    (if parent
    (case parent
      (*generic-object* (design-object/make-object-generic
object))
      (otherwise (design-object/create/component parent
obj-name)
                 (putvalue object 'parent parent))))
    (if components
    (mapcar #'(lambda (comp)
                (design-object/create/component obj-name
comp))
     components))
    (objectname object)))


;; This method modifies the name, shape, components,
and/or parents of an
;; existing design object
(defun modify-object-form.solve (constraint slot facet)
  (declare (ignore slot facet))
  (let* ((new-name (getvalue? constraint 'name))
     (new-shape (getvalue? constraint 'shape))
     (new-parent (getvalue? constraint 'parent))
     (new-components (getvalue? constraint 'components))
     (old-name (objectname (first (getfacet constraint
'object 'path))))
     (old-shape (objectname (primarygeneralization
old-name)))
     (old-parent (getvalue? old-name 'parent))
     (old-components (message* old-name 'composed-of)))
    (if (not (equal new-name old-name))
    (progn
      (renameobject old-name new-name)
      (putfacet constraint 'object 'path
            (list new-name 'form))
      (if old-parent
          (renameslot old-parent old-name new-name))))
    (if (not (equal new-shape old-shape))
    (progn
      (addgeneralizations new-name new-shape)
      (removegeneralizations new-name old-shape)
      (sortgeneralizations new-name nil
                   #'(lambda (g1 g2)
                       (generalization? g1
'design-primitives)))))
    (let ((comp-added (set-difference new-components
old-components))
      (comp-removed (set-difference old-components
```

```
new-components)))
       (if comp-added
       (mapcar #'(lambda (comp)
                 (design-object/create/component new-name
comp))
        comp-added))
       (if comp-removed
       (mapcar #'(lambda (comp)
                 (deleteslot new-name comp))
        comp-removed)))
    (if new-parent
    (if (not (equal new-parent old-parent))
        (if (equal new-parent '*generic-object*)
         (progn
           (design-object/make-object-generic new-name)
           (deleteslot new-name 'parent)
           (if (equal new-name old-name)
               (deleteslot old-parent old-name)
               (deleteslot old-parent new-name)))
         (progn
           (putvalue new-name 'parent new-parent)
           (design-object/create/component new-parent
new-name)
           (if old-parent
               (deleteslot old-parent new-name))))))
    new-name))
```

III.B.6. New-Functions.lisp

```
(in-package 'SLB-CL)

(defun delete-instances-all (object)
  (mapcar 'deleteobject (progeny* object 'design)))

;; This function deletes all design instances from the
;; knowledge base
(defun cleanup ()
  (list (delete-instances-all 'constraint-source)
    (delete-instances-all 'design-object)
    (delete-instances-all 'decision)
    (delete-instances-all 'function-module)))

(defun delete-object-all (object)
  (mapcar 'deleteobject (progeny* object)))

(defun delete-slots-all (object)
  (mapcar #'(lambda (slot)
         (deleteslot object slot))
      (listslots object)))

(defun decision.preceeding-decision (object slot facet)
  (declare (ignore slot facet))
  (mapcar #'(lambda (const)
          (message* const 'originating-decision))
   (derived-constraint.preceeding-constraint
    (message* object 'resulting-constraint) nil nil)))

(defun decision/create ()
  (let ((decision (createobject nil 'individual 'decision
'design))
    (constraint (derived-constraint/create)))
    (addvalue decision 'resulting-constraint constraint)
    (list decision constraint)))

;; This is a specialized lisp datatype that returns as its
value the
;; result of evaluating the lisp expression contained as
its value
(defun sys/printlisp-p-lisp (object slot facet &optional
val (dsp *standard-output*))
  "SYS/PRINTLISP is the printing function for an item of
the LISP datatype.  DSP is an optional
  stream."
  ;; From version of 10-Sep-84 by rgs
  (declare (ignore  facet dsp val))
  (let ((fn-val (message* object slot)))
    (write fn-val :stream dsp :escape t)))

;; This is a specialized expr datatype method that finds
```

```
the current value
;; of a design object attribute by determining the most
recent active
;; in-constraint and solving for it
(defun numeric.datum-print (object slot facet &optional
val (dsp *standard-output*))
  "the printing function for an item of the NUMERIC
datatype.  DSP is an optional
  stream."
  ;; From version of 11-Nov-82 by rgs/11-Jul-86 by jaw
  (declare (ignore facet))
  (let ((constraints (objectname (getfacet? object slot
'in-constraint))))
    (if constraints
    (let* ((val  (dolist (const constraints)
                 (if (not (equal 'in-active
                         (getvalue? const  'status)))
                  (return (message* const 'solve))))))
      (prin1 val dsp))
    (prin1 (getvalue object slot) dsp))))

;; See numeric.datum-print
(defun numeric.datum-get (object slot facet &optional
path)
  "responds to a GET message with GETVALUE."
  ;; From version of 18-Sep-86 by jaw
  (declare (ignore facet path))
  (let ((constraints (objectname (getfacet? object slot
'in-constraint))))
    (if constraints
    (dolist (const constraints)
      (if (not (equal 'in-active
                 (getvalue? const  'status)))
          (return (message* const 'solve))))
    (getvalue object slot))))


(defun decision.deletion-procedures (object)
  (mapcar #'(lambda (result)
          (if (object? result)
            (deleteobject result)))
   (getvalue object 'resulting-constraint)))


(defun reduce-list (lis)
  (let ((lis (reverse lis))
    (new-lis nil))
    (dolist (item lis)
      (pushnew item new-lis :test #'eql))
    new-lis))

(defun function-module/create ()
```

```
  (createobject (createobjectname 'function-module)
         'individual
         'function-module
         'design))

(defun updated-series (constraint slot)
  (let ((old-sequence (getvalue? constraint slot)))
    (mapcar #'(lambda (item)
            (find-last-successor item))
     old-sequence)))

(defun find-last-successor (object)
  (cond ((null object) nil)
    (t (let ((successor (getvalue? object 'successor)))
         (if successor
          (find-last-successor successor)
          object)))))

(defun uncache (source target)
  (mapcar #'(lambda (slot)
          (if (slotvaluep source slot)
            (linkslot target slot source)))
   (slotnames (listslots* source 'role)))
  (putvalue source 'successor target))


;; This function determines if a constraint is current or
not by examining
;; its status and comparing it to the in-constraints of
the dependent
;; argument specified by the constraint
(defun current? (constraint)
  (let ((path (getfacet? constraint
               (first (listslots* constraint 'role 'out))

               'path)))
    (if path
    (if (object? (first path))
        (let* ((in-consts
             (if (slot? (first path)
                     (second path))
              (getfacet? (first path)
                      (second path)
                      'in-constraint)
              (list constraint)))
            (active (dolist (const in-consts)
                   (if (null (equal 'in-active
                                (getvalue? const 'status)))

                     (return const)))))
          (same-object constraint active))
        nil)
```

```
    t)))

;;;Do you want this function to be destructive (i.e., to
destructively
;;;modify the list?).  If so, then the following code will
work:

(defun ninsert-after (newitem element list)
  (let ((tail (member element list)))
    (cond (tail
            (rplacd tail (cons newitem (cdr tail)))
            list)
          (t (error "~S is not in ~S." element list)))))

;;;If you don't want the function to make destructive
modifications to
;;;the list, then this will work:

(defun insert-after (newitem element list)
  (cond ((null list) nil)
        ((eq element (car list))
         (cons element (cons newitem (cdr list))))
        (t (cons (car list)
                 (insert-after newitem element (cdr
list))))))

;;;Note that in this second version, if ELEMENT is not
found in LIST, the
;;;function simply returns (a copy of) LIST without
reporting an error.
;;;It would be easy to make NINSERT-AFTER do this too, of
course.
```

III.B.7. Draw-Function.lisp

```
(in-package 'slb-cl)

;; This function draws csg-nodes to the vantage window
(defun draw (node)
  (fit-screen* (boun-rep* node)))


;; This function draws objects to the vantage window given

;; one or more design objects
(defun draw-objects (objects)
  (if (atom objects)
      (setq objects (list objects)))
  (solve-geometric-constraints objects)
  (if (and (null (rest  objects))
       (not (generalization? (first objects) 'composite)))

      (draw (message* (first objects) 'make-node))
      (draw (let ((nodes (mapcan #'(lambda (obj)
                        (message* obj 'make-node))
                  objects)))
          (if (listp nodes)
           (if (rest nodes)
               (progn
              (setq base-node (first nodes))
              (mapcar #'(lambda (node)
                          (setq base-node
                         (csgnode* (gentemp "NODE-")
                                'union
                                (list base-node
                                   node))))
               (rest nodes))
              base-node)
               (first nodes))
           nodes)))))

;; This function creates a bitmap for the representantion
slot
;; of a design object
(defun make-representation (object)
  (if (or  (null (slot? object 'scene))
       (null (slotvaluep object 'scene)))
      (draw-objects object))
  (setq *relevent-scene-v1v2edges* ())
  (fit-scene-to-screen* (message* object 'scene)
              'display-camera)
  (fillslots  object
          `((representation
          (datatype . BITMAP)
          (value . ,(make-bitmap
```

```
                    :width *vantage-window-width*
                    :height  *vantage-window-height*)))
          (relevant-v1v2edges
           (value . ,*relevant-scene-v1v2edges*)
           (datatype . EXPR))
          (win-origin-x
           (datatype . EXPR)
           (value . ,*win-origin-x**))
          (win-origin-y
           (datatype . EXPR)
           (value . ,*win-origin-y**))
          (zoomf
           (datatype . EXPR)
           (value . ,*zoomf*))))
  (rasterop *vantage-window* 0 0
        (message object 'representation) 0 0
        *vantage-window-width*
        *vantage-window-height*))

;;*********************************************************
**********************
;; The following function were extrated from Rich Charon's
Scene-handler.lisp
;; code.  They are used in drawing objects to the vantage
window.
;;*********************************************************
**********************

(defvar *relevant-scene-v1v2edges* ())

(defun fit-scene-to-screen* (scene camera)
  (let* ((nodes-in-scene (car (v-get-values scene
'CSG-NODE-LIST)))
     (minx most-positive-fixnum)
     (maxx most-negative-fixnum)
     (miny most-positive-fixnum)
     (maxy most-negative-fixnum)
     diffx diffy)
    (dolist (body-node nodes-in-scene)
      (let* ((body-name (v-get-value body-node
'boundary-rep))
         (ver-list (v-get-values body-name
'body-vertex-list)))
    (dolist (ver ver-list)
      (let* ((xyz (v-get-value ver 'xyz-value))
          (xyz1 (project-point camera (car xyz) (cadr xyz)
(caddr xyz)))
          (xv   (car xyz1))
          (yv   (cadr xyz1)))
        (if (< xv minx) (setq minx xv))
        (if (> xv maxx) (setq maxx xv))
        (if (< yv miny) (setq miny yv))
```

```
        (if (> yv maxy) (setq maxy yv))))))
    (setq diffx (abs (- minx maxx)))
    (setq diffy (abs (- miny maxy)))
    (set-new-display-parameters diffx diffy (/ (+ minx
maxx) 2) (/ (+ miny maxy) 2) .8)
    ;;(break "list = ~S~%" nodes-in-scene)
    (dolist (body-node nodes-in-scene)
      (scene-draw-body*  body-node camera))))

(defun scene-draw-body* (node camera)
  (let* ((name (v-get-value node 'boundary-rep))
     (edge-l  (v-get-values name 'body-edge-list))
     (face-l  (v-get-values name 'body-face-list)))
    (dolist (i edge-l) (scene-draw-edge node i camera))
;(break "i = ~s~%" i)
    (if (= 4 *hid-level*) (dolist (i face-l)
(scene-draw-face i))))
  (values node))


(defun scene-draw-edge (node edge camera)
  (if (null (v-get-value edge 'edge-children))
      (let* ((dash (if (not (= 1 *hid-level*))
(not-visible-p edge)))
         ;t means  not-visibe
         (v1 (v-get-value edge 'p-vertex))
         (v2 (v-get-value edge 'n-vertex))
         (xyz1 (v-get-value v1 'xyz-value))
         (xyz2 (v-get-value v2 'xyz-value))
         (pp1 (project-point camera (car xyz1) (cadr xyz1)

                    (caddr xyz1)))
         (pp2 (project-point camera (car xyz2) (cadr xyz2)

                    (caddr xyz2)))
         (x (floor (+ (* *zoomf* (car pp1))
*win-origin-x**)))
         (y (floor (+ (* *zoomf* (cadr pp1))
*win-origin-y**)))
         (x1 (floor (+ (* *zoomf* (car pp2))
*win-origin-x**)))
         (y1 (floor (+ (* *zoomf* (cadr pp2))
*win-origin-y**))))
    (v-replace-values  v1 'display-xy (list x y))
    (v-replace-values  v2 'display-xy (list x1 y1))
    (push (list v1 v2 edge) *displayed-v1v2edge-list*)
    (push v1 *displayed-vertex-list*)
    (push v2 *displayed-vertex-list*)
    (unless (and (or (= 4 *hid-level*) (= 3 *hid-level*))
dash)
      (vantage-line (car pp1) (cadr pp1) (car pp2) (cadr
pp2) :dash dash)
```

```
      (push (list (list (car pp1) (cadr pp1))
                  (list (car pp2) (cadr pp2)) edge node)
*relevant-scene-v1v2edges*)))
      ;;(break "edge = ~S~%node = ~S~% pp1 = ~S, pp2 =
~S~%" edge node pp1 pp2)
      (dolist (edge (v-get-value edge 'edge-children))
    (scene-draw-edge edge))))
```

III.B.8. Menu-Interface.lisp

```
;; This file contains all the interface functions needed
for the design
;; recorder.

(in-package 'slb-cl)

(defvar *menu-position* (make-position :x 400 :y 200))
(defvar *menu-font* (open-font 'Helvetica 'mrr 14))
(defvar *title-font* (open-font 'Helvetica 'brr 14))

(defvar *prompt-window*
    (make-window-stream  :left 600 :top 300 :width 700
:height 200
                :title "Prompt Widow"
                :activate-p nil))

(defvar *top-level-menu*
  '(design-object constraint decision quit))


(defvar *design-object-menu*
  '(examine-object
    delete-object
    change-object-name
    clean-knowledge-base
    reset-geometry
    draw-object
    return))

(defvar *decision-menu*
  '(make-decision
    delete-decision
    examine-decision
    return))

(defvar *constraint-menu*
  '(make-constraint
    delete-constraint
    examine-constraint
    return))

(defun start-interface ()
  (do ((top-choice (mt/singlechoicemenu *top-level-menu*
:title "DESIGN-RECORDER"
                          :position *menu-position*
                          :font *menu-font*
                          :title-font *title-font*)
             (mt/singlechoicemenu *top-level-menu* :title
"DESIGN-RECORDER"
                          :position *menu-position*
```

```
                          :font *menu-font*
                          :title-font *title-font*)))
      ((equal top-choice 'quit) "exited")
    (case top-choice
      (design-object (design-object-interface))
      (constraint (constraint-interface))
      (decision (decision-interface)))))

(defun design-object-interface()
  (do ((do-choice (mt/singlechoicemenu
*design-object-menu* :title "DESIGN-OBJECT-INTERFACE"
                            :position *menu-position*
                            :font *menu-font*
                            :title-font *title-font*)
            (mt/singlechoicemenu *design-object-menu*
:title "DESIGN-OBJECT-INTERFACE"
                            :position *menu-position*
                            :font *menu-font*
                            :title-font *title-font*)))
      ((equal do-choice 'return) t )
    (case do-choice
      (examine-object (examine-interface (get-object
'design-object)))
      (delete-object (mapcar 'deleteobject
(mt/multichoicemenu
                               (objectnames(progeny*
'design-object))
                               :abort-p t
                               :title "Select Design
Object(s)")))
      (change-object-name (let* ((object (get-object
'design-object))
                     (new-name (get-item :prompt "Enter New
Name"
                                         :default (objectnames
object))))
                  (renameobject object new-name)))
      (clean-knowledge-base
       (if (equal (get-item :prompt "Do you really want to
clean all the KB?? (yes/no)"
                  :default 'no)
           'yes)
       (cleanup)))
      (reset-geometry
       (if (equal (get-item :prompt "Do you want to  reset
all graphic representation (yes/no)"
                  :default 'no)
           'yes)
       (reset-geometry)))
      (draw-object
       (let ((obj (get-object 'design-object)))
     (if obj
```

```
        (progn
          (make-representation obj)
          (examine-interface obj)))))))

(defun create-parameter-interface ()
  (let ((object (mt/singlechoicemenu (append (progeny*
'design-object)
                                             '(design-object))
                           :title "Choose Object"
                           :position *menu-position*
                           :font *menu-font*
                           :title-font *title-font*))
    (type (mt/singlechoicemenu '(numeric text-valued)
           :title "Choose value datatype for parameter"
           :position *menu-position*
           :font *menu-font*
           :title-font *title-font*)))
    (message* object
          'create-parameter
          nil
          (get-item :prompt "Parameter Name"
              :null-response-ok-p nil)
          type)))

(defun design-object-create-interface ()
  (let* ((parent (mt/singlechoicemenu (append
                            (objectnames (instances*
'design-object 'design))
                            (list '*generic-object*))
                           :title "Choose Parent Object or
<abort> for none"
                           :font *menu-font*
                           :title-font *title-font*
                           :abort-p t))
     (obj-name (get-item :prompt "Object Name"
                     :default (createobjectname
"design-object")))
     (obj-shape (mt/singlechoicemenu
                (objectnames (subclasses*
'design-primitives))
                :title (format nil "Choose shape for ~S"
obj-name)
                :position *menu-position*
                :font *menu-font*
                :title-font *title-font*))
     (object (design-object/create obj-name obj-shape)))
    (if parent
    (case parent
      (*generic-object* (design-object/make-object-generic
object))
      (otherwise (design-object/create/component parent
obj-name)
```

```
            (putvalue object 'parent parent))))
    (if (equal obj-shape 'composite)
    (let ((components (mt/multichoicemenu
                  (remove obj-name
                       (objectnames (progeny*
'design-object)))
                  :abortp t
                  :title (format nil "Select components for
~S" obj-name)
                  :title-font *title-font*)))
      (mapcar #'(lambda (comp)
                (design-object/create/component obj-name
comp))
       components)))
    (objectnames object)))


(defun get-item (&rest askitem-args)
  (activate *prompt-window*)
  (let ((val (with-ttyin-environment *prompt-window*
           (apply 'askitem askitem-args))))
    (deactivate *prompt-window*)
    val))

(defun get-text (&rest askitem-args)
  (activate *prompt-window*)
  (let ((val (with-ttyin-environment *prompt-window*
           (apply 'askstring askitem-args))))
    (deactivate *prompt-window*)
    val))

(defun get-object (object &optional title)
  (if (null title)
      (setq title "Choose One"))
  (mt/singlechoicemenu (objectnames (progeny* object
'design))
                :title title
                :font *menu-font*
                :title-font *title-font*
                :abort-p t))

(defun decision-interface()
  (do ((decision-choice (mt/singlechoicemenu
*decision-menu* :title "DECISION"
                              :position *menu-position*
                              :font *menu-font*
                              :title-font *title-font*)
             (mt/singlechoicemenu *decision-menu* :title
"DECSION"
                              :position *menu-position*
                              :font *menu-font*
                              :title-font *title-font*)))
```

```
      ((equal decision-choice 'return) t )
    (case decision-choice
      (make-decision (make-decision-interface))
      (delete-decision (delete-decision-interface))
      (examine-decision (examine-decision-interface)))))

(defun delete-decision-interface ()
  (mapcar 'deleteobject
       (mt/multichoicemenu
        (choose-decision-menu)
        :abort-p t
        :title "Select Decision(s)"
        :centered nil)))

(defun choose-decision-menu ()
  (mapcar #'(lambda (decision)
          (list
            (format nil
                 "~s with resulting: ~{~A ~}"
                 decision
                 (mapcar #'(lambda (const)
                           (format nil "~s => ~a"
                             (objectname const)
                             (message* const 'parse)))
                  (getvalue? decision
'resulting-constraint)))
            decision))
   (objectnames(instances 'decision 'design))))


(defun make-decision-interface ()
  (let* ((dcpair (decision/create))
     (decision (first dcpair))
     (result (second dcpair))
     (input (mt/multichoicemenu (choose-constraint-menu
result)
                        :centered nil
                        :title
                        (format nil
                             "Select Input Constraint(s)
for ~a with  ~a"
                             (objectname decision)
                             (objectname result))))
     (rationale (get-text :prompt "Enter Decision
Rationale"))
     (new-consts (define-constraint-interface result)))
    (if new-consts
    (progn
      (if (member :fail new-consts)
          (progn
         (get-item
          :prompt
```

```
          "looping constraint in dependences
         - aborting decision (push return to continue)"
          :default t)
         (deleteobject decision)
         (deleteobject result))
          (progn
         (putvalue decision 'resulting-constraint
                new-consts)
         (putvalue decision 'input-constraints input)
         (putvalue decision 'rationale rationale)
         (mapcar #'(lambda (new-cons)
                     (putvalue new-cons
'originating-decision decision))
          new-consts))))
    (progn
      (deleteobject decision)))))

(defun choose-constraint-menu (&optional result
const-list)
  (if (null const-list)
      (setq const-list
        (mapcan #'(lambda (const)
                  (if (current? const)
                    (list const)))
          (progeny* 'constraint-source 'design))))
  (mapcar #'(lambda (const)
          (let* ((dep-obj (getfacet? const
                                (first (listslots const 'role
'out)) 'path))
                 (type (objectname (third (generalizations
const))))
                 (value (message* const 'parse)))
             (list (format nil "~45a   ~tOBJ:~50:a
~tVALUE:~:a"
                     const  dep-obj value) const)))
   (objectnames (remove result const-list))))

(defun define-constraint-interface (constraint)
  (let* ((role (choose-constraint-category
'constraint-role
                              "Choose Constraint Role")))
    (if role
    (progn
      (addgeneralizations constraint role)
      (let* ((language  (choose-constraint-category
                  (getvalue constraint 'default-language)

                  "Choose Constraint Language")))
        (if language
         (progn
           (if (listp language)
               (setq language
```

```
                    (create-new-constraint-interface (second
language))))
            (addgeneralizations constraint language)
            (if
             (case language
               (structured
                (make-structured-constraint-interface
constraint role))
               (object-form-language
                (create-object-constraint-interface
constraint role))
               (status-language
                (make-status-constraint-interface
constraint role))
               (function-constraint-language
                (make-function-constraint-interface
constraint role))
               (production-constraint-language
                (make-production-constraint-interface
constraint role))
               (otherwise
                (fill-constraint-slot-interface
constraint)))
             (progn
               (let* ((created-const (message* constraint
'create))
                      (all-new-consts
                       (mapcan #'(lambda (const)
                                   (append (message* const
'update)
                                           (list const)))
                         (remove nil (list created-const
constraint)))))
                 (mapc #'(lambda (const)
                           (message const 'solve)
                           (set-source-constraints const)
                           (message* const 'link-constraint))

               all-new-consts)
                 all-new-consts)))))))))


(defun choose-constraint-category (class &optional title)

  (let ((options
     (cond ((generalization? class 'equational)
         (list '*new-constraint*))
           ((generalization? class 'graphical)
         (list '*new-constraint*))
```

```
          (t ())))))
    (if (subclasses? class 'template)
    (let ((cat (mt/singlechoicemenu (append
                          (objectnames (progeny class))
                          options)
                         :title title
                         :abort-p t)))
      (if (equal cat '*new-constraint*)
          (list cat class)
          (choose-constraint-category cat title)))
    class)))

(defun fill-constraint-slot-interface (constraint)
  (do ((slot (mt/singlechoicemenu
          (define-constraint-menu constraint)
          :title constraint
          :position *menu-position*
          :font *menu-font*
          :title-font *title-font*
          :abort-p t)
         (mt/singlechoicemenu
          (define-constraint-menu constraint)
          :title constraint
          :position *menu-position*
          :font *menu-font*
          :title-font *title-font*
          :abort-p t)))
      ((or (equal slot '*done*)
       (null slot))
       (if (null slot)
       nil
       t))
    (let ((datatype (getdatatype constraint slot)))
      (case (objectname datatype )
    ((numeric-constraint-parameter
      geometric-constraint-parameter
      textual-constraint-parameter)
     (let ((path (putfacet constraint slot 'path
                    (choose-object-parameter-pair
                   "Choose object/parameter"
                   datatype))))
       (if (equal (second path) '**new-parameter**)
           (setq path (list (first path)
                    (create-parameter-interface))))
       (if (same-object datatype
                  'geometric-constraint-parameter)
           (putvalue constraint slot path)
           (putvalue constraint slot
               (message* (first path) (second path)
'get)))
       (putfacet constraint slot 'path path)))
    (design-object
```

```
     (putvalue constraint slot
            (mt/singlechoicemenu
             (objectnames (progeny* 'design-object
'design))
             :title "Choose Design Object")))
    (expr
     (putvalue constraint slot
            (get-item :prompt (format nil "Enter expr for
~s"
                              (slotname slot))
                      :default (getvalue constraint slot))))

    (text
     (putvalue constraint slot
            (get-text :prompt (format nil "Enter expr for
~s"
                              (slotname slot))
                      :default (getvalue constraint slot))))

    (otherwise
     (putvalue constraint slot
            (get-text :prompt (format nil "Enter text for
~s"
                              (slotname slot))
                      :default (getvalue constraint
slot)))))))


(defun define-constraint-menu (constraint)
  (let ((slot-path-vals (mapcar #'(lambda (slot)
                           (list slot
                           (getfacet? constraint slot 'path)

                           (message*  constraint slot
'get)))
                (slotnames (listslots* constraint
'role)))))
    (append (mapcar #'(lambda (set)
                (list (format nil "~a [~:a]   (~:a)"
                           (first set) (second set) (third
set))
                      (first set)))
         slot-path-vals)
        '(*done*))))


(defun choose-object-parameter-pair (request &optional
datatype)
  (if (null datatype)
      (setq datatype 'numeric-constraint-parameter))
  (let* ((object (mt/singlechoicemenu (objectnames
(progeny* 'design-object 'design))
```

```
                          :title request))
     (parameter (mt/singlechoicemenu
                 (case (objectname datatype)
                   (numeric-constraint-parameter
                 (append
                  (slotnames (listslots* object 'datatype
'numeric))
                  '(**new-parameter**)))
                   (geometric-constraint-parameter
                 (mapcar #'first (getvalue? object 'faces)))

                   (textual-constraint-parameter
                 (append
                  (slotnames (listslots* object 'datatype
'text-valued))
                  '(**new-parameter**)))
                   (constraint-parameter
                 (append
                   (slotnames (listslots* object 'datatype
'numeric))
                   (slotnames (listslots* object 'datatype
'text-valued))
                  '(**new-parameter**))))
                 :abort-p nil
                 :title "Select Parameter")))
     (if (equal parameter '**new-constraint**)
     (setq parameter (create-parameter-interface)))
     (list object parameter)))

(defun create-object-constraint-interface (constraint
role)
   (if (equal role 'modify-object-form)
       (let* ((obj (get-object 'design-object "Choose
Object To Modify"))
          (old-const (first (getfacet obj 'form
'in-constraint))))
     (linkslot constraint 'object old-const)
     (uncache old-const constraint)))
   (do ((slot (mt/singlechoicemenu
            (define-constraint-menu constraint)
            :title constraint
            :position *menu-position*
            :font *menu-font*
            :title-font *title-font*
            :abort-p t)
           (mt/singlechoicemenu
            (define-constraint-menu constraint)
            :title constraint
            :position *menu-position*
            :font *menu-font*
            :title-font *title-font*
            :abort-p t)))
```

```
      ((or (and (equal slot '*done*)
           (slotvaluep constraint 'name))
       (null slot))
       (if (null slot)
       (return nil)
       (progn
         (if (null (slotvaluep constraint 'shape))
          (putvalue constraint 'shape 'composite))
         t)))
    (putvalue constraint
          slot
          (case (slotname slot)
         (parent
          (mt/singlechoicemenu
           (append
            (objectnames (instances* 'design-object
'design))
            (list '*generic-object*))
           :title "Choose Parent Object or <abort> for
none"
           :font *menu-font*
           :title-font *title-font*
           :abort-p t))
         (name
          (let ((obj-name
              (get-item :prompt "Object Name"
                        :default (createobjectname
"design-object"))))
            (if (object? obj-name)
                (progn
                 (get-item :prompt "Object Already Exist
!!!! (return to continue)"
                           :default t)
                 nil)
                 obj-name)))
         (shape
          (mt/singlechoicemenu
           (objectnames (subclasses* 'design-primitives))

           :title  "Choose shape for Object"
           :position *menu-position*
           :font *menu-font*
           :title-font *title-font*))
         (components
          (mt/multichoicemenu
           (objectnames (progeny* 'design-object))
           :abortp t
           :title "Select components for Object"
           :title-font *title-font*))
         (source
          (get-text
           :prompt "Enter origin of constraint"
```

```
           :default (getvalue constraint slot)))
          (time-code
           (get-item
            :prompt "Enter time code from Protocol"))
          (*done*
           (get-item
            :prompt "Please Make Sure the NAME slots are
defined (return to continue)"
            :default t))
          (otherwise
           (getvalue constraint slot))))))


(defun make-status-constraint-interface (constraint role)

  (do ((slot (mt/singlechoicemenu
              (define-constraint-menu constraint)
              :title constraint
              :position *menu-position*
              :font *menu-font*
              :title-font *title-font*
              :abort-p t)
             (mt/singlechoicemenu
              (define-constraint-menu constraint)
              :title constraint
              :position *menu-position*
              :font *menu-font*
              :title-font *title-font*
              :abort-p t)))
       ((or (and (equal slot '*done*)
          (slotvaluep constraint 'feature-affected)
          (slotvaluep constraint 'constraint-affected))
        (null slot))
        (if (null slot) nil t))
    (case slot
      (feature-affected
       (putvalue constraint slot
          (choose-object-parameter-pair
           "Choose object/parameter"
           (getdatatype constraint slot))))
      (constraint-affected
       (let ((feature (getvalue? constraint
'feature-affected)))
     (if feature
         (let ((const (mt/singlechoicemenu
                 (choose-constraint-menu
                  nil
                  (getfacet? (first feature)
                          (second feature)
                          'in-constraint))
                 :title "Choose constraint for status
change"
```

```
                :font *menu-font*
                :title-font *title-font*)))
          (putvalue constraint slot const)
          (putfacet constraint slot 'path (list const
'status)))
        (get-item :prompt "Must choose feature first
(<cr> to continue)"
                :default t)))))))

(defun  make-function-constraint-interface (constraint
role)
  (if (equal role 'general)
      (let ((function-module (function-module-interface
constraint)))
    (putvalue constraint 'function-module
function-module))
      (let* ((object (get-object 'design-object
                         (format nil
                              "Choose Design Object for ~s
Constraint"
                              role)))
        (previous (first (getfacet? object role
'in-constraint))))
    (putvalue constraint 'object object)
    (putfacet constraint 'object 'path (list object role))

    (if previous
        (uncache previous constraint))
    (do ((slot (function-constraint-menu constraint role)

           (function-constraint-menu constraint role)))
        ((or (equal slot '*done*)
          (null slot))
         (if (null slot) nil t))
      (case slot
        (*add*
         (let ((new-slot
             (dolist (slot (reverse (listslots* constraint

                                    'role
                                    'function-parameter)))

               (if (null (slotvaluep constraint slot))
                (return slot)))))
           (putvalue constraint new-slot nil)))
        (*remove*
         (let ((old-slots
             (mt/multichoicemenu
              (listslots constraint 'datatype
'function-module)
              :title (format nil
                       "Choose ~s Slots to Remove" role)
```

```
              :abport-p t)))
            (if old-slots
             (mapcar #'(lambda (slot)
                       (let ((fm (getvalue? constraint
slot)))
                         (if fm
                          (case role
                            (purpose
                             (removevalue fm 'object/purpose
object))
                            (behavior
                             (removevalue fm
'object/behavior object))))
                         (deleteslot constraint slot)))
                     old-slots))))
          (otherwise
           (case (objectname (getdatatype constraint slot))

             (design-object
            (let* ((object (get-object 'design-object
                               (format nil
                                    "Choose Design Object for
~s Constraint"
                                    role))))
              (putvalue constraint 'object object)
              (putfacet constraint 'object 'path (list object
role))))
              (function-module
             (let ((fm (function-module-interface
constraint)))
              (if fm
                  (progn
                 (putvalue constraint slot fm)
                 (case role
                   (purpose (addvalue fm 'object/purpose
                                 object))
                   (behavior (addvalue fm 'object/behavior
                                  object)))))))))))))


(defun function-constraint-menu (constraint role)
  (mt/singlechoicemenu
   (append
    (list (list (format nil
                "~20s ~t~30s ~t~15s"
                'object
                (getfacet? constraint 'object 'path)
                (getvalue? constraint 'object))
          'object))
    (mapcar #'(lambda (slot)
             (list
```

```
                (let* ((fm (find-last-successor
                            (getvalue? constraint slot)))
                       (fm-val
                        (if fm
                         (message* fm 'parse))))
                  (format nil
                       "~20s ~t~20s ~t~40a"
                       slot
                       fm
                       fm-val))
                slot))
     (slotnames (listslots constraint 'datatype
'function-module)))
    (list (list (format nil "*ADD-NEW-~s*" (slotname
role)) '*add*))
    (list (list (format nil "*REMOVE-~s*" (slotname role))
'*remove*))
    '(*done*))
   :title (format nil "~s constraint ~s" role constraint)

   :position *menu-position*
   :font *menu-font*
   :title-font *title-font*
   :abort-p t))


(defun function-module-interface (constraint)
  (let* ((current
      (remove-duplicates
       (mapcar #'(lambda (fm)
                 (find-last-successor fm))
        (progeny* 'function-module 'design))))
     (choice (function-module-menu current
                        '(*add* *change*)))
     (new-fm ()))
    (if choice
    (progn
      (case choice
        (*add*
         (setq new-fm (function-module/create))
         (addgeneralizations new-fm 'structured))
        (*change*
         (let ((old-fm (function-module-menu current)))
           (setq new-fm (function-module/create))
           (addgeneralizations new-fm 'structured)
           (uncache old-fm new-fm))))
      (if new-fm
          (define-function-module new-fm constraint)
          choice)))))

(defun define-function-module (new-fm constraint)
```

```
  (putvalue new-fm 'originating-constraint constraint)
  (do ((slot
    (mt/singlechoicemenu
     (define-constraint-menu new-fm)
     :title new-fm
     :position *menu-position*
     :font *menu-font*
     :title-font *title-font*)
    (mt/singlechoicemenu
     (define-constraint-menu new-fm)
     :title new-fm
     :position *menu-position*
     :font *menu-font*
     :title-font *title-font*)))
      ((equal slot '*done*) t)
    (let ((value
       (case (objectname (getdatatype new-fm slot))
         (design-object (get-object 'design-object))
         (expr (get-text))
         (object (define-function-series new-fm slot)))))

      (putvalue new-fm slot value)))
  new-fm)


(defun define-function-series (fm slot-type)
  (let ((series (updated-series fm slot-type))
    (constraint (getvalue? fm 'originating-constraint)))
    (do ((choice (mt/singlechoicemenu
           (append (textual-constraint-menu series)
                (list '*add* '*delete* '*done*))
           :title "Structure Series"
           :position *menu-position*
           :font *menu-font*
           :title-font *title-font*)
          (mt/singlechoicemenu
           (append (textual-constraint-menu series)
                (list '*add* '*delete* '*done*))
           :title "Structured Series"
           :position *menu-position*
           :font *menu-font*
           :title-font *title-font*)))
    ((equal choice '*done*) t)
      (case choice
    (*add*
     (let ((new-module (function-module-interface
constraint))
           (position (mt/singlechoicemenu
                (append (list '*top*)
                     (textual-constraint-menu series))
                :title "Choose Preceding Operation"
                :position *menu-position*
```

```
                :font *menu-font*
                :title-font *title-font*)))
        (if (equal position '*top*)
            (setq series (append (list new-module)
                                 series))
            (progn
           (ninsert-after new-module position series)
           (putvalue position 'post-condition new-module)
           (putvalue new-module 'pre-condition
position)))))
    (*delete*
     (let ((delete-ops (mt/multichoicemenu
                  (textual-constraint-menu series)
                  :title "Choose  Operation to Remove"
                  :position *menu-position*
                  :font *menu-font*
                  :title-font *title-font*
                  :abortp t)))
       (mapcar #'(lambda (ops)
                 (setq series (delete ops series)))
        delete-ops)))
    (otherwise
     (let* ((new-module (function-module/create)))
       (addgeneralizations new-module 'structured)
       (uncache choice new-module)
       (define-function-module  new-module constraint)
       (setq series (substitute new-module choice
series))))))
    series))


(defun function-module-menu (modules &optional options)
  (mt/singlechoicemenu
   (append
    (mapcar
     #'(lambda (fm)
         (let* ((name (objectname fm))
             (value (message* fm 'parse))
             (series-name
              (setq series
                 (slotname
                  (first
                   (listslots fm 'role 'series)))))
             (series-value
              (if series-name
               (objectnames (getvalue? fm series-name)))))

           (list
            (format nil
                 "~20s ~t~50a ~t~10:s  ~30:a"
                 name
```

```
                value
                series-name
                series-value)
            fm)))
     modules)
    options)
   :title "Current-Function-Modules"
   :centered nil
   :font *menu-font*
   :title-font *title-font*
   :abort-p t))


(defun  make-production-constraint-interface (constraint
role)
  (let* ((object (get-object 'design-object
                     (format nil
                          "Choose Design Object for ~s
Constraint"
                          role)))
     (previous (first (getfacet? object role
'in-constraint))))
    (putvalue constraint 'object object)
    (putfacet constraint 'object 'path (list object role))

    (if previous
    (uncache previous constraint))
    (do ((slot (production-constraint-menu constraint
role)
           (production-constraint-menu constraint role)))

    ((or (equal slot '*done*)
         (null slot))
     (if (null slot) nil t))
      (case (slotname slot)
    (*add*
     (let ((new-slot
         (dolist (slot (reverse (listslots* constraint
                                'role
                                'production-parameter)))
           (if (null (slotvaluep constraint slot))
               (return slot)))))
       (putvalue constraint new-slot nil)))
    (*remove*
     (let ((old-slots
         (mt/multichoicemenu
          (mapcar #'(lambda (slot)
                    (list
                      (getvalue constraint slot)
                      slot))
           (listslots constraint 'role
'production-parameter))
```

```
          :title (format nil
                   "Choose ~s Slots to Remove" role)
          :abport-p t)))
       (if old-slots
           (mapcar #'(lambda (slot)
                       (deleteslot constraint slot))
         old-slots))))
    (otherwise
     (case (objectname (getdatatype constraint slot))
       (text
        (putvalue constraint
               slot
               (get-text
                :prompt (format nil
                          "Enter ~s of Constraint" slot)
                :default (getvalue constraint slot))))
       (expr
        (putvalue constraint
               slot
               (get-item :prompt (format nil
                                   "Enter ~s of Constraint"
slot)
                   :default (getvalue constraint slot))))


       (design-object
        (let* ((object (get-object 'design-object
                         (format nil
                              "Choose Design Object for ~s
Constraint"
                              role))))
          (putvalue constraint 'object object)
          (putfacet constraint 'object 'path (list object
role)))))))))

(defun production-constraint-menu (constraint role)
  (let ((slot-list
     (append
      (listslots constraint 'role 'production-parameter)
      (set-difference
       (listslots* constraint 'role)
       (listslots* constraint 'role
'production-parameter)))))
    (mt/singlechoicemenu
     (append
      (mapcar #'(lambda (slot)
              (list (format nil
                      "~20s ~t~30s"
                      (slotname slot)
                      (getvalue? constraint slot))
                slot))
       (slotnames slot-list))
```

```
      (list (list '*add-new-step* '*add*))
      (list (list '*remove-step* '*remove*))
      '(*done*))
     :title (format nil "~s constraint ~s" role
constraint)
     :position *menu-position*
     :font *menu-font*
     :title-font *title-font*
     :abort-p t)))


(defun make-structured-constraint-interface (constraint
role)
  (let* ((object (get-object 'design-object
                      (format nil
                           "Choose Design Object for ~s
Constraint"
                           role)))
     (previous (getfacet? object role 'in-constraint)))
    (if previous
    (let ((choice
           (mt/singlechoicemenu (append
(textual-constraint-menu previous)
                               '(*new*))
                        :title (format nil "Existing ~s"
role)
                        :position *menu-position*
                        :font *menu-font*
                        :title-font *title-font*)))
      (if (equal choice '*new*)
          (progn
         (putfacet constraint 'object 'path (list object
role))
         (putvalue constraint 'object object))
          (progn
         (uncache choice constraint)
         (removefacet object role 'in-constraint
choice))))
    (progn
      (putfacet constraint 'object 'path (list object
role))
      (putvalue constraint 'object object)))
    (define-structured-constraint constraint)))

(defun textual-constraint-menu (constraints)
  (mapcar #'(lambda (const)
          (list (format nil "~S ==>  ~A"
                      const (message* const 'parse))
            const))
   constraints))
```

```
(defun define-structured-constraint (constraint)
  (do ((slot (mt/singlechoicemenu
          (define-constraint-menu constraint)
          :title constraint
          :position *menu-position*
          :font *menu-font*
          :title-font *title-font*)
         (mt/singlechoicemenu
          (define-constraint-menu constraint)
          :title constraint
          :position *menu-position*
          :font *menu-font*
          :title-font *title-font*)))
      ((equal slot '*done*) t)
    (let ((value (case (objectname (getdatatype constraint
slot))
            (design-object (get-object 'design-object))
            (expr (get-text))
            (object (define-structured-series constraint
slot)))))
      (putvalue constraint slot value))))


(defun define-structured-series (constraint slot)
  (let ((series (updated-series constraint slot)))
    (do ((choice (mt/singlechoicemenu
          (append (textual-constraint-menu series)
               (list '*add* '*delete* '*done*))
          :title "Structure Series"
          :position *menu-position*
          :font *menu-font*
          :title-font *title-font*)
         (mt/singlechoicemenu
          (append (textual-constraint-menu series)
               (list '*add* '*delete* '*done*))
          :title "Structured Series"
          :position *menu-position*
          :font *menu-font*
          :title-font *title-font*)))
    ((equal choice '*done*) t)
      (case choice
    (*add* (let* ((generalizations (generalizations*
constraint))
              (new-constraint
               (case (objectname (first generalizations))

              (derived-constraint
(derived-constraint/create))
              (given-constraint (given-constraint/create))

              (introduced-constraint
(introduced-constraint/create))))
```

```
                (position (mt/singlechoicemenu
                     (append (list '*top*)
                          (textual-constraint-menu series))

                     :title "Choose Preceding Operation"
                     :position *menu-position*
                     :font *menu-font*
                     :title-font *title-font*)))
           (addgeneralizations new-constraint (second
generalizations))
           (addgeneralizations new-constraint (third
generalizations))
           (make-structured-constraint-interface
            new-constraint
            (objectname (second generalizations)))
           (if (equal position '*top*)
               (setq series (append (list new-constraint)

                                    series))
               (progn
                 (ninsert-after new-constraint position
series)
                 (putvalue position 'post-condition
new-constraint)
                 (putvalue new-constraint 'pre-condition
position)))))
     (*delete*
      (let ((delete-ops (mt/multichoicemenu
                    (textual-constraint-menu series)
                    :title "Choose  Operation to Remove"
                    :position *menu-position*
                    :font *menu-font*
                    :title-font *title-font*
                    :abortp t)))
        (mapcar #'(lambda (ops)
                  (setq series (delete ops series)))
         delete-ops)))
     (otherwise
      (let* ((generalizaions (generalizations* constraint))

          (new-constraint
           (case (objectname (first generalizations))
             (derived-constraint
(derived-constraint/create))
             (given-constraint (given-constraint/create))
             (introduced-constraint
(introduced-constraint/create)))))
        (addgeneralizations new-constraint (second
generalizations))
        (uncache choice  new-constraint)
        (make-structured-constraint-interface
new-constraint
```

```
                                    (objectname (second
generalizations)))
       (setq series (substitute new-constraint choice
series)))))
    series))


(defun choose-constraint-form-interface (constraint type)

  (if (equal type 'quit)
      nil
      (let* ((form (mt/singlechoicemenu
                (append (objectnames (subclasses* type
'template))
                     '(**new-constraint**))
                :title "Choose Type"
                :abort-p t)))
    (if (equal form '**new-constraint**)
        (create-new-constraint-interface type)
        form))))


(defun create-new-constraint-interface (type)
  (let* ((name (get-item :prompt "Enter new constraint
name"))
     (num-of-var (+ 1 (get-item :prompt "Enter number of
independent variables"
                    :default 1)))
     (new-const (case type
                ((equality-constraint
                inequality-constraint
                quality-constraint
                conditional-constraint
                other)
                  (let* ((var-list (get-variable-list
num-of-var))
                         (nc
(equational-constraint/create/subclass name var-list
type)))
                (putvalue nc 'equation
                       (get-item :prompt "Enter equation in
lisp form"))
                nc)))))
    (examine-interface new-const)
    new-const))

(defun get-variable-list (num-of-vars)
  (let ((var-list (list (get-item :prompt "Entert
Dependent Variable Name or Symbol"))))
    (if (> num-of-vars 0)
    (do ((i 1 (+ i 1)))
        ((= i num-of-vars))
```

```
      (setq var-list
         (append var-list
              (list
               (get-item
                :prompt
                (format nil
                     "Enter Independent Variable-~s Name
or Symbol" i)))))))
    var-list))

(defun constraint-interface ()
  (do ((constraint-choice (mt/singlechoicemenu
*constraint-menu* :title "CONSTRAINT"
                                  :position *menu-position*
                                  :font *menu-font*
                                  :title-font *title-font*)
                 (mt/singlechoicemenu *constraint-menu*
:title "CONSTRAINT"
                                  :position *menu-position*
                                  :font *menu-font*
                                  :title-font *title-font*)))

      ((equal constraint-choice 'return) t )
    (case constraint-choice
      (make-constraint (make-constraint-interface))
      (delete-constraint (delete-constraint-interface))
      (examine-constraint (examine-constraint-interface))

      (return nil))))

(defun make-constraint-interface ()
  (let* ((const-type (mt/singlechoicemenu '(given
introduced return)
                :title "Choose Constraint Type"
                :font *menu-font*
                :title-font *title-font*)))
    (if (equal const-type 'return)
    nil
    (let* ((constraint (case const-type
                    (given (given-constraint/create))

(introduced(introduced-constraint/create))))
           (new-consts (define-constraint-interface
constraint)))
      (if new-consts
          (if (equal new-consts :fail)
           (progn
             (get-item
              :default
              "looping constraint in dependences
         - aborting decision (push return to continue")
             (deleteobject constraint))
```

```
            new-consts)
           (deleteobject constraint))))))


(defun delete-constraint-interface ()
  (mapcar 'deleteobject (mt/multichoicemenu
                (choose-constraint-menu nil (progeny*
'constraint-source 'design))
                :abort-p t
                :title "Select Constraint(s) To Delete")))


(defun examine-constraint-interface ()
  (examine-interface
   (mt/singlechoicemenu
    (choose-constraint-menu
    nil
    (progeny* 'constraint-source 'design))
    :title "Choose Constraint to Examine"
    :centered nil
    :abort-p t)))

(defun examine-decision-interface ()
  (examine-interface
   (mt/singlechoicemenu
    (choose-decision-menu)
    :abort-p t
    :centered nil
    :title "Select Decision to Examine")))

(defun examine-design-object-interface ()
  (examine-interface (get-object 'design-object)))

(defun examine-interface (object)
  (let ((kb-base (knowledge-base-name)))
    (rkbeval 'impulse
          (sendq fastobjecteditor starteditor nil
              (make-strobetuple :kb kb-base :object
object)))))
```