

**From Symbol to Form: A Framework for Design Evolution**

by

**Sreenivasa Rao Gorti**

Bachelor of Technology, 1988  
Indian Institute of Technology, Bombay, India.

Master of Science in Civil Engineering, 1990  
Johns Hopkins University, Baltimore, Maryland.

**Submitted to the Department of Civil and Environmental Engineering  
in partial fulfillment of the requirements for the degree of**

**Doctor of Philosophy in Computer Aided Engineering**

**at the**

**Massachusetts Institute of Technology**

**February 1995**

© Massachusetts Institute of Technology 1995. All rights reserved.

Author \_\_\_\_\_  
Department of Civil and Environmental Engineering  
November 4, 1994

Certified by \_\_\_\_\_  
D. Sriram  
Principal Research Scientist, Thesis Supervisor

Accepted by \_\_\_\_\_  
Joseph M. Sussman  
Chairman, Departmental Committee on Graduate Students

**ARCHIVES**

MASSACHUSETTS INSTITUTE  
OF TECHNOLOGY

**MAR 07 1995**

LIBRARIES



# From Symbol to Form: A Framework for Design Evolution

by

Sreenivasa Rao Gorti

Submitted to the Department of Civil and Environmental Engineering  
on November 4, 1994, in partial fulfillment of the requirements for the degree of  
Doctor of Philosophy in Computer Aided Engineering

## Abstract

Engineering design involves a strong notion of the geometric structure of artifacts. Existing methods for supporting the geometric aspects of design have had limited impact at the conceptual design stage. This is due to three main reasons: (a) CAD systems have concentrated on the capture and representation of geometric shape, as opposed to providing support for *form conception*; (b) Systems which attempt to provide conceptual design support are based on little explicit relation to function; and (c) CAD systems require a detail of representation which is too restrictive for conceptual design.

This thesis presents an approach to support and explicitly capture the process by which the form of artifacts is *conceived*. It develops a framework to derive a geometric structure of a system from an evolving symbolic description. The distinct elements of the *symbol-form* mapping are (a) deriving spatial relationships between objects as a consequence of the functional relationships; (b) instantiating alternative feasible solutions subject to these relationships; and (c) presenting the evolving descriptions of geometry.

Computational support for each of these elements is provided within CONGEN, a conceptual design agent developed as part of the DICE effort. The key contributions from the research include: (a) Automatic evolution of form from an evolving symbolic representation of design, allowing designers to explore multiple alternatives. The issues addressed are function-form mapping, instantiation of alternatives subject to arbitrary constraints, and geometric representation for conceptual design; (b) A representation scheme for symbolic design knowledge: the design model integrates product and process approaches, allowing the user to control design flow; and (c) Integration of paradigms for conceptual design: A system architecture which integrates representation, problem-solving and visualization support for evolution of design.

Thesis Supervisor: D. Sriram

Title: Principal Research Scientist





---

# Acknowledgments

I extend my sincere gratitude to all those who have contributed, in one way or the other, to make my stay at MIT a truly memorable one:

Firstly, I thank Prof. Sriram, my advisor: for all his support and encouragement through different stages of my research; for his confidence and trust in me, which allowed me to mature both as a person and a researcher; and for all those informal discussions and free lunches at Larry's.

To Prof. Bob Logcher, I express my appreciation of all his advice and input, especially in the early stages of my research.

I will always remember Prof. Jerry Connor, for his constant support and goodwill, for pointing out that "Life is good to us; all our stiffness matrices are symmetrical."

Prof. John Williams helped make IESL a better place to be in, with his infectious enthusiasm, good cheer, and excellent sense of humor.

I express my gratitude to Prof. Mitchell, who took time off from his incredibly busy schedule to offer extremely useful suggestions.

I thank Dr. Seshasayee Murthy; he introduced us to ATeams at a time when we were struggling for ideas on solving the constraint satisfaction problem.

I enjoyed my discussions with Salal Humair; I thank him for all the challenges and arguments, for his ideas and help with the ATeams implementation.

Georgios Margelis was of invaluable assistance with the geometric representation implementation.

I owe a great deal to Nabha Rege, for allowing me to put her through the painfully laborious process of reading drafts of my thesis. She suffered through my frustrations and excitements during my research; for this, I will be eternally grateful.

Patrick Kinnicutt, always short of plans, yet always ready to "do something", has been my good friend through my stay here. So too have Rahul Dighe and Kevin Amaratunga. I thank them for all the good times I have shared with them.

Members of the DICE group have always stood by me through this research: I wish to make special mention of Murali Vemulapati, Feniosky Pena Mora and Albert Wong. Each of them has strongly influenced my research and approach; I cannot do justice to this influence here.

Rory O'Connor is probably the most helpful person I have seen in my entire life. He would also probably be embarrassed if I spent too many lines gushing over this.

I thank Jesus Favela ("Don Jesus"), who helped me through my mid-PhD crises with

---

his philosophical advice.

Joan McCusker deserves special mention, for all her support and concern about the students.

To all my other friends at MIT: Thank you.

Last, but by no means the least, my deepest affection for my parents and sisters. They always believed in me (“We have full confidence in you”), and but for their love and support, I would have never made it this far. I dedicate this thesis to them.

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>12</b>
1.1	Motivation . . . . .	13
1.1.1	Primary Research Hypothesis . . . . .	15
1.2	CONGEN: An Integration of Paradigms for Conceptual Design . . . . .	17
1.3	Organization of the Thesis . . . . .	19
<b>2</b>	<b>Background</b>	<b>22</b>
2.1	Modeling Technologies . . . . .	22
2.1.1	Object-Oriented Database Management Systems (OODBMS) . . . . .	24
2.1.2	Knowledge-Based Expert Systems . . . . .	26
2.1.3	Geometric Modeling . . . . .	27
2.1.4	Constraint Theory . . . . .	31
2.2	DICE . . . . .	32
2.3	Summary . . . . .	34
<b>3</b>	<b>A Model of Integrated Product-Process Representation in Design Synthesis</b>	<b>35</b>
3.1	Introduction . . . . .	35
3.2	Object Model . . . . .	37
3.2.1	Definition of Objects . . . . .	38
3.2.2	Relationships . . . . .	39
3.2.3	Artifact . . . . .	41
3.3	Product and Process Representation . . . . .	44
3.3.1	Model Description . . . . .	45
3.3.2	Process Enaction . . . . .	52

## CONTENTS

---

3.4	Related Research . . . . .	53
3.5	Discussion . . . . .	55
<b>4</b>	<b>Symbol-Form Mapping: Issues and Approach</b>	<b>56</b>
4.1	Introduction . . . . .	56
4.2	Symbol-Form Mapping Framework: Studying the Basis . . . . .	57
4.2.1	Relating Function and Form . . . . .	57
4.2.2	Computability . . . . .	59
4.3	An Approach Based On Localized Function Form Mappings . . . . .	61
4.4	Qualitative Spatial Relationships . . . . .	63
4.4.1	Motivation . . . . .	65
4.4.2	Implementation . . . . .	66
4.5	Summary . . . . .	70
<b>5</b>	<b>Asynchronous Teams of Agents: An Approach to Constraint Satisfaction</b>	<b>71</b>
5.1	Introduction . . . . .	71
5.1.1	Motivation . . . . .	72
5.2	Asynchronous Teams of Autonomous Agents . . . . .	73
5.2.1	A Conceptual ATeam for the Constraint Satisfaction Problem . . . . .	75
5.3	Constraint Evaluation and Improvement . . . . .	76
5.3.1	Constraint Evaluation for Qualitative Relationships . . . . .	76
5.3.2	Improvement . . . . .	79
5.3.3	Modification Operators for Algebraic Constraints . . . . .	81
5.4	Implementation . . . . .	83
5.4.1	Solution Representation and Storage Classes . . . . .	83
5.4.2	Operators . . . . .	83
5.4.3	Overall ATeams Algorithm . . . . .	85
5.5	Summary and Discussion . . . . .	86
<b>6</b>	<b>Geometric Representation for Conceptual Design</b>	<b>87</b>
6.1	Knowledge Representation for Geometry . . . . .	88
6.1.1	Abstraction as a Representation Mechanism . . . . .	88
6.1.2	Multiple Levels of Abstraction in a Unified Framework . . . . .	89
6.1.3	Evolving Geometric Descriptions of Objects . . . . .	89

# CONTENTS

---

6.1.4	Domain Taxonomy . . . . .	91
6.2	Implementation . . . . .	91
6.2.1	Taxonomy . . . . .	92
6.2.2	Evolving Description . . . . .	92
6.2.3	Geometry Interface . . . . .	95
6.3	Summary . . . . .	96
<b>7</b>	<b>CONGEN Implementation</b>	<b>97</b>
7.1	Overall Implementation Framework . . . . .	97
7.1.1	COSMOS . . . . .	98
7.1.2	GNOMES . . . . .	99
7.2	Modeling Process Information . . . . .	100
7.3	Modeling Product Information . . . . .	103
7.3.1	Defining Design Relationships . . . . .	106
7.4	Modeling Geometric Information . . . . .	109
7.5	Constraint Representation and Satisfaction . . . . .	109
7.6	User Interface Components . . . . .	110
7.6.1	Main Console . . . . .	110
7.6.2	Synthesizer . . . . .	111
7.6.3	GRAPHITI . . . . .	112
7.7	Summary . . . . .	113
<b>8</b>	<b>Example and Results</b>	<b>115</b>
8.1	Modeling the Design Products . . . . .	115
8.2	Modeling the Design Process . . . . .	116
8.3	Illustrating the Design Flow . . . . .	117
8.3.1	One Pier Design . . . . .	121
8.3.2	Two Pier Design . . . . .	122
8.3.3	Design Refinement . . . . .	127
8.4	Summary . . . . .	128
<b>9</b>	<b>Conclusions</b>	<b>136</b>
9.1	Summary . . . . .	136
9.2	Contributions . . . . .	138

## CONTENTS

---

9.3 Comparison with Related Research . . . . .	139
9.4 Future Directions . . . . .	141
<b>A Sample CONGEN class declarations</b>	<b>150</b>

---

# List of Figures

1-1	CONGEN: overall architecture, showing interacting modules . . . . .	18
1-2	Organization of the thesis . . . . .	21
2-1	GNOMES system architecture. . . . .	30
2-2	GNOMES primitive classes . . . . .	31
2-3	Cooperative product development . . . . .	33
3-1	CONGEN class abstractions . . . . .	49
4-1	Task map for symbol-form mapping scheme . . . . .	60
4-2	An illustrative classification of relationships . . . . .	62
4-3	Point-interval algebra formulation . . . . .	64
4-4	Reference frames used for the QSRs . . . . .	67
4-5	QSR class hierarchy . . . . .	68
5-1	A schematic diagram of the ATeam for solving the conceptual design problem. . . . .	77
5-2	Evaluation function for point interval relationships . . . . .	78
5-3	Improving numerical constraints . . . . .	82
5-4	Class hierarchy of operators . . . . .	84
6-1	Shape classification . . . . .	93
6-2	Communication between modules . . . . .	95
7-1	CONGEN class hierarchy . . . . .	100
7-2	Context tree showing a particular design alternative. . . . .	101
7-3	Goal editor . . . . .	102
7-4	Specification editor . . . . .	104

## LIST OF FIGURES

---

7-5	Product knowledge: COSMOS console . . . . .	105
7-6	Sample generated code from COSMOS . . . . .	106
7-7	COSMOS Instance editor: can be invoked at any time during the design process . . . . .	107
7-8	Palate for choosing spatial relationships . . . . .	108
7-9	Abuts relationship editor . . . . .	108
7-10	CONGEN main console . . . . .	110
7-11	Synthesizer . . . . .	112
7-12	Ateams user interface . . . . .	113
7-13	GNOMES User interface . . . . .	114
8-1	Design flow for the bridge example . . . . .	118
8-2	Expanding the bridge . . . . .	119
8-3	Functional relationship is created between the slabssystem and piersystem. . . . .	120
8-4	Bridge geometry: Only the slab geometry is known at this point. . . . .	121
8-5	Constraint violation notification . . . . .	122
8-6	Results: Alternative 1 (after 3000 iterations) . . . . .	124
8-7	Results: Alternative 2 (after 3000 iterations) . . . . .	125
8-8	Results: Alternative 3 (after 3000 iterations) . . . . .	126
8-9	Results: Alternative 4 (after 3000 iterations) . . . . .	127
8-10	Results: Alternative 1 (after 6000 iterations) . . . . .	128
8-11	Results: Alternative 2 (after 6000 iterations) . . . . .	129
8-12	Results: Alternative 3 (after 6000 iterations) . . . . .	130
8-13	Results: Alternative 4 (after 6000 iterations) . . . . .	131
8-14	Y shaped emergent alternative . . . . .	132
8-15	Bridge geometry: Piersystem alternative has been accepted. . . . .	133
8-16	Bridge geometry: Each of the piers consists of two columns. . . . .	134
8-17	Bridge geometry: Final design at the end of the conceptual design stage. . . . .	135



---

# List of Tables

4.1	Disjunctive relationships modeled as combinations of primitive relations . .	69
4.2	3D relations modeled using lower level relationships . . . . .	69

---

# Chapter 1

## Introduction

Engineering design involves a strong notion of the geometric structure of artifacts. At a conceptual design stage, the focus of design is on identifying and designing the parts of the system to meet an abstractly specified functionality. These parts must further be interrelated in useful ways to derive a coherent and functional engineering structure. At a detailed design stage, the focus shifts from an overall arrangement of components to detail on individual component shapes. A substantial portion of designer time is thus spent on deriving the geometric structure of artifacts. There has been a great deal of research on supporting the geometric aspects of design. Yet, traditional CAD systems have had limited impact at the conceptual design stage:

- CAD systems are intended for the capture and representation of the geometric shape, as opposed to providing support for *conception*;
- CAD systems require a detail of representation which is too restrictive for conceptual design; and
- The geometric primitives used by these systems have little *relevance* to the domain.

Traditional geometric modeling support has focused on *representation* of form. But how did this form come about? Traditional CAD systems have delegated this responsibility of form conception to the designer. The computer is merely the recorder of information. This is undoubtedly a very important facility. Is it possible to extend this support role of the computer into the conception phase? This thesis advances the belief that this is indeed conceivable.

## 1.1 Motivation

---

This thesis describes an effort to support an evolution of design at a conceptual design stage. More specifically, it presents a framework for the *support* and *explicit capture* of the process by which design artifact form is *conceived*. This research on developing a framework for form conception represents an effort to push computer support upstream in the design process. It explicitly identifies and decouples the various systematic elements involved in form conception. Computational support for each of these elements is provided within the framework of CONGEN, a knowledge-based conceptual design support system implemented as part of this research. The key contributions from the research include:

- *Automatic evolution of form from an evolving symbolic representation of design.* This allows designers to explore multiple alternatives. The issues addressed are function-form mapping, a representation scheme for geometry, and instantiation of alternatives subject to arbitrary constraints;
- *A representation scheme for design knowledge.* A design model which integrates product and process approaches, allowing the user to control design flow; and
- *Integration of paradigms for conceptual design.* A system architecture which integrates representation, problem-solving and visualization support for evolution of design.

Section 1.1 outlines the requirements for a conceptual design shell, motivates this research and formulates the primary research hypothesis. Section 1.2 briefly touches on the requirements for a conceptual design shell. To meet these requirements, an integrated solution architecture is proposed. Section 1.3 outlines the organization of the rest of the thesis.

## 1.1 Motivation

The design of an engineering artifact may be perceived as realizing a physical implementation of a solution described in symbolic terms. Engineering design involves an identification of the needs, formulating the functional requirements, and then through an evolving series of refinements, presenting a design solution. Engineering systems are often assemblies of components, possibly related in many different ways. The designer must make decisions at every step of the process. During the design process, geometry is often an important determining factor for further design decisions. Deriving the geometric structure

## 1.1 Motivation

---

of a high-level system is a very important design task. Consequently, there has been a proliferation of research into modeling the geometry of design artifacts.

Of special interest to this research are three geometry-based approaches: feature-based design [10], parametric design [17] and shape grammars [56]. Feature-based design is an effort to augment geometric detail with non-geometric information related to the artifact. The primitive geometric building blocks are replaced with higher-level modeling elements which are more directly of significance in a given domain. This approach has been embraced widely in mechanical engineering, and several useful applications have been demonstrated. Yet, this approach concentrates on representation of form. It provides no support for conceiving the process by which the form was derived. Feature-based design is often coupled with parametric design. Here, design shape modifications are made in a constraint-directed manner. This constraint-directed approach allows for studying variations in dimensioning and other parameters. This is, however, a post-conceptual design stage, where the components and their topological connectivities are already known. Shape grammars have become very popular in architecture. They have been shown to be a highly successful approach to form automation. Again, this is a purely geometry-directed approach, and useful for a class of graphical object synthesis problems. There is little explicit relation to the design intent, though the design relationships are captured implicitly by rules.

While the representation of form is explicit in these systems, function is implicit in the designers' manipulations with geometric primitives. An extensive survey of existing CAD systems underscores the importance of intelligent CAD systems as a basis for design support for form conception. Traditional approaches to design support have been divided sharply into *geometry-based* and *AI-based* design. Woodbury and Oppenheim [67] present an excellent discussion on the hitherto disparate fields of artificial intelligence and geometric modeling. These fields have dealt with the issues of *complexity* and *geometry* respectively, from vastly different computational viewpoints. AI approaches to support design have dealt with modeling design processes and products. Geometric modeling approaches have concentrated on the geometric aspects of design. Thus treatment of form in design has often been viewed independently of a design process model. Approaches which treat form have evolved independent of a strong representational model. Smithers argues for a tightly integrated combination of AI techniques and geometric modeling techniques [46]. A comparison of AI-based design and geometry-based design [46] clearly supports this argument. Geometry-based approaches primarily suffer from an ontological impoverishment and from

## 1.1 Motivation

---

lack of adequate problem solving support. The expressive power of an AI-based framework can lift the CAD system to the role of an intelligent support system.

The preceding discussion presents a case for a mating of AI and geometry. Indeed some systems already achieve this in a limited sense by providing geometric modeling facilities within the broader context of product modeling. Zamanian [73], Wong and Sriram [68], and others have noted the need for representing evolving geometric forms at different levels of abstraction. Woodbury and Oppenheim present an integration of AI-based techniques and geometric modeling approaches as an architecture for geometric reasoning[67].

Still, support for the geometric aspects of design at the conceptual design stage has been arguably inadequate. An implied notion of geometry and geometric relationships is often extensively used at this stage. Visualization support must provide for explicit representation of these implicit notions. These relationships are often only qualitatively stated; notions of geometry are abstract and subject to change, nevertheless, highly important for a specification of the product at this stage. Traditional CAD systems are overly restrictive in the amount of detail they require of the designer. Further, support for form conception require confrontation of the issue of the relation between function and form, which has proved an elusive quarry. Finally, the computational complexities associated with automated approaches present severe challenges to such an effort. The motivation of this research has been to design and develop a computational framework to solve some of these problems and finesse some of the others. This thesis proposes an overall approach which demonstrates the feasibility of an effort to push computer support upstream in the design process to the conceptual design stage.

### 1.1.1 Primary Research Hypothesis

This research is predicated on the following hypothesis:

**Support for form conception is intimately linked to the symbolic evolution of the design.**

To clarify this claim further: A symbolic description of artifacts is derived during the process of mapping the specifications into a rough design. Such a description of design includes a specification of the components and also the functional relationships which are developed between design artifacts. The term *symbolic description* represents the design of an artifact based on a design process model, guided by design knowledge, resource constraints and designer input at various stages. Such a notion of the design process

## 1.1 Motivation

---

incorporates a strong sense of the design intent inherent in the process. This description could serve as the basis for deriving the geometric shape of the system.

The effort to derive the geometry based on a strong symbolic decomposition support for design is motivated by the following reasons:

- The derived geometric structure would capture the *essential functional intent* of the design;
- The geometry would serve as a basic template for the designer to interactively modify and detail;
- At the conceptual design stage, the geometry would serve as visual feedback for the user in the design process. It may thus further assist in the symbolic aspects of design; and
- The system would allow the user to consider several alternatives at the conceptual design stage, both in terms of functional decomposition, and in terms of physical implementations of these decompositions.

In conceptual design, the focus is on the geometric arrangement of the components to meet the overall system functionality. The components of the design are a fall-out of the symbolic evolution of design. This process also allows the specification of the functional relationships between components. These functional relationships determine the relative sizes, positions and orientations of components in the design. This thesis proposes an explicit representation of design relationships between design objects as a basis for function-form mapping. The scheme for deriving spatial relationships between objects as a consequence of the functional relationships is based on the following observation: In general, it may not be feasible to find function-form mappings in a given domain at overall system levels. Yet, this problem can be solved in part by localizing such mappings to primitive relationships at the component level. Engineering design knowledge about geometry, in fact, to a large degree consists of these domain mappings of one or more commonly used spatial realizations to achieve specialized functions. Such mappings allow the capture of the design intent behind spatial connectivities.

## 1.2 CONGEN: An Integration of Paradigms for Conceptual Design

---

### 1.2 CONGEN: An Integration of Paradigms for Conceptual Design

This section presents the requirements of a design support system, which motivated the architecture of CONGEN.

- *Knowledge Representation.* An integrated engineering design and analysis environment requires a wide variety of knowledge structures and reasoning mechanisms. It is in fact the knowledge intensive nature of the engineering process, and the engineering judgment developed based on this knowledge, that distinguish an expert engineer from the novice. The underlying representation scheme for a design support system should be structured, localized and flexible. The knowledge about the design domain may consist of knowledge about:
  - (a) *The design products and processes.* This is the domain knowledge about the design components used and the process followed to put these components together into a coherent design.
  - (b) *The interrelationships between products and processes.* These interrelationships both increase the complexity of the problem and make it challenging. In many ways, they may be as important as the design entities themselves. These relationships may take various forms: functional relationships, constraints, analogies, spatial relationships, causal patterns, etc.
- *Visualization support.* The design process should support an evolution of form of the designed artifact, and communicate this form effectively to the designer;
- *Problem solving support.* Problem solving support should provide for a variety of reasoning approaches: top-down refinement, bottom-up reasoning, constraint propagation, etc. It should also enable the use of various problem-solving techniques: production systems, constraint management, qualitative and quantitative analysis and case-based reasoning systems have all been demonstrated to be valuable aids for design systems; and
- *Database Support.* The system should provide database support for design. This allows persistence of design data, maintaining alternatives and design histories, versioning, etc.

## 1.2 CONGEN: An Integration of Paradigms for Conceptual Design

In response to these requirements, CONGEN is proposed as a solution architecture for form conception. CONGEN is implemented as an application over a layered architecture in a highly modular, object-oriented manner. The different base modules are independent systems developed as part of the DICE (Distributed and Integrated Environment for Computer aided Engineering) project at the Intelligent Engineering Systems Laboratory, M.I.T (An overview of DICE is provided in section 2.2). These modules provide the basic design functionalities for CONGEN.

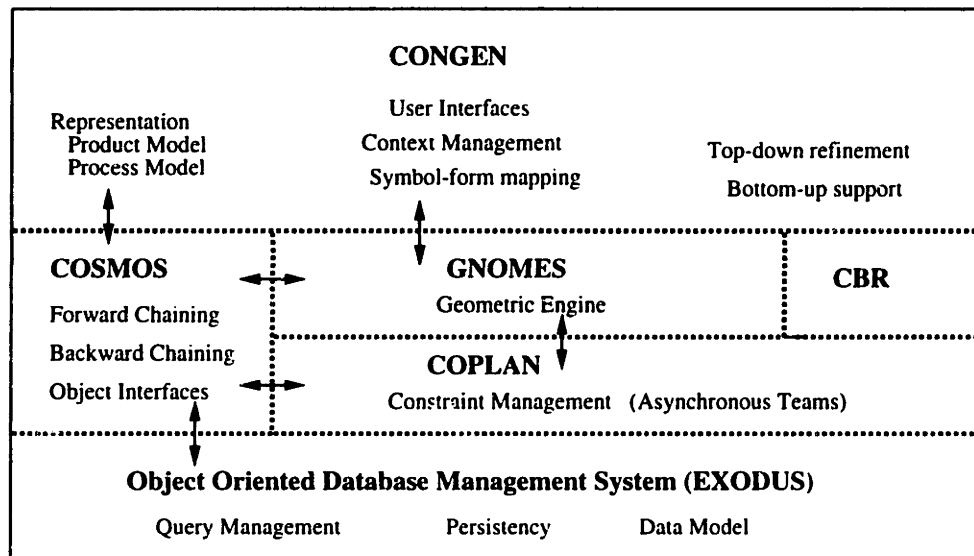


Figure 1-1: CONGEN: overall architecture, showing interacting modules

The overall architecture is shown in Figure 1-1, and is explained in detail below:

- (1) *Object Management* is implemented over an object-oriented database (EXODUS) provided by the University of Wisconsin, Madison [7]. This object management facility provides persistent support, versioning, dynamic schema evolution, and dynamic composition hierarchies;
- (2) *COSMOS* is an object-oriented expert system shell which provides a production system facility with both forward chaining and backward chaining abilities [48]. It also provides a set of generic object interfaces, including class definition tools, a run-time object management facility, and object browsers;



### 1.3 Organization of the Thesis

---

- (3) *GNOMES* is a non-manifold geometric modeler. *GNOMES* provides facilities to represent evolving design geometries in a uniform framework [52];
- (4) *COPLAN* provides a constraint satisfaction framework based on a concept called Asynchronous Teams of Agents; and
- (5) *CONGEN* is the design application shell built as a layered application interacting with these modules. It provides a powerful design knowledge representation scheme, maintains design alternatives and context information, and allows visualization of alternatives. *CONGEN* supports design as a synthesis process, involving an arrangement of pre-defined building blocks to compose the design. While such a synthesis is primarily an aspect of routine design, the flexibility of representation and a fine granularity of the building blocks allow for potentially innovative design solutions to be materialized. The synthesis is based on an integration of three problem-solving approaches: the process-based hierarchical decomposition (or alternately stated, a functional decomposition) of design, the product-oriented bottom-up models, and constraint propagation approaches.

Each of these systems is implemented in a modular, independent manner, using C++ and X-Windows/Motif on a UNIX platform to run on SUN-SPARC workstations. Extensive intercommunication between these modules provides an integrated set of tools for design support in a collaborative framework.

### 1.3 Organization of the Thesis

The organization of the thesis is shown in Figure 1-2:

Chapter 2 presents a brief discussion of background material. It discusses relevant modeling technologies, database management, knowledge-based systems, geometric modeling, and constraint management issues. It also presents an overview of the DICE (Distributed and Integrated Environment for Computer aided Engineering) project at the Intelligent Engineering Systems Laboratory, M.I.T.

Chapter 3 presents a discussion of design knowledge representation, and provides an integrated product-process model which forms the basis for the symbolic design evolution.

Chapter 4 discusses the nature of the symbol-form mapping and describes our approach.

Chapter 5 describes constraint propagation approaches and how they relate to the

### **1.3 Organization of the Thesis**

---

current effort. It identifies requirements for a constraint management scheme and proposes a solution architecture based on Asynchronous Teams of agents.

Chapter 6 describes the need for evolving form descriptions in conceptual design, and proposes a representation scheme which allows multiple levels of abstraction and detail of representation.

Chapter 7 provides details of the CONGEN implementation.

Chapter 8 demonstrates the the feasibility of the approach with a bridge design example.

Chapter 9 presents a summary and discussion of the research presented in the thesis. It compares the research with other related approaches, and identifies the main contributions. It also identifies some promising areas for continued research in this effort.

### 1.3 Organization of the Thesis

---

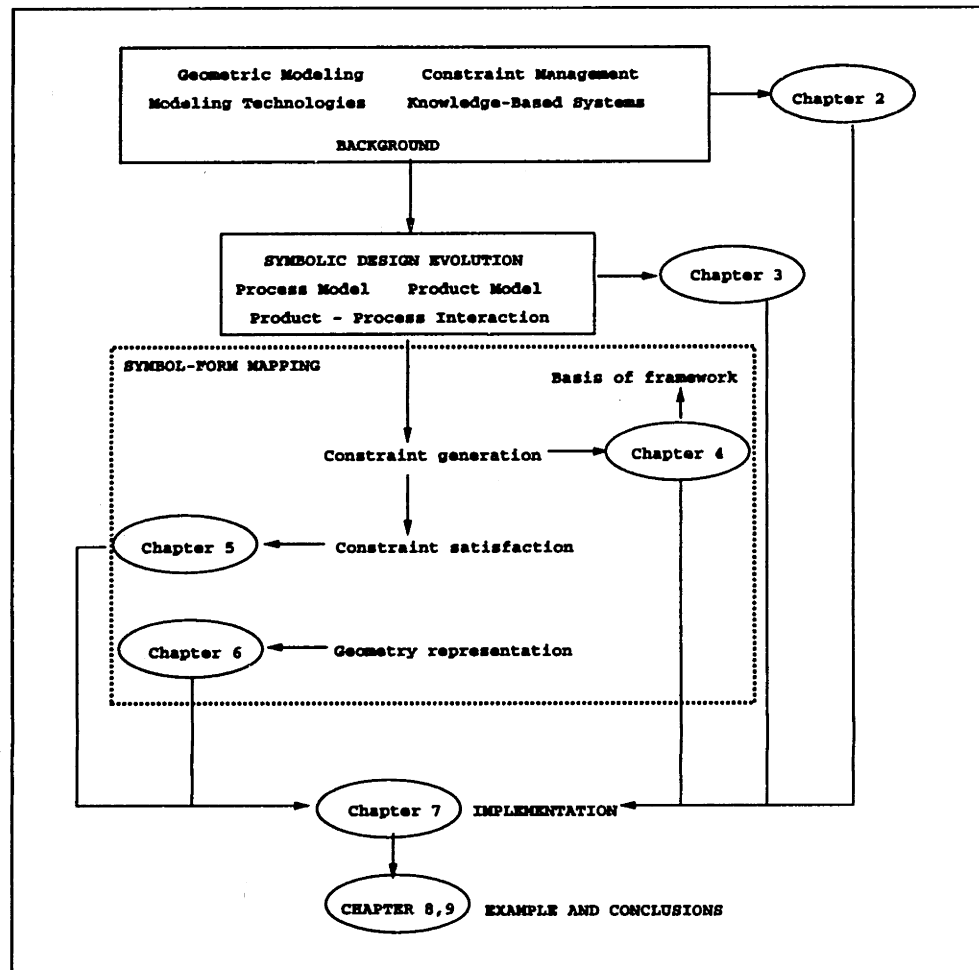


Figure 1-2: Organization of the thesis

---

## Chapter 2

# Background

This chapter presents a broad survey of some of the main themes in this thesis. Section 2.1 provides some background information on relevant modeling technologies, database management, knowledge-based expert systems, geometric modeling issues and constraint representation and management. Section 2.2 presents some contextual background, positioning this research within a larger framework called DICE (Distributed and Integrated Environment for Computer Aided Engineering). Some of the sections are based on the work presented in [21] and [71].

### 2.1 Modeling Technologies

A model is an abstraction of a problem as conceived by a human. In general, we are concerned with two types of models: a conceptual model of the problem domain, and a software model used to actually represent the problem in the computer. The advent of object-oriented modeling technologies has enabled us to now think of these models in a unified fashion. i.e., the conceptual model of the problem domain need no longer be at variance with the representation of the problem in a computer solution. The object-oriented methodology is a philosophy and style of modeling and programming that involves the use of objects and message passing between these objects. As defined by Stefik and Bobrow in [55]:

Objects are entities that combine the properties of procedures and data since they perform computation and save local state.

## 2.1 Modeling Technologies

---

The object-oriented methodology is centered around *data*. This data is represented by *objects*, which contain *attributes* and *methods* (which are procedures that are attached to the objects). Thus a set of objects captures the state of a system at any point, since the state parameters are stored in the attributes. The dynamic *behavior* of the system is governed by the interactions between the behaviors of the individual objects in the system. The behavior of each object is represented by its methods. A central concept in object-oriented methodology is *message-passing*. All interactions between objects, and hence the overall behavior of the system, is modeled by passing messages between objects. Each object may react to a message by invoking one of its methods in appropriate response.

Some of the key characteristics of an object-oriented system are summarized below [41], [71]:

- *Identity*. All objects have a unique identity. This implies that the data space is partitioned into discrete distinguishable entities which can be referenced uniquely;
- *Encapsulation (Information hiding)*. Encapsulation is a mechanism to allow the implementation of an object to be separated from its external interface. This separation permits the actual implementation of the object to be changed without affecting the applications that interact with this object. From an engineering perspective, we observe that encapsulation provides an excellent mechanism to localize knowledge in engineering objects. This approach allows us to cleanly build very general reasoning mechanisms to manipulate this knowledge;
- *Abstraction*. Abstraction is an important knowledge structuring mechanism which allows human beings to efficiently organize information and reduce the complexity of data. In object oriented systems, *classes* provide a means of developing user-defined abstract and logical complex datatypes. An object can then be perceived as a snapshot of a particular abstraction at any point in time. In our language of implementation (C++), an object is referred to as an *instance*;
- *Inheritance*. Inheritance is a higher-level knowledge structuring concept over and above abstraction. Inheritance provides reuse of information, and allows an efficient organization of knowledge. New classes (*subclasses*) can be derived from existing classes (*superclasses* or *base classes*), inheriting the latter's attributes and methods. A subclass *specializes* its superclass. The superclass is thus a *generalization* of a number of its subclasses;

## 2.1 Modeling Technologies

---

- *Polymorphism.* Polymorphism allows objects to present uniform public interfaces while the internal implementation, and thus the response to a message, may be different. Typically, polymorphism is used to retain a uniform message interface across all objects which are derived a base class;
- *Reusability.* Reusability is one of the key benefits gained from encapsulation, abstraction and inheritance; and
- *Extensibility.* New classes and new types with specific semantics can always be created and added to an object-oriented system without needing to modify the existing system.

This thesis demonstrates that the object-oriented methodology is a powerful modeling tool for modeling complex engineering information. The object-oriented methodology is used to represent design artifacts, design processes and design relationships. Inheritance is used to be used to build a layered approach to organizing an engineering knowledge base. In this work, encapsulation is used as a very powerful knowledge structuring mechanism. Encapsulation permits very general reasoning mechanisms to operate on objects, independent of the internals of these objects. Moreover, the software architecture of the overall computer framework follows the object-oriented methodology, demonstrating reusability of code and extensibility of the system.

### 2.1.1 Object-Oriented Database Management Systems (OODBMS)

An object-oriented database management system combines database facilities with an object-oriented data model for the definition and manipulation of data in a persistent store. One of the main advantages of object-oriented database systems is the lack of *impedance mismatch*: the underlying database schema, the programming model used by the application and the database management system, and the data itself, all follow the object-oriented paradigm. This eliminates the need for tedious conversions from one model to the other. OODBMS also permit handling of arbitrarily complex user-defined types in an elegant manner. Some key features of an object-oriented database include [2]:

- *Persistence.* The data resides in persistent storage and can be used across data sessions;
- *Concurrency.* Multiple users can access and use the database simultaneously;

## 2.1 Modeling Technologies

---

- *Transaction management.* This process monitors user interactions and ensures consistency and stability of data;
- *Recovery.* OODBMS have the ability to recover from a crash to some defined stable state;
- *Query language.* A high-level, easy-to-use language for accessing information systematically;
- *Performance.* Efficient data structures and algorithms for retrieving large amounts of persistent data from secondary storage; and
- *Security.* This allows protection of data from unauthorized access.

OODBMS provide a powerful medium for representation, storage and management of complex engineering information. The reader is referred to [2] for a detailed description of the advantages of OODBMS over traditional relational database management systems for capturing engineering information. Some of these are: (1) a more realistic and powerful data model; (2) an environment which allows easier schema development; (3) lack of impedance mismatch between the database manipulation language and the general-purpose programming language in which the rest of the application is written; (4) object identity; and (5) a more unified framework for knowledge representation [2].

OODBMS support for an engineering design system addresses the important issues of persistence and scalability. Persistence provides a mechanism to store not only knowledge, but also information generated during the design process (specifications, constraints, rationale, etc). The information is stored on the persistent database as opposed to the main memory of the computer. This allows designers to pursue a large number of alternatives, and yet manage the massive amounts of data that may typically be generated during the design.

This research uses an object-oriented database management system (EXODUS) developed at the University of Wisconsin. EXODUS is based on a client-server architecture. It provides a programmatic interface to the database for client applications to directly communicate with objects on the database. This programmatic interface is a language called E, which is built on top of C++.

## 2.1 Modeling Technologies

---

### 2.1.2 Knowledge-Based Expert Systems

Knowledge-based systems provide a higher level programming technology as compared to conventional programming environments. A KBES can be defined as [47]:

A computer program which incorporates knowledge and reasoning in solving difficult tasks usually performed by an expert.

A KBES consists of three basic components:

- *Knowledge Base* is a collection of general facts and rules about the problem domain;
- *Inference Mechanism* combines the facts and rules to deduce new facts. Different types of inference mechanisms are available. Typical types are forward chaining, backward chaining, hierarchical refinement, etc. [47]; and
- *Context* is the workspace for the solution constructed by the inference mechanism from the information provided by the user and knowledge base.

The key concept in knowledge-based system technology is the separation of the data (knowledge) from the controlling mechanisms needed to reason about this data (inference mechanisms). This separation of data and control is definitional to the field [67]. It allows a *declarative* mechanism for knowledge representation, where bits of knowledge can be represented independent of the potential ways in which this knowledge may be used. This is very important from a design perspective. It enables designers to just state chunks of knowledge and leave the system to reason with this knowledge and propagate the effects through the set of facts in the context.

In an object-oriented framework, both the knowledge and the inference mechanisms can be treated as objects. In such a framework, methods of objects can be fired conditional upon certain facts being satisfied. These methods may perform various actions. e.g., setting links, triggering daemons which compute other object attributes, distributed constraint checking [71], etc. This permits a seamless integration of the paradigms of procedural programming and heuristic programming.

### An Overview of COSMOS

The framework presented in this thesis (CONGEN) is built over a knowledge-based system building tool called COSMOS(C++ Object-oriented System Made fOr expert System

---



## 2.1 Modeling Technologies

---

development). COSMOS was developed as part of the DICE effort and is a result of several man-years of effort. The system is implemented in C++, over an object-oriented database (EXODUS). It provides the following functionalities:

- *It extends C++ to provide a runtime environment for class evolution.* COSMOS provides a high-level interface to support incremental product model development. It has an extensive set of user interfaces to allow the user to enter C++ classes and rules, instantiate and browse C++ objects, etc. The system generates code corresponding to user-defined classes, and incrementally loads these classes to link with the rest of the program;
- *It provides a persistent object store.* COSMOS provides a persistent repository for user-defined classes and rulebases by communicating with the EXODUS storage manager server. The object management provides a transparent interface to the database. The persistent store also allows programs to share data. This ability is significant for engineering applications.
- *It provides problem solving support.* The inference mechanisms of COSMOS (forward and backward chaining) are developed as independent modules which are tightly integrated;
- *Parts of COSMOS can be integrated into engineering software.* Each of the COSMOS modules is implemented independently in C++, and they can be easily plugged into existing software. The interfaces between these modules are clearly defined. This provides problem solving capabilities to conventional applications;
- *It supports links to external programs;* and
- *It runs on Unix workstations supporting X Window/Motif toolkits.*

A detailed description of COSMOS is given in [48].

### 2.1.3 Geometric Modeling

A geometric model is one of the most common forms of communicating design information. Geometric modeling deals with the representation and manipulation of the geometric properties of a physical object. Typical geometric modeling systems use certain base primitives which are then used to build up more complex models. The classification of these

---

## 2.1 Modeling Technologies

---

systems may be based on the nature of these primitives. Computer-based representation of geometric objects can then be grouped into the following categories [28, 65]:

- *Wireframe modeling.* Wireframe models represent objects by edges and points on the surface of the object. Wireframe models allow the generation of 2D drawings from any view (hence preventing consistency problems). However, they do not contain enough information to capture shape. As a result, some wireframe models of solids cannot be interpreted unambiguously (for examples, see [28]);
- *Surface modeling.* Surface modeling extends the wireframe models by providing mathematical descriptions of the shapes of surfaces of objects. Surface models use surfaces as the basic representation primitive. However, they still do not always give sufficient information for determining all geometric properties and usually offer few integrity checks (e.g., detection of illegal intersection of surfaces);
- *Solid modeling.* Solid models deal with “complete” representation of solid objects. The connectivity between surfaces of objects is either explicitly or implicitly captured. Volumetric information is represented and mass properties such as volume, center of gravity, etc. can be deduced algorithmically. Besides, the integrity of solid models is usually checked such that any model created is guaranteed to be a valid solid (i.e., it is bounded, closed, and has no self intersections). Two-manifold representation schemes are usually used in traditional solid modeling systems.<sup>1</sup> This implies that every point on the surface of the model has a neighborhood which is topologically isomorphic to a two-dimensional open disk. Objects with non-two-manifold conditions (e.g., a cube with dangling edges or faces or an object with interior structures) cannot be modeled in such systems; and
- *Non-manifold modeling.* Non-manifold modeling provides a unified representation that encompasses the capabilities of all the three modeling schemes described above.<sup>2</sup> Non-manifold models remove the restriction of closed, bounded solids made by solid modelers. They are thus able to represent incomplete and evolving geometries within a unified framework. The development of non-manifold modeling schemes has been driven by design applications which need to represent evolving concepts.

---

<sup>1</sup>Cell decomposition models might allow some form of non-two-manifold conditions.

<sup>2</sup>Strictly speaking these systems are “non-two-manifold.” However, in the literature the term “non-manifold” is widely used to connote “non-two-manifold.” This thesis follows the same terminology.

---

## 2.1 Modeling Technologies

---

### GNOMES Geometric Modeler

GNOMES is a non-manifold geometric modeler based on the Selective Geometric Complexes (SGC) model [38]. The SGC model provides a unified representation and manipulation of models of different dimensionality. It has a larger representation domain than other geometric models including non-manifold modelers [4, 39]. It allows representation of non-manifold and inhomogeneous point sets, non-closed point sets (with incomplete boundaries), point sets with missing points or cracks, and disjoint regions. For example, systems with interior boundaries, finite element models, open spaces with incomplete boundaries, and point sets with cracks can be modeled in a single framework. Such flexibility is needed in design applications to model the evolution of design concepts and operations on these concepts. The SGC model supports the development of a number of useful operations (e.g., boolean operations and topological operations) based on only three basic operators (*subdivision, selection, and simplification*) [38].

The basic entities in SGC are known as *cells*. *Cells* are open connected subdivisions of n-dimensional manifolds (i.e., their boundaries are not included in their point sets). Cells generalize the concepts of topological elements (face, edge, vertex) in current modelers and also encompass higher dimensional elements (e.g., volume). Associated with each cell is an *extent*, which represents the geometry of the cell, and *boundaries* which are lower dimensional cells that bound it or are embedded in it (interior boundaries). A boolean “active” attribute is associated with each cell. A *selective geometric complex* (SGC) or model which defines a point set occupied by an object is then represented by a set of these cells. The point set of the complex is defined by the union of the point set of all its “active” cells (i.e., cells whose active attributes are TRUE). Hence, by setting the “active” attribute to either *TRUE* or *FALSE*, one can associate various point sets with a single collection of cells. Non-closed point sets such as those with cracks and incomplete boundaries can be also represented by making corresponding cells *inactive* (setting its “active” attribute to *false*).

Based on this SGC model, GNOMES was implemented as a part of the DICE project to provide geometric support to design applications at various levels. The system architecture for the GNOMES geometric engine is shown in Figure 2-1. GNOMES was originally developed using a commercial object-oriented database management system (OODBMS), ObjectStore<sup>TM</sup>. It has since been ported to EXODUS. The base layer in the GNOMES architecture is the client library of the OODBMS. Above this layer, a number of utility

## 2.1 Modeling Technologies

---

classes (e.g., vector and matrix classes) were implemented. The geometric modeling objects were implemented using the previous two layers. Client applications (e.g., graphical user interfaces or specific CAD programs) are implemented as another layer above the geometric modeling layer. A high-level object hierarchy representing the GNOMES design is shown in Figure 2-2. The class **GNmanager** is the interface for application programs to access the geometric engine functionalities. **GNmodel** is a geometric model which may be an SGC complex (**GNcomplex**) or an assembly of complexes (**GNassembly**). Each **GNcomplex** consists of **GNcells** which represent the topological information, as described above. The **GNcells** have links with **GNextents** which represent the geometry. **GNcells** have boundary and *star* links (to other cells of which they form boundaries). Details of the GNOMES design, its classes and method implementations can be found in [24, 69].

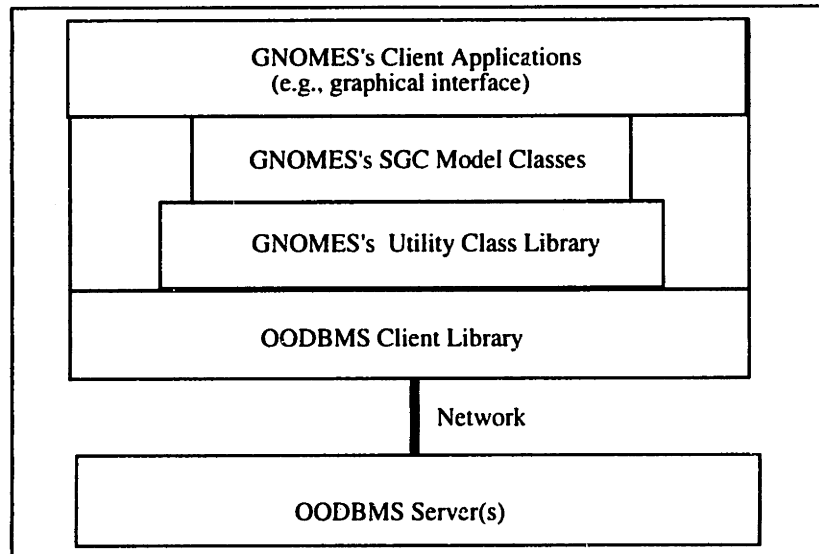


Figure 2-1: GNOMES system architecture.

### Features

Features represent a geometric structuring scheme above a geometric model. Many different definitions have been presented for the concept of features. Some of these are summarized in [45]: “A feature is an entity used in reasoning about the design, engineering, or manufacturing of a product,” “A feature is a region of interest,” “A feature is a collection (set) of faces of a boundary model”.

## 2.1 Modeling Technologies

---

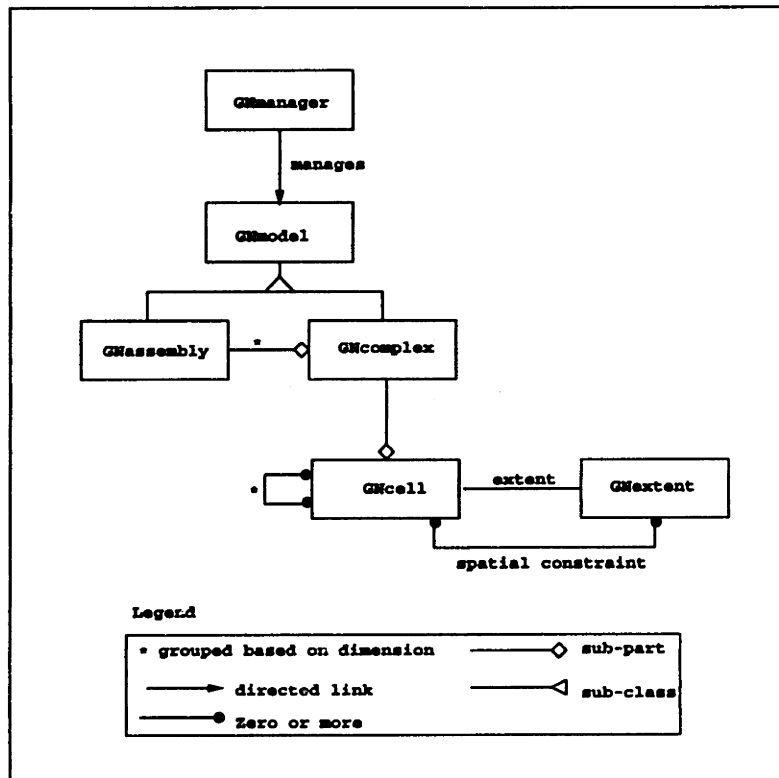


Figure 2-2: GNOMES primitive classes

Features are essentially a representation mechanism within design systems to allow a computer to reason more intelligently about geometry. They are higher level modeling elements which relate more directly to the engineering domain. Extensive work on features and feature recognition has been reported in literature [28, 44]; we limit our description of features to these passing comments.

### 2.1.4 Constraint Theory

Constraints present another powerful declarative mechanism for knowledge representation during design. Constraints arise naturally during design. They may be a statement of resource restrictions; they may represent relationships between design objects; they may represent restrictions arising due to the design context conditions [60]. Constraint declaration during design is rarely a one-time process; constraints are incrementally generated during the evolution of the design.

## 2.2 DICE

---

Constraint satisfaction problems (CSPs) involve the assignment of values to variables which are subject to a set of constraints. The representation of these constraints has traditionally been algebraic or some form of logic statement. Recent approaches have dealt with qualitative constraint formulations based on interval algebra. These typically require a less rigid statement of the constraints. The main elements of a constraint *management* framework are (a) a representation language for constraints; (b) an approach to verify consistency of the constraints; (c) a technique to solve these constraints to *instantiate* a *feasible* solution; (d) handling incrementally added constraints; and (e) studying parametric modifications for values of the variables.

CSPs have been widely studied in the Artificial Intelligence community. Constraint propagation approaches attempt to derive a solution sequence for the determination of a consistent set of assignments to the variables involved in the constraints. Propagation approaches have been used with the algebraic or the qualitative formulations. Albeit theoretically elegant, implementations of these approaches have had computational problems. Constraint management techniques centered around numerical, iterative techniques have been reported by researchers in the design community [25], [43]. These techniques however suffer from considerable computational and robustness problems. Iterative solution techniques are sensitive to the starting point chosen and may accumulate considerable numerical error. These techniques are designed around algebraic specification of equality constraints and are not able to cope with inequalities very well.

This thesis presents a constraint satisfaction architecture which overcomes some of the problems cited above.

## 2.2 DICE

The research presented in this thesis is supported within the framework of the DICE (Distributed and Integrated Environment for Computer aided Engineering) project at the Intelligent Engineering Systems Laboratory, M.I.T. DICE was originally developed to address the following objectives [51]:

- Facilitate effective coordination and communication in various disciplines involved in engineering;
- Capture the process by which individual designers make decisions, that is, what information was used, how it was used, and what did it create;

## 2.2 DICE

- Forecast the impact of design decisions on manufacturing or construction;
- Provide designers interactively with detailed manufacturing process or construction planning; and
- Develop a few design agents for illustrating the approach.

Essentially, DICE can be envisioned as a network of collaborating design agents where the communication and coordination is achieved through a global database and a control mechanism [51]. In this view, an *agent* is a human being working in tandem with an intelligent design support system. Figure 2-3 shows our view of co-operative product develop-

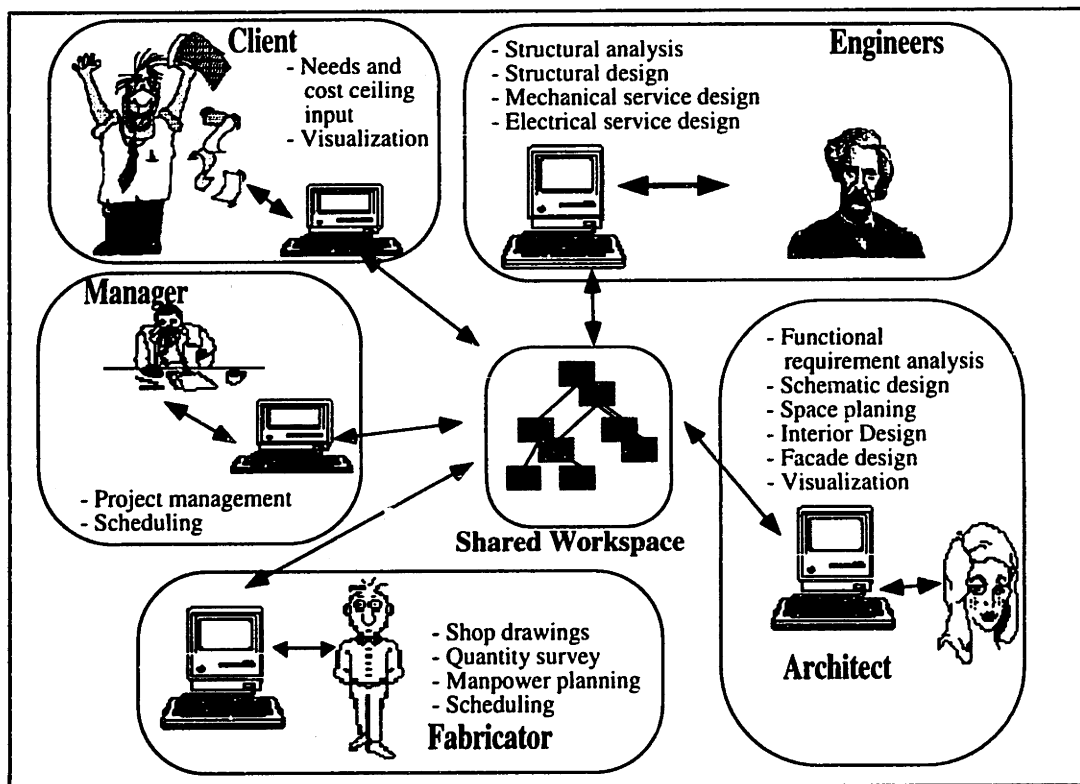


Figure 2-3: Cooperative product development

ment based on this philosophy. The architecture of DICE is based on a Blackboard (global database), several Knowledge Modules and a Control Mechanism. The global database stores the design information, negotiation traces and coordination information. The design information is generated by the various agents taking part in the design process. CON-GEN is a domain-independent knowledge-based design support framework which is used to

### **2.3 Summary**

---

build each of these design agents. Control is achieved by the object-oriented nature of the Blackboard, with the database being populated by "intelligent" objects [51].

CONGEN is implemented as an application over a layered architecture in a highly modular, object-oriented manner. Some of the base modules are independent systems developed as part of the DICE project. These have been tightly integrated with CONGEN and they provide the basic design functionalities for CONGEN. The representation model for CONGEN is based on the DICE philosophy of objects residing on a shared database.

### **2.3 Summary**

In this chapter, some background information on various modeling techniques was presented. The focus of this discussion has been on object-oriented systems coupled with declarative schemes like knowledge-based and constraint-based methodologies. We have also discussed some aspects of geometric modeling relevant to the fundamental goal of this thesis: form conception in engineering design. The following chapters describe in detail how we have used each of these basic methodologies in an integrated environment for design.



---

## Chapter 3

# A Model of Integrated Product-Process Representation in Design Synthesis

This chapter introduces an approach to symbolic evolution in design. There are two main components to the approach: a structured model which defines some primitive constructs and their interactions; and a reasoning approach which operates on this representation.

The chapter is organized as follows. Section 3.1 provides an introduction to representation in engineering design and the underlying philosophy of the model. Section 3.2 presents a formal object model which forms the basis of the representation. It is based on the SHARED object model, defined in [71]. It is extended to design knowledge representation in Section 3.3. The main elements of this representation are: product, process, and the interactions between product and process (Section 3.3.1). Section 3.3.2 presents the different reasoning strategies that the model permits. Related research is presented in Section 3.4. Section 3.5 concludes the chapter with a discussion of the model and how it relates to support for design concept evolution.

### 3.1 Introduction

The need for a model of a problem arises from an effort to have a computer reason about the problem. Early design research concentrated on the problem-solving techniques needed to provide this reasoning support for engineering design. These techniques are rel-

### 3.1 Introduction

---

actively mature now; yet there is a clear realization that techniques operating on a weak representational foundation must necessarily be inadequate to support engineering design. Consequently, recent trends have seen a shift in paradigm to a concentration on representation issues.

What knowledge elements need to be represented? Engineering design involves mapping a specified function into a description of a realizable physical structure – the designed *product* or *artifact* [60]. Unfortunately, this is a non-trivial *process*. Engineering artifacts are typically assemblies of components. These components can be put together in various ways. The effectiveness of the design is determined by what components are chosen, and how well they are put together. Some elements of the knowledge constructs required for a model of engineering design now start to fall into place. We need a model of the design *product*; we need a model of the design *process*. We also need to represent external factors. Design is not merely a technical problem, it is a socio-economic problem. We need to model in some way the effect of *context* conditions: designer *intent*, *decisions*, *objectives*, *constraints*, etc.

The representation problem presented above poses a formidable challenge for researchers. Particular approaches taken by researchers typically limit the scope of the overall problem in some way. Usually, these approaches also adopt particular models of the reasoning process: e.g., rule-based systems, model-based systems, constraint-based systems, qualitative reasoning, etc. Tong and Sriram [60] present an excellent review of the various approaches in design and various representation models. The current research attempts to address the issue of developing a flexible knowledge representation scheme structured to support the development of engineering knowledge bases. The primary focus of such a representation is on modeling the design products (**Artifact**). Nevertheless, modeling the design process is an important aspect. We present an integrated approach to modeling the design enterprise as a whole.

The work presented here reflects two objectives:

- *To support conceptual design*, using a representation which allows considerable latitude in the innovativeness of design alternatives generated. Here, *routine* design refers to a routine combination of primitives and *innovative design* refers to a non-standard, novel combination of primitives from the knowledge base. In our representation, the primitives are organized in a layered hierarchy in the knowledge base. The granularity of description of the components governs the innovativeness of the design. The

### 3.2 Object Model

---

object model allows both coarse-grained system level descriptions and fine-grained component descriptions to co-exist in the knowledge base; and

- *To allow eventually for support for a collaborative model of the design enterprise.* A note on this collaboration process: any model of coordination must account for *communication* and for *conflict*. That agents working on a project have an overlapping vocabulary allows for communication; yet these vocabularies model essentially different views, and this causes conflict. This argues that comprehensive engineering knowledge bases should be developed within the limitations imposed by this model of communication. i.e., the representation scheme must exploit this *commonality* of vocabulary in the form of primitives for product (and process) representation. For engineered artifacts, a fundamental common element is the actual physical realization of the artifact. Wong and Sriram [71] have shown that this fact can be exploited with a model of collaboration based on the notion of shared workspaces. The model presented in this thesis is designed to be consistent with this philosophy.

### 3.2 Object Model

This section presents the object model<sup>1</sup> which forms the basis for the design knowledge representation. The discussion is based on the SHARED object model, defined in [71]. The SHARED model essentially extends the object-oriented methodology in the following manner [71]:

1. *It provides explicit relationship entities with associated semantics and constraints,* instead of just using attribute references to objects. These relationships are associated with relationship classes and can be arranged in inheritance hierarchies;
2. *It associates constraints with objects and relationships.* Constraints are used to maintain the consistency and integrity of a product model; and
3. *It provides a mechanism for handling the concept of "similar objects".* Objects may be similar in that they represent alternative concepts which may be used to satisfy a design intent. An object may also be similar to another object by virtue of being a version of the other. Here, a version corresponds to an incremental refinement of detail in the object.

---

<sup>1</sup>Parts of this section are directly reproduced from [71], with permission from the authors.

## 3.2 Object Model

---

The next section provides a brief overview of the SHARED object model to serve as the basis for further discussion.

### 3.2.1 Definition of Objects

A SHARED object,  $o$ , is defined as a *unique, identifiable entity* in the following form:

**Definition:**

$$o = (\text{uid}, \text{oid}, \mathbf{A}, \mathbf{M}, \mathbf{R}, \mathbf{C}) \quad (3.1)$$

- **uid** is the unique identifier of an object. The set of all unique object identifiers is **UID**;
- **oid** is a non-unique similar object identifier. It is used to refer to one of a set of similar objects which can be used to replace each other in relationships. Typically, this concept is used to model *alternatives* or *versions* of objects. Note that all versions of an object must be instantiations of the same class, whereas alternatives could represent any class. The set of all **oids** is **OID**.
- $\mathbf{A} = \{(t_i, a_i, v_i)\}$ . Each  $a_i$  is called an *attribute* of  $o$  and is represented by a *symbol* which is unique in  $\mathbf{A}$ . Associated with each attribute is its *type*,  $t_i$ . Each  $t_i$  has an associated domain,  $\text{domain}(t_i) = \{v_i\}$ . Then, for  $(t_i, a_i, v_i)$ ,  $v_i$  is called the *value* of  $a_i$  and  $v_i \in (\text{domain}(t_i) \cup \text{nil})$ . If  $v_i = \text{nil}$ ,  $(t_i, a_i, v_i)$  can be written as  $(t_i, a_i)$ .  $\mathbf{A}$  can also have meta-attributes, which have a similar connotation to the attributes. i.e., each  $a_i = \{(t_i, \text{ma}_i, v_i)\}$ , where  $\text{ma}_i$  is a meta-attribute. The meta-attributes have additional information about the attributes: ranges, defaults, etc.
- $\mathbf{M} = \{(m_i, \text{tc}_1, \text{tc}_2, \dots, \text{tc}_n, \text{tc})\}$   
Each element of  $\mathbf{M}$  is a *method signature* which uniquely identifies a method.  $m_i$  is the method name represented by a symbol and  $\text{tc}_i$  is a *type*. The returned type of the method is specified by the last element in the tuple and the other elements define the types of the arguments of the method. Methods define operations on objects and have associated code. A method is defined as (*method signature, code*).

## 3.2 Object Model

---

- $\mathbf{R} = \{\mathbf{rid}\}$ , where  $\mathbf{rid}$  is an identifier for a relationship. Section 3.2.2 discusses these relationships.
- $\mathbf{C} = \{\mathbf{cname}\}$ . Each  $\mathbf{cname}$  is a unique identifier for a constraint,  $\mathbf{c}$  defined as  $(\mathbf{cname}, \mathbf{code})$ . A constraint can be viewed as  $\mathbf{cname}() \rightarrow \mathbf{TRUE|FALSE}$ , that is a function which returns either  $\mathbf{TRUE}$  or  $\mathbf{FALSE}$ . Constraints may be used to restrict ranges of attributes, to define complex expressions on object attributes through rules, etc.

For example, an object is defined as follows: <sup>2</sup>

$(\mathbf{uid1}, \mathbf{oid1}, \{(\mathbf{int}, \mathbf{a}, 10), (\mathbf{String}, \mathbf{b}, \text{"abc"})\}, \{(\mathbf{get\_a}, \mathbf{int}, \mathbf{int})\}, \{\mathbf{r1}, \mathbf{r2}\}, \{(\mathbf{c1}, \mathbf{a} < 20)\})$   
where  $\mathbf{uid1}$  is the unique identifier,  $\mathbf{oid1}$  is a non-unique object identifier,  $\mathbf{int}$  and  $\mathbf{String}$  are primitive data types,  $\mathbf{r1}$  and  $\mathbf{r2}$  are relationship identifiers.  $\mathbf{c1}$  is a constraint on the value of the attribute  $\mathbf{a}$ .

### 3.2.2 Relationships

The SHARED model represents relationships between objects as objects themselves, thus making their semantics explicit. In particular, the relationships defined include **composition**, **functional** and **spatial** relationships, **version-of**, **alternative**, **sub-function**, **satisfied-by** and **requires** [71]. A generic SHARED relationship may be defined as follows:

**Definition:**

$$\mathbf{r} = (\mathbf{rid}, \mathbf{RO}, \mathbf{A}, \mathbf{M}, \mathbf{C}) \quad (3.2)$$

where

- $\mathbf{rid}$  is a unique identifier of the relationship  $\mathbf{r}$ . The set of all unique relationship identifiers is  $\mathbf{RID}$ ;
- $\mathbf{RO} = \{(t, \mathbf{ro}, \mathbf{v})\}$   
Each  $\mathbf{ro} \in \mathbf{RO}$  is called a *role* of a relationship.  $\mathbf{ro}$  is the role name of a role and  $\mathbf{v}$  is the *value* of a role: a wellformedness condition is that  $\mathbf{v} \in \{\mathbf{OID} \cup \mathbf{UID}\}$  or  $\mathbf{v} \subset \{\mathbf{OID} \cup \mathbf{UID}\}$ , and  $\mathbf{v} \in \text{domain}(t)$  where  $\mathbf{OID}$  is the set of all object identifiers and  $\mathbf{UID}$ , the set of all unique object identifiers, and  $t$

---

<sup>2</sup>Object, relationships, method names, types are denoted by boldface fonts.

## 3.2 Object Model

---

is a type. Furthermore, there must be at least two objects partaking in the roles of a relationship. For a relationship among a particular set of objects to be valid, each of the objects must be identified by some role in the relationship. Each of the objects must include the particular relationship in the relationship set  $\mathbf{R}$  of the object's definition;

- $\mathbf{A}$  is a set of attributes of a relationship, defined in a manner similar to  $\mathbf{A}$  of an object;
- $\mathbf{M}$  is a set of methods, defined in a manner similar to  $\mathbf{M}$  of an object. The methods define operations on the roles and attributes of the relationships; and
- $\mathbf{C}$  is a set of constraints on objects associated with the roles of the relationship and its attributes (interaction constraints). It includes constraints on cardinality of roles. It is defined in the same way as  $\mathbf{C}$  of an object.

For example, a relationship could be defined as follows:

$(\mathbf{r1}, \{(\mathbf{System}, \mathbf{composite}, \mathbf{s1}), (\mathbf{Set\_System}, \mathbf{subsystems}, \{\mathbf{s11}, \mathbf{s12}, \mathbf{s13}\}), (\mathbf{String}, \mathbf{description}, \text{"a part of rel"})\}, \{(\mathbf{get\_subsystems}, \mathbf{Set\_System})\}, \{\mathbf{c1}\})$

where  $\mathbf{System}$  is a class,  $\mathbf{Set\_System}$  denotes a set of  $\mathbf{Systems}$ ,  $\mathbf{s11}$ ,  $\mathbf{s12}$ , and  $\mathbf{s13}$  are identifiers of objects which constitute this set and  $\mathbf{c1}$  is a constraint. The method  $\mathbf{get\_subsystems}$  is the access function to return the subsystems, and does not take any arguments.

**Classes** are defined on the objects and relationships defined above, as abstraction mechanisms to make the common properties and semantics explicit. For formal definitions of these mechanisms, the reader is referred to [71].

Moving on to the object and relationship classifications: A SHARED object,  $\mathbf{o}$ , is classified as an *instance* of a class,  $\mathbf{c}$ , if

- $\forall \mathbf{ac} \in \mathbf{c.A}, \exists \mathbf{a}$  in  $\mathbf{o.A}$  such that  $\mathbf{a} = \mathbf{ac}$  or  $\mathbf{a}$  is the same as  $\mathbf{ac}$  except  $\mathbf{a}$  is bounded to a value in  $\mathbf{o}$  while value of  $\mathbf{ac}$  is  $\mathbf{nil}$ ;
- $\forall \mathbf{r} \in \mathbf{o.R}, \exists \mathbf{cr} \in \mathbf{c.R}$  such that  $\mathbf{r} \in \mathbf{domain}(\mathbf{cr})$ ;
- $\forall \mathbf{m}$  in  $\mathbf{c.M}, \exists \mathbf{mc} \in \mathbf{o.M}$  such that  $\mathbf{m} = \mathbf{mc}$ ; and
- $\forall \mathbf{con}$  in  $\mathbf{c.C}, \exists \mathbf{ccon} \in \mathbf{o.C}$  such that  $\mathbf{con} = \mathbf{ccon}$ .

## 3.2 Object Model

---

Furthermore, since an object  $o$  must be in one of the roles of all its relationships, the type (class) of the role in which  $o$  is in must be one of the class in which  $o$  is an instance of.

Generalization and specialization are also defined in terms of the class abstractions. These are relationships between classes which define a partial order on the set of all classes (i.e., they are *reflexive*, *antisymmetric*, and *transitive*). Generalization is also used as an implementation mechanism for sharing code among more specialized classes. That is, a specialized class can inherit properties of a number of more general classes, in a process known as *multiple inheritance*.

We do not go into further specifics of the formal SHARED model, but proceed now to the definition of an **Artifact**, which is the fundamental element representing a design object.

### 3.2.3 Artifact

An **Artifact** is defined to consist of *function*, *form* and *behavior*. An artifact is recursively defined: i.e. an artifact may further consist of components. Moreover, an artifact is based on the SHARED model and hence includes the notions of relationships and constraints. In terms of the SHARED definition, **Artifact** is based on the **System** abstraction [70]:

**Definition:**

An artifact is defined as a tuple:

$$(\mathbf{Artifact}, \mathbf{uid}, \mathbf{oid}, \{\mathbf{Fn}, \mathbf{Fm}, \mathbf{B}\}, \mathbf{M}, \mathbf{R}, \mathbf{C}) \quad (3.3)$$

where  $\mathbf{Fn}$  is a set of functions,  $\mathbf{Fm}$  is the form,  $\mathbf{B}$  is a set of physical behaviors,  $\mathbf{M}$  is a set of methods (in an object-oriented sense),  $\mathbf{R}$  is a set of the various functional, spatial and composition relationships, and  $\mathbf{C}$  is the set of constraints.

In terms of the SHARED object model, the *attribute set* of an **Artifact** comprises of *function*, *form*, and *behavior*. Each of these is further an object at a lower level in the layered knowledge hierarchy:

1. The **function** of a physical system is an abstraction of the required behavior of the system as abstracted by the designers [61]. A function inherently encapsulates some *intent*. Thus, each design view has its own set of functional abstractions of product

## 3.2 Object Model

---

information [70]. **Function** objects are used to capture the functional requirements of a design problem. A **Function** object specifies a requirement which is *satisfied by* an artifact (which may be regarded in this sense as a functional abstraction).

**Definition:**

(**Function**, {(**String**, **description**)}, **M**, **R**, **C**)

**description** = character string representation of the **Function** object, of type **String**. (Note that the tuple notation (*type*, *attribute-name*) is used consistently throughout this thesis to denote such attributes).

**M** = various methods, which may include consistency checks on pre-conditions for the function to be satisfied.

**R** = {**Satisfied\_by**, **Requires**, **Sub\_function**}. These relations refer to the artifacts that satisfy these functions. They are set on instantiation of the objects. Thus these relations define an *object set* associated with this function, akin to that defined in [63]. In case of function refinement, the function shares the **Sub\_function** relationship with other **Function** objects.

**C** = {(**rel**, must have *satisfied\_by* relationship), (**rel1**, if satisfied by a component then there are no sub-functions)}.

2. **Form** represents the physical properties and structure as well as geometric shape and structure of an artifact. The definition of **Geometry** encompasses the **Space** object in the SHARED primitive classes.

**Definition:**

(**Form**, {(**Geometry**, **geom**), (**Material**, **material**), (**Attr**, **Set\_attr**)}, **M**, **R**, **C**)

**geom** is an object attribute whose class is **Geometry** (representing geometric abstractions and spatial relationships).

**material** represents the material and related properties.

**Set\_attr** is a set of attributes describing the non-geometric structure of the object (physical attributes are represented by the **geom** object).

**M** = access methods, dimension independent spatial queries, queries on physical properties, geometric transformation, **display\_selection** operators, an accumulation method which accumulates properties of objects related



## 3.2 Object Model

---

through composition relationships, a top-level method for propagating operations through relationships.

**R** = {**Spatial\_rel**}. These spatial relationships may be qualitatively or quantitatively stated.

**C** = {(**check\_spatial\_consistency**, 3D abstraction should enclose lower level abstractions)}.

3. The **behavior** of a product specifies the response of an object to the context conditions, not how it is used in a given context. The behavior object has an *external* behavior description, as also an *internal* description covering assumptions, pre-conditions and other constraints.

**Definition:**

(**Behavior**, {(**String**, **description**)}, **M**, **R**, **C**)

**description** = external behavior description.

**M** = Methods to combine behaviors of components, to assert causal effects.

**R** = Relationships with the artifacts on which this behavior is binding.

**C** = {(**pre\_cond**, set of preconditions)}.

Each of these objects may be a *composite* object, at various levels of abstraction relevant to the granularity of the system description, e.g., **Behavior** objects may be represented at various levels by qualitative descriptions, by approximate models or by exact equations. Iwasaki and Chandrasekharan [22] describe the representation of behavior following qualitative process theory. This representation includes a description of the pre-conditions, causal effects asserted by the behavior, and the set of objects on which this behavior is valid. An extension to function representation, presented in [63], enables a clear interpretation of function in terms of behavior. The current model and implementation *do not* address such causality, but these representations are essentially consistent with the object model. We believe they can be eventually used to support the model; a rough idea of how this might occur is now sketched out. Representations of the functions as text strings serve as references to the **Function** objects described above. The text strings in this case refer to the **description** of the **Function** objects. In a similar manner, the behavior attributes of an object refer to **Behavior** objects. Relationships between the relevant artifact objects and their function and behavior objects are set on instantiation and retrieval. Retrieval of artifacts to satisfy a set of functions is based on matching of the function description.

### 3.3 Product and Process Representation

---

The behavior verification phase proceeds on identifying the causal process chain expected for successful delivery of the function [63], to match the behavior delivered by the artifact description. Thus we hope to eventually leverage the results from the research in functional reasoning.

The model is designed to support layered development. In addition to artifact descriptions, functions and behaviors are also represented in the appropriate layers of knowledge-use. The next section describes a particular design representation, following the basic model described above.

### 3.3 Product and Process Representation

The search for a representation is inherently related to the view of design that we seek to model. An abundance of definitions may be found in literature (see [60]). For example, design has been defined by Tomiyama and Yoshikawa [59] as the “mapping of a point in the function space onto a point in the attribute space”. Traditional approaches which stress a functional hierarchy deal with a top-down functional decomposition of design, organizing the design product elements as functional primitives. An alternate view has held design to be a bottom-up synthesis of component elements to constitute the whole. In either case, the complex interplay and interconnections among the components serve to provide the overall system function.

Our representation consists of modeling two stages: model description (where the basic constructs to model the problem are defined), and subsequent design process enaction (which is the actual reasoning process to operate on these constructs). The process enaction stage must follow from the model description stage. To a large extent, flexibility afforded by the model description governs the innovation in the process. The approach to model description aims to encapsulate domain concepts through a structured approach to knowledge encoding. Such an approach would structure the design knowledge at various levels, from knowledge of physical principles common to engineering problems through to the very domain-specific heuristic knowledge. The subsequent sections detail our representations for design products, processes and their interactions.

### 3.3 Product and Process Representation

---

#### 3.3.1 Model Description

Much of the design research reported in literature exhibits a clear dichotomy in the representation of design processes and the products that they operate on. Process-based design approaches usually deal with a functional decomposition of design. More generally, the process may be viewed as encompassing not only the functions, but also as a vehicle for expressing design constraints, objectives and specifications. Capturing the design process also involves capture of design rationale and intent within such a framework. The interested reader is referred to related work within the DICE project [35].

Product representation presents several additional well-documented research issues. Object-oriented approaches to design have enabled a natural decomposition and hierarchical structuring of design product knowledge. The dynamically evolving nature of the composition hierarchies, evolving form descriptions, multiple functional and geometric abstractions and multiple levels of constraints have all been identified as crucial issues for product representation [70]. The representation must thus provide for the evolutionary nature of design process enactment. It must also account for the fact that the domain description in the database may be evolving (e.g., during the development of comprehensive knowledge bases).

We hold that design representation should include the function-driven approach of process models and yet retain the expressive power of product models. CONGEN provides support for both process and product hierarchies in engineering design. The following sections discuss each of these models, as well as an integration approach.

#### Product Description

Design products are the ultimate tangibles in any real design process, a mapping from the abstract functional description to real world entities. Such an abstract functional description may be decomposed hierarchically. Decisions are involved at various points, making for the open-ended nature of design. For *function* often implies physical *behavior*, and decisions on sub-systems directly impact the mechanisms chosen for achieving the behavior to realize a given function. The fact that the *form* relationships between the sub-systems affect the behavior of the overall system further complicates the problem. Thus, a robust, flexible representation must necessarily view *function*, *form* and *behavior* from an integrated viewpoint.

### 3.3 Product and Process Representation

---

In general, a desired function of an artifact encapsulates some design intent. Hence, it can be usually specified only as relevant to a context. Behavior is dependent on form relationships. For primitive components, it is possible to specify the essential behavioral properties of an object independent of context. The actual behavior exhibited, of course, depends on the context conditions. Context encompasses *viewpoint* in this case, and contains design-specific information. The model presented in this thesis exploits this separation of context-dependent and context-independent representational elements to support the goal of comprehensive knowledge bases. Thus the context-independent knowledge is comprised of *artifacts* (akin to the concepts represented by *generic components* in [3]). The context information provides the knowledge required to combine these concepts in functionally useful ways.

The observant reader may spot a curious contradiction in the definition of an artifact presented earlier, in particular with regard to the argument presented in the preceding paragraph. We speak of function being view-dependent, and make a case for a separation of context-dependent and context-independent information. This means that the artifact representations in the database should be truly *generic*, as argued by Alberts et al. [3]. Determining function is legitimately a part of the design process. Thus function-form mapping is generally through a set of expected behaviors. In principle, such a representation would be more flexible and supportive of innovation. However, *routine* design incorporates the notion of a direct function-form mapping. In the conceptual model, each artifact description introduced into the knowledge base encapsulates the *originally intended* functions of the artifact. This function representation is a compromise for allowing evolution of the comprehensive knowledge bases. Routine design can thus be facilitated without an extensive re-haul of existing process knowledge. This compromise comes from a realization that a representation tailored to be flexible must yet not be cumbersome from a routine design standpoint.

#### Process Description

Process descriptions concern the knowledge about synthesis of generic artifacts to provide a combined system functionality. This synthesis knowledge further involves relationships between various design entities within the design context and constraints on form attributes of artifacts. The process also captures the decision making and rationale involved in the design process.

### 3.3 Product and Process Representation

---

The following discussion describes a few important primitive constructs relevant to process representation:

- (a) **Goal** constitutes a decision point for a task in the process hierarchy. The goal may be to achieve a function in the functional hierarchy. Function thus serves as a referent into the product world descriptions, which are in the form of abstractions (**Artifact**) in the domain knowledge database. More generally, the goal may introduce a constraint, modify an artifact, introduce a new artifact, or have further sub-goals. Thus a goal may link into the product hierarchy, or alternately, we may pursue the sub-goals, which constitutes a move in the process hierarchy. These moves are further dictated by two considerations: a testing of alternatives specified for the goal, and decisions made by the user to pick from a set of valid alternatives. Following the basic model, a goal may be defined as follows:

**Definition:**

(**Goal**, {(**String**, **name**)}, **M**, **R**, **C**)

**M** = Methods which attempt to achieve this goal: these methods relate to the retrieval and matching procedures discussed in Section 3.3.2. Also included are methods to assert the consequences of asserting a decision for this goal.

**R** = Relationships associate this goal to a parent **Plan** in the process hierarchy and also to the asserted effect conditions of achieving this goal.

**C** = The constraints control the interactions between conflicting goals which drive the design process.

- (b) **Plan** represents a sequential ordering of goals as a task plan. Plans are also associated with artifacts and comprise the link from the product into the process hierarchy (The reverse link, from the process to the product is through **Goal**). Plans may have a set of planning rules associated with them. These rules allow re-ordering of goals and set priorities on the sub-goals.

**Definition:**

(**Plan**, {(**String**, **name**)}, **M**, **R**, **C**)

**M** = Methods which set the schedule for task achievement as an ordering on the goals.

### 3.3 Product and Process Representation

---

**R** = Relationships associate this plan to a parent **Goal** or **Artifact** in the process hierarchy, depending on whether the move being pursued is a refinement of the function or a product decomposition.

- (c) **Specification** represents user input and may be of various forms. The specifications involve some of the important bottom-up elements of the design process: the user may choose to add a new product concept, add relationships between artifacts, specify values of artifact attributes, modify geometry, introduce a constraint, etc.
- (d) **Decision** refers to all user decisions which govern choices for further expansion of a goal. The decisions record the alternatives chosen for a goal within a given design context. Each new **Decision** spurs a new design context (potentially an entirely different design alternative!). The system allows the designer to pursue *multiple* design alternatives simultaneously. **Decision** objects capture the justifications for validity of each alternative as generated by the system. They may also capture the rationale for the choice by the user.

#### Product-Process Interaction

Figure 3-1 describes the object-oriented model used in the CONGEN framework, using standard object notation [41]. Describing the interactions depicted in this figure, a **Context** describes the design context which represents a particular design alternative. A **Context** thus consists of the design tasks (**Goal**) relevant to the current design alternative, the user specifications, the decisions that have been made, and the artifacts which are created as part of the design process. The **Decision** objects refer to choices made for a **Goal**. The **Artifact** is comprised of **Function**, **Form** and **Behavior**. The product-process interactions are shown in the form of links between the goals, plans and artifacts. Artifacts further have sub-components, which are themselves modeled by the class **Artifact**. Similarly, goals can have an ordered listing of subgoals. This is modeled by the class **Plan**.

Formalizing the design process as a mapping from functional specification to a description of one or more alternative artifacts [72], we write

$$D(G, \text{Set}_C) \rightarrow S \quad (3.4)$$

where **D** is the *design process*, **G** is the *design goal* which specifies the desired functionalities,

---

### 3.3 Product and Process Representation

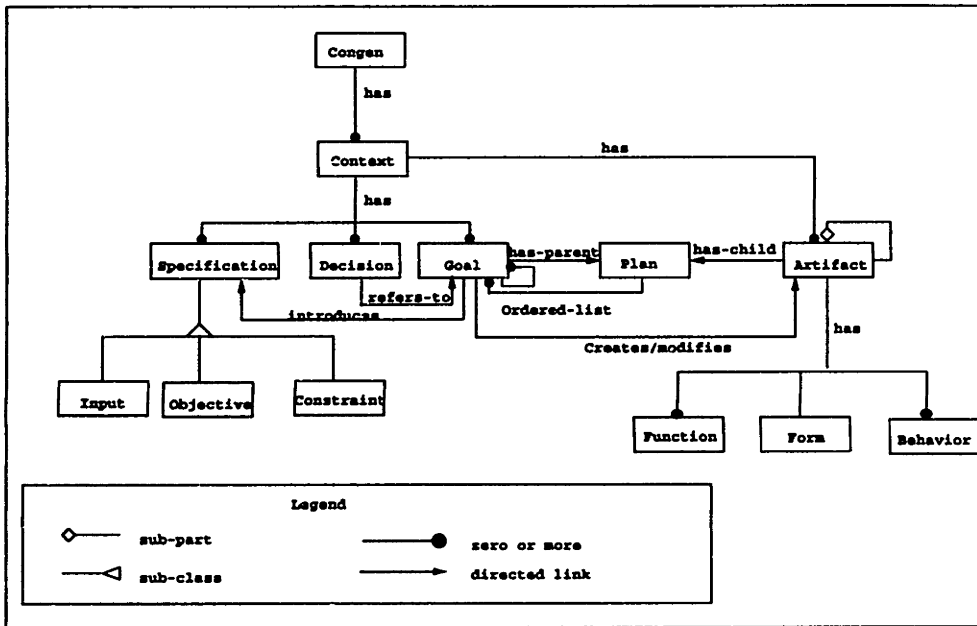


Figure 3-1: CONGEN class abstractions

**Set\_C** is a set of constraints on the design, and **S** is one or more possible *design solutions*.

The design process consists of operations on the different kinds of objects defined above.

Using the notations:

**Set\_Cont** is the set of all contexts,  $C_i$  is the current context of design.

**Set\_G** is the set of all goals in the current context,  $G_i$  is a particular goal.

**Set\_Art** is the set of all Artifacts in the current context (including all components of the high-level system to be designed),  $Art_{i,t}$  is a particular artifact at time state  $t$ .

**Set\_P** is the set of all plans in the current context,  $P_i$  is a particular plan.

**Set\_R** is the set of all the design relationships currently asserted.

**Set\_C** is the set of all the constraints currently asserted.

**Set\_S** is the set of design specifications in the current context, which includes constraints, input and objectives.

**Set\_F** is the set of functions that the system,  $sys$ , needs to satisfy.

### 3.3 Product and Process Representation

---

Formally defining some of the permissible operations:

$$\begin{aligned} \text{expand\_goal}(\mathbf{G}_i, \text{sys}) \rightarrow & (\text{modify}(\mathbf{Art}_i) \mid \text{find\_artifact}(\mathbf{G}_i) \mid \\ & \text{expand\_plan}(\mathbf{P}_i) \mid \text{add\_spec}(\mathbf{Specification})) \end{aligned} \quad (3.5)$$

This operator represents a pursuit of a design task which may lead to a set of potential alternatives. The task may be to modify an artifact  $\mathbf{Art}_i$ , to find an artifact to match the function modeled by  $\mathbf{G}_i$ , or to further refine the function by expanding the goal into a series of tasks modeled by the plan  $\mathbf{P}_i$ .

$$\text{modify}(\mathbf{Art}_{i,t}) \rightarrow ((\mathbf{Art}_{i,t+1}) \ \& \ \text{update\_context}(\mathbf{C}_i)) \quad (3.6)$$

This operator represents changes on an artifact. Such changes may involve changing attribute values, setting components, etc. The effect of these changes in the current context is asserted by firing the operator *update\_context*.<sup>3</sup>

$$\text{find\_artifact}(\mathbf{G}_i) \rightarrow (\text{req}(\mathbf{f}, \text{sys}) \ \& \ \text{retrieve\_match}(\text{req}, \mathbf{C}_i)) \quad (3.7)$$

$$\text{retrieve\_match}(\text{req}, \mathbf{C}_i) \rightarrow (\mathbf{Alt}_i, i = 1 \dots n) \quad (3.8)$$

$$\text{pursue\_alternative}(\mathbf{Alt}) \rightarrow (\text{sys} \ \& \ \mathbf{Art}_k, (\mathbf{C}_k \ \& \ \text{update\_context}(\mathbf{C}_k))) \quad (3.9)$$

where **req** denotes that the function **f** is required by the overall system **sys**. Setting the relation **req** is the task of *model formulation*. It is used as the first phase in retrieving a match to a given functionality.  $\mathbf{Alt}_i$  are the alternative artifact candidates to provide the function in the context conditions  $\mathbf{C}_i$ .  $\mathbf{Art}_k$  is a new **Artifact**,  $\mathbf{C}_k$  is a new design **Context** added to the context set. This new context is necessitated by a user decision to pursue the alternative **Alt**. Updating the new context involves setting the new relationships with the newly created artifact, and updating the constraint set.

$$\text{expand\_plan}(\mathbf{P}_i, \text{Set\_G}_i) \rightarrow (\text{Set\_G}_{i+1} \ \& \ \text{update\_context}(\mathbf{C}_i)) \quad (3.10)$$

where expanding a plan adds introduces a new ordered set of goals into the already existing goal set.

---

<sup>3</sup>The & symbol denotes a conjunction and | symbol denotes a disjunction.



### 3.3 Product and Process Representation

---

$$\text{update\_context}(\mathbf{C}_i) \rightarrow (\text{update\_relationships}(\mathbf{Set\_R}) \mid \text{verify\_constraints}(\mathbf{Set\_C}, \mathbf{Set\_Art})) \quad (3.11)$$

$$\text{update\_relationships}(\mathbf{Set\_R}_t) \rightarrow (\mathbf{Set\_R}_{t+1}) \quad (3.12)$$

$$\text{verify\_constraints}(\mathbf{Set\_C}_t, \mathbf{Set\_Art}_t) \rightarrow (\mathbf{Set\_C}_{t+1}, \mathbf{Set\_Art}_{t+1}) \quad (3.13)$$

Updating a context is an operator which is applied to assert consistency of changes in the current design context. Updating design relationships may include setting component relationships, spatial relationships and various functional relationships between artifacts in the current design context. Any assignment of attributes or asserting relationships may involve propagation of constraints to update values of related attributes. In the above equations, such updates are reflected by the change in the time state of the sets  $\mathbf{Set\_Art}_t$ ,  $\mathbf{Set\_C}_t$ .

Each of the above operations has been implemented in CONGEN. The user may also directly affect the design at any point:

$$\text{add\_spec}(\text{Specification}, \mathbf{Set\_Art}_t \mid \mathbf{Set\_R}_t \mid \mathbf{Set\_C}_t) \rightarrow ((\mathbf{Set\_Art}_{t+1} \mid \mathbf{Set\_R}_{t+1} \mid \mathbf{Set\_C}_{t+1}) \& \text{update\_context}(\mathbf{C}_i)) \quad (3.14)$$

where the subscript  $t$  denotes a time state for each of the sets  $\mathbf{Set\_R}$ ,  $\mathbf{Set\_C}$  and  $\mathbf{Set\_Art}$ . This operator allows the user to directly affect the state of design, and also the further design flow.

The flow of design is from a functional description into a description of the product, including the selection of form for the product. Note that the model neither requires the functional hierarchy to be fully pre-specified, nor the product hierarchy to be pre-specified. i.e., neither entry nor exit points into the respective hierarchies are pre-specified. This implies that the further decomposition of a chosen artifact determines the further *functional or product hierarchy*. Form determination at any stage includes specifying the topological connectivity of the components and their structural relationships. Details of geometry of the components are relegated to the next stage. As mentioned earlier, behavior cannot be

### 3.3 Product and Process Representation

---

examined in isolation of the structural configurations. But the determination of the form at this stage allows us to analyze the behavior. It also enables us to examine the feasibility of the form chosen to satisfy a given function.

Such behavior verification and the representation of function which enables behavior verification has received considerable attention in literature (e.g., [22]). Vescovi et. al [63] argue that causality is an essential element of description of function. They develop a representation formalism for functions which provides a means of expressing causal process chains, as a step to function/behavior verification. These approaches develop on research reported in qualitative physics and process theory which address the problem of model formulation. While this thesis *does not* explicitly address these issues, we still work within the function-form-behavior cycle as part of an iterative design process.

The expected behaviors are used to identify all the possible artifacts which could be used in the given context. The model allows for an external description of artifact behavior, including assumptions and conditions governing this behavior, to be used in matching the artifact to deliver the function requirements at this stage. The internals of how this behavior is to be achieved is a consequence of the component behaviors and their structural relationships. This behavior of the artifact governs further decomposition in the design process. In this case, design as a synthesis process involves top-down refinement of this artifact. Some aspect of this decomposition must address the issue of providing the high-level behavior that the artifact has been advertised to possess. Behavior verification is a function of the components and their structural configurations. The appropriate behavior verification procedures may be built in as knowledge at the respective levels of abstraction. These may include idealized models at the conceptual design level (following the work reported above), or invoke analysis packages at more detailed levels.

#### 3.3.2 Process Enaction

The process enaction stage is the actual process of design. This stage encompasses the inference task of operating on the representation model within a particular context. It also includes well-defined user interactions with a set of inference tools in a design environment. The **Context** represents an explicit mechanism for capturing the user interaction: in terms of decisions, specifications and design rationale. The inferencing approaches are primarily based on retrieval and matching operations, guided by user interactions and decisions.

### 3.4 Related Research

---

#### Retrieval and Matching

The design process primarily follows a top-down functional decomposition approach with the system generating alternative solution paths at various stages of the process. It also provides support for a bottom-up approach. The following discussion illustrates this idea through examples of retrieval and matching:

- *Retrieval through direct function-form mapping.* An aspect of routine design, this is possible when the process hierarchies directly have feasible artifact alternatives encoded. In the context of an evolving comprehensive knowledge base, newly developed artifact concepts can be retrieved through a match on the function. Feasibility is however, conditional on the context. Context information is used by a rule-mechanism and constraint propagation to prune the search.
- *Retrieval through function-behavior mappings.* This can be used in cases where it is possible to formulate a set of behaviors expected by a function (It is outside the scope of this thesis to formally address the issue of model formulation to obtain this set of expected behaviors). Behavior matching is then done by an external symbolic representation of behavior within the artifacts. The assumptions and conditional requirements are consequently introduced by the internal representation of the behavior objects. These are an integral part of the behavior verification mechanism.
- *Case-based reasoning techniques.* Each artifact could be considered to represent a case, and indexing techniques used in case based reasoning may be applied here. Since the artifacts (including abstract descriptions of classes) are stored in an object-oriented database, the designer could conceivably perform generalized database queries on the object base directly. This idea could be extended to automate the case retrieval procedure.

The current implementation is limited to the first of the above retrieval approaches. We have spent considerable effort on the conceptual model, so as not to preclude extension. We hope to explore these extensions eventually.

### 3.4 Related Research

Design as a synthesis procedure from elementary building blocks has been dealt with at considerable length in literature ([37], [31], etc.). Representation studies from an analysis

---

### 3.4 Related Research

---

viewpoint have focussed attention on reasoning from first principles [9], [33].

Most of the current engineering systems are very problem-specific in nature. Falkenheimer and Forbus[11] advocate the more robust approach of attempting to develop a general-purpose *domain theory*. They present *Compositional modeling* as a strategy for organizing and reasoning about models of physical phenomena that address a general class of related problems. The explicit representation of modeling assumptions and behavior of physical systems is of particular interest to engineering design. Bylander and Chandrasekharan [6] present the deficiencies of qualitative simulation from the viewpoints of complexity, causality and representation. They argue for *consolidation* as a more general technique for representing behavior. This paper makes an important conceptual distinction between representation of structure and structural relationships, and behavior and behavioral relationships. However the behavior of a device cannot be examined in isolation to its structural configuration. The causal patterns encode this knowledge. The external behavior description of a device states only *what the device does*, not *how it is achieved*. This is consistent with recent object-oriented design approaches. Causality is thus linked directly to the components of the device. From a design synthesis point of view, these preliminary investigations are of significant importance. Any representation model of design must eventually tie in with representations of causal knowledge.

Design prototypes [15] have been presented as an integrated knowledge structuring scheme for engineering design. A prototype in this context is defined as a conceptual schema for bringing design knowledge appropriate to a *typical* design situation within a single schema. This knowledge includes function, behavior and form within a single *situational* synthesis framework. Implementations based on prototypes have been limited to routine design. Alberts et al. [3] argue that this limitation is due to the fact that prototypes represent specific, situational and experience-based knowledge about design. To extend this approach to innovative design, they propose that *generic components* could serve as the basis for construction of these design prototypes. These generic components represent the bottom-up element of design. They are based on physical theory, and represent combinations of basic components that implement commonly used behaviors.

The generic components are similar to the definition of **Artifact**. They are complementary to the design prototypes in a manner similar to our design processes and products being complementary. The approach presented in this thesis differs in allowing an evolving representation of an artifact. This obviates the need for translations between so-called

### 3.5 Discussion

---

“technology-based” layers. These layers represent levels of abstraction in the design process [3]. The artifacts are recursively defined to consist of further artifacts. We place no pre-defined granularity limits on this recursive decomposition. Lastly, we present our own representation within a unified product-process framework for conceptual design.

### 3.5 Discussion

This chapter has presented a conceptual approach to structuring design knowledge. The design concepts are clearly separated into context-dependent and context-independent parts, representing the top-down and bottom-up knowledge respectively. Artifact knowledge is identified as being essentially context-independent. The design process imposes context assumptions and conditions on the selection process for products to deliver overall system functionality. The granularity of the artifact descriptions governs, in part, the innovativeness of the design. The model formulation and behavior verification phases allow for physical principles to drive the process of retrieval.

The implementation of the model attempts to be consistent with the philosophy outlined in this chapter. It is described in detail in Chapter 7. The basic primitives are all classes in C++. The *form* of each primitive has been implemented as defined. The behavior of the base primitives is implemented as C++ methods. The **Artifact** class builds upon the basic C++ class evolution facilities provided by COSMOS. The *structure* of each user-defined domain concept may be defined directly as attributes. The behavior of each user-defined **Artifact** concept may be described in methods. Rulebases and constraints may also be associated with each user-defined class. This provides a more realistic, high-level mechanism for a declarative specification of behavior. COSMOS allows a seamless integration of procedural and rule-based programming approaches. The rules may thus introduce constraints into the context, set various relationships between artifacts, check the consistency of constraints and fire general procedural code. These declaratively specified rules and constraints provide a partial mechanism to specify behavior. The user may also directly affect the state of the design flow. We have thus chosen to build a semi-automated design environment as opposed to a fully automated design system. The issue of behavior representation, is however, by no means complete. It is outside the scope of this thesis to pursue the matter further than presented here.

---

## Chapter 4

# Symbol-Form Mapping: Issues and Approach

This chapter describes the issues involved in mapping a symbolic description of an artifact into a geometric structure. Section 4.1 defines form conception. Section 4.2 presents the basis for the generalized symbol-form mapping framework. Section 4.3 identifies the distinct elements which define the symbol-form mapping, and discusses the transformation from functional relationships to spatial relationships. Section 4.4 presents a specification framework for spatial relationships based upon qualitative interval algebra.

### 4.1 Introduction

This thesis is largely concerned with developing an approach for conception of design form. The objectives are two-fold:

- (1) To provide support for form conception: to help a designer experiment with alternative geometries for the design; and
- (2) To explicitly capture the functional intent behind the design.

The term *form conception* refers to the very initial phase of design, when some broad functional description of the design is somehow mapped into geometry. This mapping may not be unique. Design is often an underconstrained problem: the constraints may not be adequately specified to limit the feasible space to one solution. Support for form conception involves helping a designer to constrain this space. This can be done by helping the designer

## 4.2 Symbol-Form Mapping Framework: Studying the Basis

---

identify multiple alternative candidates for the design. A designer choice to further develop one or more of the designs thus implicitly constrains the space of all designs. The resulting geometric structure serves several purposes. e.g.,

- (a) It captures the essential *functional intent* of the design;
- (b) It serves as a template for interactive modification and refinement; and
- (c) It provides visual feedback to a designer, and may assist in the further symbolic aspects of design.

Two important questions are left implicit in the above discussion:

- (1) What are the issues which constitute form conception?
- (2) How is the functional description mapped into the geometric alternatives?

The first of these questions is easier to answer. The fundamental issues which constitute form conception are: to decide on the components of a design; to decide on an approximate sizing of the components; and, to determine the relative positioning and orientation of the components.

The second question is more difficult. Many researchers have addressed this issue: approaches have tended to concentrate on the details of the computational problems presented by different aspects of the overall problem. This thesis takes a top-down approach to solving the problem. It explicitly identifies and decouples the elements of form conception. It defines the requirements and presents an approach to solve each aspect. The overall framework is developed by defining the interactions between the various solution components.

The following section presents the basis for the overall framework. It examines the symbol-form mapping from two aspects. The first of these addresses the logical character of the relation between function and form, following an analysis by Mitchell [30]. The second aspect is concerned with the computability of the framework. This issue is addressed in light of a computability theory for design developed by Fitzhorn [12].

## 4.2 Symbol-Form Mapping Framework: Studying the Basis

### 4.2.1 Relating Function and Form

Quoting Mitchell [30],

---

## 4.2 Symbol-Form Mapping Framework: Studying the Basis

---

To claim that “form follows function” in some given physical system is to claim, at the very least, that the system’s significant geometric and material properties can be shown to have some utility; they are not merely irrelevant accidents. A stronger claim that can be made is that they have greater utility than the other obvious possibilities.

Mitchell presents an analysis of the above idea, and the conditions under which it might be valid. The framework presented in this thesis closely reflects this idea. Hence, it would be a useful exercise to subject it now to a similar kind of analysis.

The analysis draws from an observation by Mitchell [30] that designers establish a relation between function and form based on a notion of *functional adequacy*. The conditions necessary for functional adequacy are satisfied by existence of certain physical properties. All viable designs are observed to possess these properties. Mitchell formally states this argument [30]:

- (1) At a certain time  $t$ , a system  $s$  functions adequately in a context of kind  $c$ ;
- (2) System  $s$  performs adequately in a context of kind  $c$  only if certain necessary condition(s)  $n$  is satisfied;
- (3)  $P$  is the class of properties that empirically satisfy condition(s)  $n$  under the specified circumstances; and
- (4) Some one of the present individual items  $p$ , which is included in the class  $P$ , is present in system  $s$  at time  $t$ .

Each of these propositions is now informally discussed within the context of an effort to support form conception. The objective of the design is to satisfy premise 4. In general, choosing  $p$  from the *functional equivalence class* [30]  $P$ , is a task which needs evaluation of the relative merits of the alternatives considered within  $P$ . For the form conception case then, the goal is to satisfy premise 3. i.e., find a set of alternative physical implementations (in terms of positions, orientations and sizes) of the components of the design, so as to satisfy a *set* of design constraints. The task then becomes one of specifying the necessary conditions  $n$ : assuming that the symbolic evolution of the design has tentatively accepted 1 as holding true.

Mitchell states this elegantly [30]:



## 4.2 Symbol-Form Mapping Framework: Studying the Basis

---

Clearly then, any satisfactory account of relations between form and function must be based upon:

- (1) Specification of the system in question's functions - that is, the behaviors that are of value to us;
- (2) Specification of the relevant universe **U** of geometric and material possibilities - the possible alternative forms of the system; and
- (3) Specification of the relevant conditions of functional adequacy - the predicates which apply to **U** to establish a subset **P** of functionally adequate possibilities.

These considerations define the approach presented in this thesis. The model of symbolic evolution presented in the previous chapter derives a functional decomposition of a design in an abstract sense. Also derived are some of *the conditions of functional adequacy*: these may be the various physical relationships between the artifacts which constitute the overall system, or numerical constraints on properties of the artifacts. The final task then remains of applying these conditions to the universe of geometric possibilities to derive a set of geometric alternatives.

The central idea of the function-form mapping framework is as follows:

*It is usually not possible to derive a direct function-form mapping for a system composed of many artifacts. But, it is possible to specify domain-dependent spatial equivalents for functional relationships at a localized level. i.e., In general, it is possible to define spatial mechanisms which are commonly used within the domain as implementations for achieving different functions. It is thus conceivable that a set of spatial relationships between the chosen design components can then be used to instantiate various feasible alternatives.*

Figure 4-1 illustrates the tasks defining the overall framework for form conception. This chapter focuses on representation of design relationships, both functional and physical. It explains an approach which allows derivation of spatial equivalents from the functional relationships. Subsequent chapters focus on deriving the geometric alternatives which satisfy the constraints, and the representation of the geometry.

### 4.2.2 Computability

This section presents an analysis of the computability of the symbol-form framework, based on the ideas presented by Fitzhorn [12]. Fitzhorn describes a computational theory for

---

## 4.2 Symbol-Form Mapping Framework: Studying the Basis

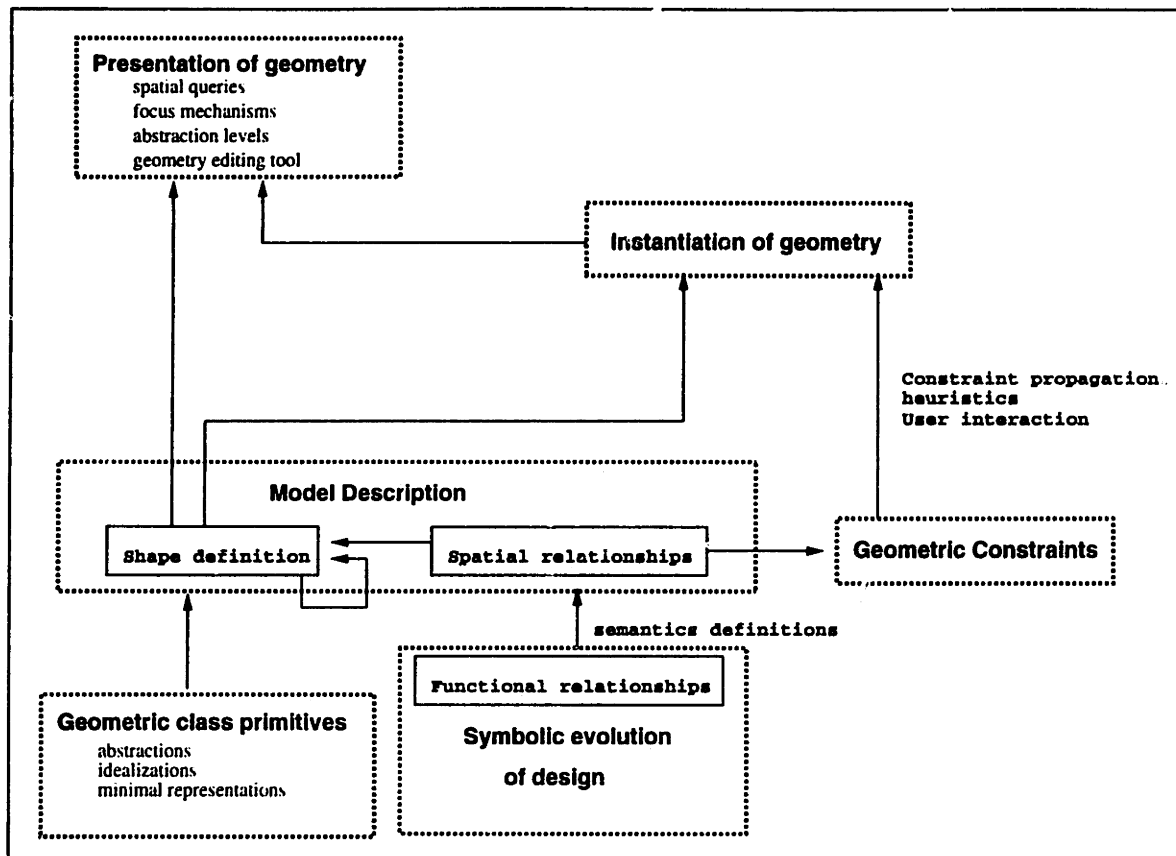


Figure 4-1: Task map for symbol-form mapping scheme

design which argues that design is a computable function. The argument presented therein is about the holistic design process, as a procedure with functional input and spatial output. This section draws on some of these ideas. It presents a restricted form of the argument presented by Fitzhorn. In particular, the analysis is restricted to the spatial aspects of design and the *symbol-form mapping process*.

Fitzhorn's analysis identifies the characteristics of a domain that allow design to be computable within this domain. Quoting his definition of computability [12]:

Let  $\Sigma$  be an alphabet, and let  $f$  map  $\Sigma^*$  to  $\Sigma^*$ . Then  $f$  is grammatically computable if and only if there exists a grammar  $G$  defined on  $\Sigma$  such that for any  $s$  in  $\Sigma^*$ ,  $f(s) \rightarrow s_0$  if  $s$  is derivable from  $s_0$  in the grammar  $G$ .

From a design perspective,  $\Sigma$  represents the language alphabet defining an artifact in the

### 4.3 An Approach Based On Localized Function Form Mappings

---

design,  $\Sigma^*$  represents a design combining many artifacts (\* denotes an arbitrary number of artifacts), and  $f$  is a function mapping an initial state  $s_0$  of the design to another state  $s$ .

Then to show that  $f$  is computable, one must [12]

- (1) choose a sufficiently restricted design domain, all of whose designs are in a language  $L$  defined over an alphabet  $\Sigma$ ;
- (2) show that a grammar  $G$  exists that enumerates  $L$ ; and
- (3) choose a start string  $s_0$  in  $L$  and show that any design  $s$  in  $L$  can be derived from  $s_0$  using  $G$ .

Fitzhorn argues that shape representations of abstract artifacts can be equivalently specified as a finite string in a formal language [12]. He further mentions that while these strings are described in finite lengths, an infinite set of attribute values may exist. For the form conception problem, an artifact alphabet is defined to comprise of the nine parameters which completely define an object in space. These nine parameters are the three parameters defining location of an object in space, the three orientation parameters, and the three parameters which define a box enclosure for the object. A complete description of the design is defined in terms of these nine parameters for each of the design components. It is straightforward to see that there exists a grammar which enumerates the language and that a mapping exists from any design  $s_0$  in  $L$  to any other design  $s$ .

We have not yet expanded on the exact nature of this mapping  $f$  from  $s_0$  to  $s$ .  $f$  defines a state transition function which permits design computation, the translation of an initial shape into an output shape. Following the theory, the design methodology should derive shape from an initial state  $s_0$  subject to the evaluation of sets of constraints or specifications [12]. In this case, the mapping  $f$  is the *constraint satisfaction process*. The form conception methodology presented in Figure 4-1 is thus broadly consistent with this theory.

### 4.3 An Approach Based On Localized Function Form Mappings

Representing design relationships is an important element of the proposed scheme for form conception. Design relationships provide the overall functional and spatial coherence for the design. The previous chapter has hinted at the four broad classes of relationships that we are primarily concerned with: functional relationships, composition, aggregation,

---

### 4.3 An Approach Based On Localized Function Form Mappings

and spatial relationships. It also discussed the basic object model, and how it provides for an *explicit* representation of design relationships.

The classification of functional relationships pertains to the representation of function in a specific domain. Figure 4-2 shows an illustrative classification in a structural engineering context, drawn from [42]. This is *not* a complete classification, but it is used here for illustrative purposes. The figure also shows a tentative, conceptual classification of spatial relationships. The idea here is to demonstrate the kinds of spatial relationships that might conceivably be needed to support such a function-form mapping at the level of relationships. The figure shows three distinct classes of these spatial relationships: **Spat\_constraint** are spatial constraints and restrictions on individual objects; **QuantRel** are more general numerical constraints; and **QSR** represents spatial relationships which are merely specified qualitatively. The qualitative relationships shown here are intuitively convenient; they permit us to conceive of abstract physical interfaces between design objects.

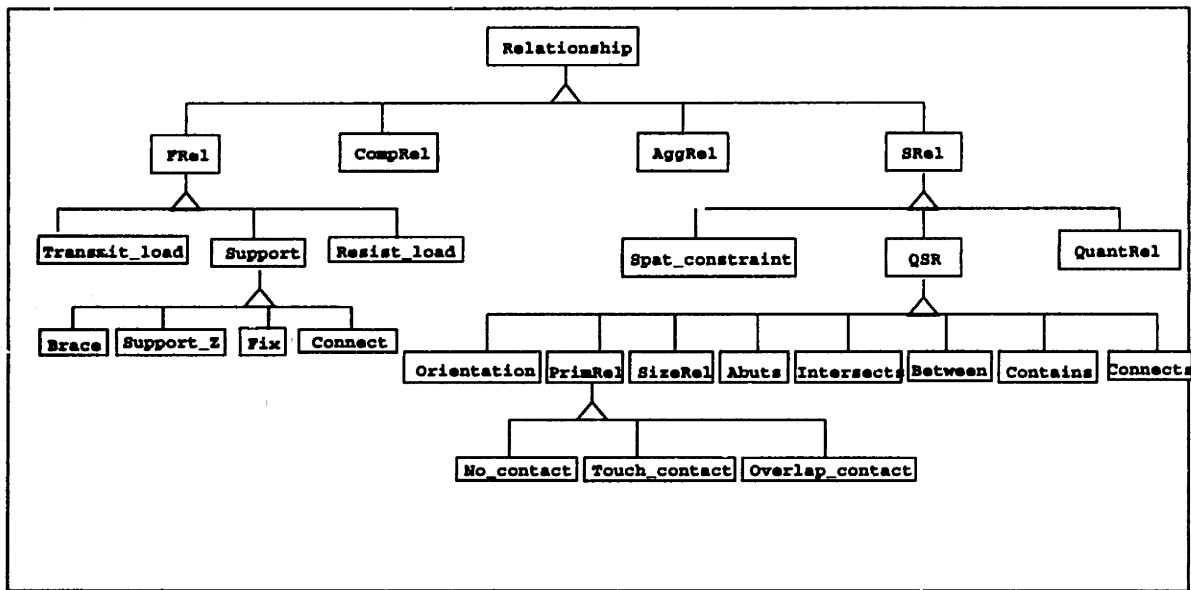


Figure 4-2: An illustrative classification of relationships

The question then remains of obtaining the spatial equivalents of various functional relationships. This is in general defined by the semantics of the functional relationships within the domain. For example, the **Support<sub>z</sub>** relationship is a binary relationship between two

#### 4.4 Qualitative Spatial Relationships

---

objects, implying support in the Z direction, against gravity. Hence we can define:

$$\text{Support}_z := \text{Abuts}_z (\text{Overlap}_x, \text{Overlap}_y, \text{Touch}_z)$$

Similarly, **Connect** is a ternary relationship, where one object connects two others.

$$\text{Connect} : \text{ObjectB connects Object A and Object C.}$$

$$\Rightarrow \text{A} < \text{Overlap} > \text{B}$$

$$\Rightarrow \text{B} < \text{Overlap} > \text{C}$$

When an object has to directly resist a load, that functional constraint usually translates to a spatial constraint on its size.

This thesis reflects a belief that this localized domain knowledge presents an opportunity to build a generalized symbol-form mapping framework. In cases where the system is unable to access such knowledge, the designer may directly specify these spatial equivalents of the corresponding functional constraints. The following section presents a framework for specifying the qualitative relationships at the intuitive, qualitative level used above. This modeling approach for relationships is well-suited to conceptual design, as discussed in the next section.

#### 4.4 Qualitative Spatial Relationships

The previous section mentioned that spatial relationships are in general imposed by the design realities of composition, support, spatial exclusivity, etc. It also introduced the idea of specifying spatial relationships at an intuitive level, allowing them to relate more directly to function. This section presents the specification framework in detail. It discusses the qualitative interval-algebra [32] which forms the basis for the specification framework developed in this thesis.

Qualitative modeling approaches differ from quantitative approaches in allowing abstraction of detail. Usually this abstraction is based on identifying *qualitative changes of state*. This central idea has been used in qualitative physics, temporal algebra and the spatial algebra we present below. The qualitative spatial relationships (QSRs) are based on three qualitatively different spatial states: *no-contact*, *tangency* and *overlap-contact*. Mukerjee presents a point-interval algebraic formulation [32] based on these states.

#### 4.4 Qualitative Spatial Relationships

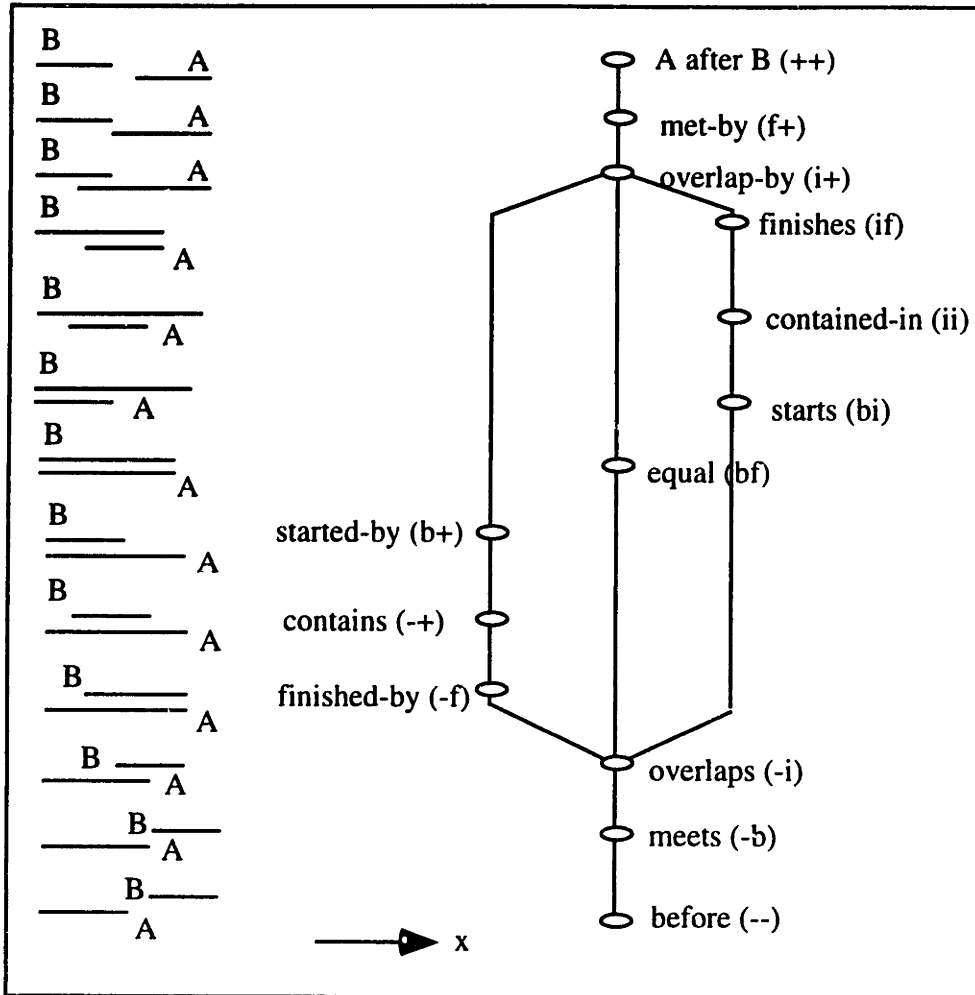


Figure 4-3: Point-interval algebra formulation

A study of the possible relationships between a point  $a$  and an interval  $B$  leads to five positions of interest: 1)  $a$  is *ahead* of  $B$ ; 2)  $a$  is at the *front* of  $B$ ; 3)  $a$  is *inside*  $B$ ; 4)  $a$  is at the *back* of  $B$ ; and 5)  $a$  is *behind*  $B$ . These relations are denoted by  $(+, f, i, b, -)$  respectively. These relationships could be extended to the two-interval case as shown in Figure 4-3. In the figure, each algebraic relationship expresses the relation between the positions of the end points of the two intervals. For example, *A after B* or  $A(++)B$  denotes that both end points of  $A$  are ahead of both end points of  $B$ . Intermediate relationships as  $A$  and  $B$  move toward, and then away, from each other are shown in Figure 4-3. These relationships capture both the relative positions and the relative sizes of the intervals.

## 4.4 Qualitative Spatial Relationships

---

The primitive relations described in Figure 4-3 can also be grouped into higher order relation classes and associated with spatial relation abstractions which are commonly used in engineering (e.g., overlap, inside-of, next-to). For example, a generalized *Overlap* relationship ( $\langle \rangle$ ) can be specified as consisting of *any one* of the following relations (-i, bi, ii, -f, bf, if, -+, b+, i+). Similarly, *No-touch* (i.e., two intervals do not touch each other) can be specified with (++, - -) and *Next-to* (i.e., two intervals only touch at their end points) with (-b, +f). For higher dimensional cases, the problem may be reduced to one dimensional cases by projection onto each of the linearly independent axes. The relationships between the objects are thus represented as vectors whose elements correspond to QSRs in the different axes.

### 4.4.1 Motivation

This research adopts the point-interval algebra approach presented above for several important reasons:

- (1) *QSRs are based on a qualitatively complete, domain-independent categorization of space using interval algebra.* The algebra is independent of the application domain and is related only to space. This retains the flexibility to model diverse spatial design tasks. This model of space is complete in the sense that it can identify all *qualitatively distinct configurations* [32]. The model does not, and cannot, distinguish between other concepts like *near* and *far*, since these concepts are more dependent on *context*;
- (2) *It is possible to verify the consistency of relationships specified in the algebra.* This relates directly to the first point. Since the QSRs are based on a complete algebra, transitive relations can be built algorithmically. Mukerjee gives examples of such transitive inferences [32];
- (3) *QSRs allow abstraction of detail, and are amenable to hierarchical relationships.* Section 4.4.2 describes some of these hierarchical relationships as implemented in this work;
- (4) *QSRs allow spatial relationships between design objects to be specified at a semantic meaningful level.* This flexibility to specify relationships at a level compatible with the intuitive notions in a designer's mind is important from a conceptual design

## 4.4 Qualitative Spatial Relationships

---

perspective. Human language terms like “top,” “between,” “right,” etc. can directly be mapped to QSRs between objects in the sense that they provide an ordering of objects along a chosen direction. This is as opposed to traditional constraint frameworks which require design relationships to be specified as algebraic constraints relating object parameters at a very low level;

- (5) *QSRs can also easily model disjunctions.* For example, touch-contact  $\{-b, +f\}$ , no-contact  $\{-, ++\}$ , front-overlap  $\{f+, i+, if\}$ , etc. are disjunctions formed by clustering lower-level relationships. Disjunctions stipulate that *at least one* of the element relationships in the cluster be satisfied. It is also possible to conceive of disjunctions of the form  $(\text{Pier} < \text{Abuts} > (\text{Bank1 or Bank2 or Bank3}))$ ; and
- (6) *The hierarchical abstractions developed based on QSRs can be more easily related to function,* as discussed earlier on.

### 4.4.2 Implementation

This section explains how higher level relationships between objects can be modeled using primitive relationships. It defines three levels in the relationship hierarchy. The lowest level primitive relationships are available directly from the point interval algebra. The next level consists of relationships modeled as disjunctions. The last level uses combinations of the primitive relationships and disjunctions along each of the axes of a reference frame to model 3D relationships. In the current research, a limited number of 3D relationships have been modeled; yet the idea that this relationship set is easily extensible is to be emphasized.

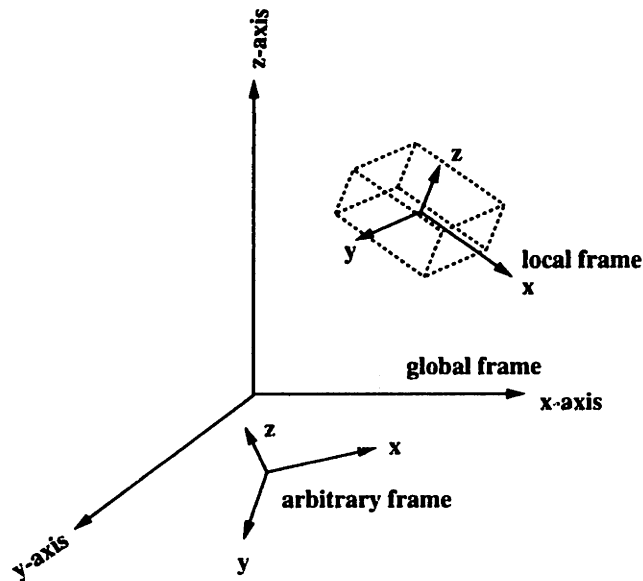
#### Axis System

This section details the approach to defining the axes for the QSRs. Relationships in 3D need to be defined relative to a reference axis system. The implementation allows relationships relative to three basic types of reference frames: a global frame, along an arbitrarily specified axis system, or with reference to another object's local coordinate system (Figure 4-4). Note that in the third case, the reference frame may be changing, depending on the object it belongs to. Interval relationships between objects are specified in terms of the projections on the axes of the reference frame. Thus  $A(++)B$  along  $x$  implies that the projection of A on the  $x$  axis of the chosen frame is ahead of the projection of B.



## 4.4 Qualitative Spatial Relationships

---



The frames used in modeling qualitative spatial relationships

Figure 4-4: Reference frames used for the QSRs

### QSR Hierarchy

Overall, the hierarchical scheme lends itself very naturally to an object-oriented implementation. The class hierarchy for the QSRs is shown in Figure 4-5. The base class **Srel** contains the defining axis for a one-dimensional relationship. It forms the base class for all spatial relationships. It also contains a method to evaluate whether the relationship is satisfied given a configuration for two objects, and a method to suggest improvements to a given configuration of two objects such that the relationship can be satisfied. These two methods reflect a modeling perspective derived from the use of these constraints in randomized evaluation-based search algorithms. The next chapter explains this in detail. Note further that the class **Srel** merely provides a uniform polymorphic interface for the constraint hierarchy. Each of these methods is redefined in the subclasses as appropriate. These refinements are also discussed in the next chapter. The class **QSR** is the base class for relationships formulated in the point-interval algebra. The lowest building blocks of the system are the point-interval classes, derived from **PI\_base**. The interval-interval relationships are derived from **II\_base**. Each of these internally encapsulates two point-

## 4.4 Qualitative Spatial Relationships

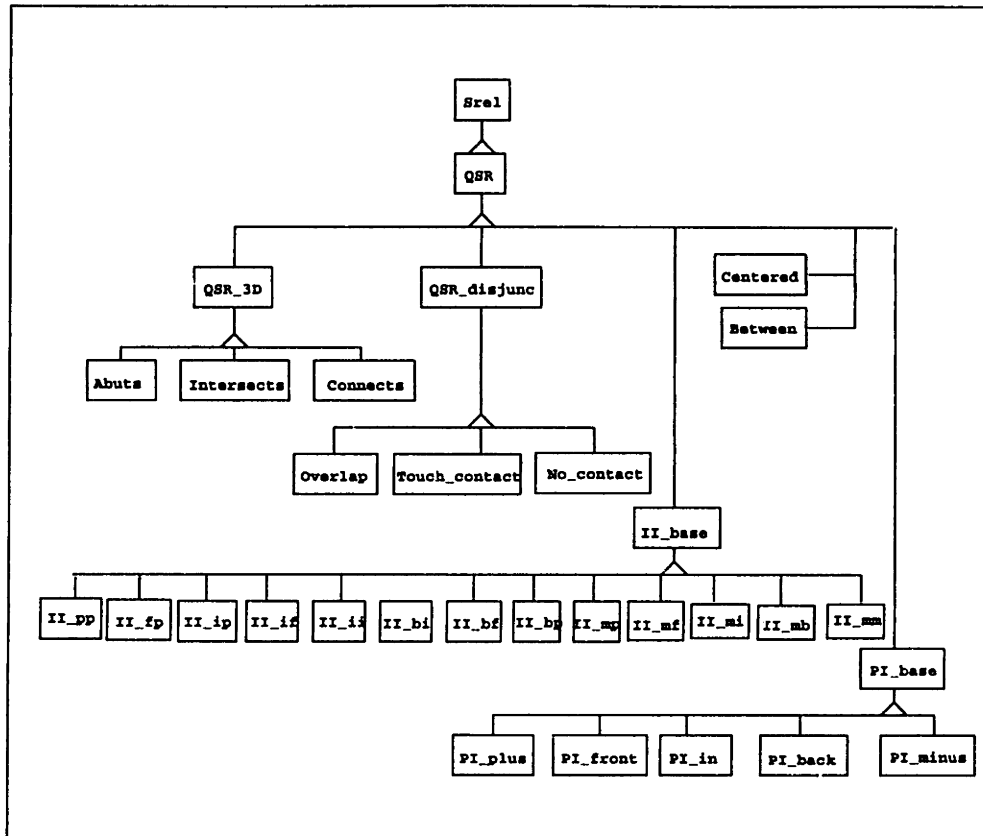


Figure 4-5: QSR class hierarchy

interval relationships. For example, an **II-ii** representing an interval containment contains two **PI\_in** relationships. The class **QSR\_disjunc** is the base class for all disjunctions. It contains an array of objects of type **QSR** which form the components of the disjunction. The class **QSR\_3D** forms the base class for all 3D relationships. It contains an array of three **QSR** objects, one along each direction of a reference frame.

The size and orientation relationships specify relative sizes and orientations of two objects. We have further modeled some disjunctions and 3D relationships, which are by no means a complete set. Table 4.1 and Table 4.2 show these relationships.

### Numerical Constraints

In addition to the QSRs, a specification mechanism for algebraic constraints is also needed. This recognizes the fact that certain important constraints, even at the conceptual

#### 4.4 Qualitative Spatial Relationships

---

Interval Relationship	Components
<i>A overlaps B</i>	A (i+) B A (if) B A (ii) B A (bi) B A (bf) B A (b+) B A (-+) B A (-f) B A (-i) B
<i>A overlaps – front B</i>	A (f+) B A (i+) B A (if) B
<i>A overlaps – back B</i>	A (-f) B A (-i) B A (-b) B
<i>A inside B</i>	A (if) B A (ii) B A (bi) B
<i>A touch – contact B</i>	A (f+) B A (-b) B
<i>A no – contact B</i>	A (++) B A (--) B

Table 4.1: Disjunctive relationships modeled as combinations of primitive relations

Relationship	Along Defining Axis	Along Other Axes
<i>A abuts B</i>	<i>A touch – contact B</i>	<i>A overlap B</i> <i>A overlap B</i>
<i>A intersects B</i>	<i>A overlaps B</i>	<i>A overlap B</i> <i>A overlap B</i>
<i>A contains B</i>	A (ii) B	A (ii) B A (ii) B
<i>C between B and A</i>	<i>C abuts B</i> <i>C abuts A</i>	

Table 4.2: 3D relations modeled using lower level relationships

## 4.5 Summary

---

design stage, may be numerical in nature. The specification framework allows arbitrary algebraic constraints defined on real-valued variables, both discrete and continuous. Any arbitrary algebraic expression for a constraint is parsed into a standard expression tree. An expression is a string of symbols where any string representing an algebraic expression is valid. During the parsing phase, an expression is recursively decomposed into operators, variables and constants.

This specification framework has only been *partially implemented*. Currently, the implementation allows restrictions on parameter variables only (e.g., Less-than and Greater-than constraints). But the overall framework is extensible. A parser to handle arbitrary numerical constraints should eventually be built.

## 4.5 Summary

This chapter has presented the symbol-form mapping framework and the specification of spatial relationships. It presented a basis for the overall framework, discussed the need for qualitative specification, and proposed a solution to the specification problem. Unfortunately, the arbitrary specification poses computational problems in terms of obtaining feasible solutions. The next chapter explains these problems, and also proposes a solution algorithm to tackle them.

---

## Chapter 5

# Asynchronous Teams of Agents: An Approach to Constraint Satisfaction

### 5.1 Introduction

The previous chapter presented a general approach to constraint specification. The resulting generality of the constraints, however, poses severe computational problems. This chapter presents an approach to deal with some of the problems associated with constraint satisfaction. The approach is an effort to reconcile the issues of representational flexibility for constraints and the computational tractability of the resultant framework. This approach for instantiating solutions, based on CMU's Asynchronous Teams of autonomous agents (ATEams), is very general and can handle arbitrary constraint formulations.

The rest of the chapter is organized as follows: Section 5.1.1 presents the motivation for the approach adopted. Section 5.2 presents Asynchronous Teams of autonomous agents, developed by Talukdar et al. [58]. This section also presents a conceptual ATeam for the constraint satisfaction problem. Section 5.3 presents the various agents which form the ATeam for the constraint satisfaction problem. Section 5.4 gives details of the implementation. Section 5.5 presents a discussion of the approach and some of its advantages.

## 5.1 Introduction

---

### 5.1.1 Motivation

The advantages of allowing qualitative specification of relationships have been discussed in the previous chapter. It also discussed the need for handling numerical constraints in the specification framework. The goal of this chapter is to present an approach to solve the *instantiation* problem: i.e., infer a family of feasible solutions. Each solution is a configuration of design objects in space which satisfies the relationships and constraints. Thus each solution is consistent with the essential functional intent of the design.

The overall objective to support conceptual design places some requirements on the constraint satisfaction framework:

- The constraint satisfaction algorithm should be able to handle mixed formulations: qualitative relations and arbitrary numerical constraints;
- Families of feasible solutions should be generated to allow exploration of the design space;
- The framework should be computationally efficient in a dynamic constraint environment, recognizing the fact that constraints are incrementally generated in an evolutionary design process; and
- The framework should be able to report on any subsets of the constraint set that are conflicting and render the problem infeasible.

Several existing constraint satisfaction approaches tackle the instantiation problem. Freidman and Leondes present the mathematical basis of constraint theory and the formulation of the general CSP [13]. Sutherland's SKETCHPAD was a pioneering constraint-driven graphics system [57]. SKETCHPAD solves numerical constraints using propagation combined with relaxation techniques. It deals only with systems of equalities. Steele and Sussman used local propagation for the solution of hierarchical constraint networks [53]. Steele presented a language for the construction of constraint networks[54]. Approaches to deal with constraints which originate in the design community mostly deal with numerical methods, the most significant of these being Variational Geometry [25], [26]. Variational geometry approaches use some form of the Newton-Raphson method to solve systems of non-linear equations. Serrano [43] presents a graph-based constraint management system which allows detection of inconsistencies in a system of equality constraints. Buchanan and Pennington report on a computer algebra-based approach to solving geometric constraint

## 5.2 Asynchronous Teams of Autonomous Agents

---

problems, implemented as CDS [5]. The system is able to handle redundancies and inconsistencies in an elegant fashion. It is also reported to be very slow, and unable to handle incremental definitions of constraints[5]. AI planning techniques were used by Fromont and Sriram [14]. Although very general, the method was computationally very expensive.

Many of the problems in propagation and numerical iterative approaches stem from the fact that the algorithms perform the assignments sequentially. For example, special steps have to be taken when constraint loops are encountered (as in solving simultaneous equations). Further, most of the above-mentioned approaches have concentrated on obtaining a single consistent instantiation. In order to develop multiple feasible solutions, we have experimented with population-based techniques (genetic algorithms and the ATeams approach). Both these techniques are implicitly parallel in nature: they operate on populations of solutions generated randomly. These techniques avoid the computational problems associated with sequential algorithms reported above. Both these techniques are evaluation-driven and independent of the nature of the constraint space. Thus, they offer the robustness across a wide range of constraints, allowing representational flexibility.

The constraint satisfaction approach presented here addresses most of the problems above. The specification framework handles qualitative constraints, disjunctions of constraints, and arbitrary numerical constraint expressions. The instantiation generates families of feasible solutions. This thesis does not completely address the issue of constraint consistency; Section 5.5 discusses the problems associated with this issue.

## 5.2 Asynchronous Teams of Autonomous Agents

ATEams are a relatively new organization architecture of problem-solving agents for solving computationally complex problems. ATeams were developed by Talukdar et al. [58]. As defined by Murthy in his PhD thesis [34]:

An ATeam can be described as an organization of autonomous problem-solving agents, operating asynchronously and cyclically on shared memories.

The essential elements of an ATeam are:

- *A shared memory of candidate solutions to a problem;*
- *A set of operators or agents which operate on these solutions.* The operators may be:

## 5.2 Asynchronous Teams of Autonomous Agents

---

- *Improvement operators.* These suggest local improvements to a randomly picked solution and return a new solution to the store. Note that the original solution is also retained without change;
- *Evaluation operators.* They evaluate a design solution with respect to some objective; and
- *Destruction operators.* They keep the size of the store in check and ensure that the efforts of the improvement operators are not wasted on “relatively poor” solutions.

Note that there is no control or flow to the overall solution scheme. A random store of design solutions is initialized. Each solution is evaluated according to the criteria for a design and the evaluation is tagged along with the design. In an implementation on a serial computer, operators are randomly picked and they are fired on random designs. The operators may be any known solution techniques to the problem being considered. The solution architecture affords the flexibility to allow integration of very well-tested techniques for subsets of the problem, with purely heuristic, localized operations on other regions of the problem space. This flexibility cannot be over-emphasized. Further, the absence of any control implies that changes made by operators are purely localized. Thus improvements made by operators may be at odds with each other over regions of the problem space! Hence, the destructors are extremely important operators, since they help herd the population of solutions towards improvement by weeding out bad solutions.

Talukdar and Souza [58], and Murthy [34] have demonstrated that the ATeams architecture can efficiently find solutions for very hard problems (Traveling Salesman Problem, Robot Manipulator design). We decided to use the ATeams architecture for the constraint satisfaction problem for the following additional reasons:

- It is an inherently parallel technique: it yields families of feasible solutions;
- It is possible to construct an ATeam with extremely simple and localized modification operators, as shown in the following section;
- The resulting framework is extremely general. The only requirements are that the constraints can be evaluated, and that qualitative knowledge about localized improvements can be built in. It can thus handle mixed constraint formulations: qualitative



## 5.2 Asynchronous Teams of Autonomous Agents

---

constraints and arbitrary numerical constraints; and in fact, any constraint which can be captured in a programming language;

- The solution architecture is inherently object-oriented. encapsulation of localized evaluation and improvement knowledge, message-passing and polymorphism are concepts easily supported by C++, the language of implementation for this research; and
- ATeams support modularity. Agents can be added or removed at any point. This is particularly useful in a dynamic constraint environment, when constraints are regarded as agents.

### 5.2.1 A Conceptual ATeam for the Constraint Satisfaction Problem

The various components of a conceptual ATeam for the constraint satisfaction problem are:

- *Memory.* The shared memory containing the candidate solutions may be seeded with randomly initialized solutions, or with solutions obtained in any other manner. The size of this memory is implementation dependent; and
- *Agents.*
  - *Evaluation.* For the constraint satisfaction problem, the objective is to satisfy each of the constraints. Hence the evaluation criteria is merely an evaluation of the constraints. The evaluation of a design solution is a combination of the evaluations returned by each constraint. Each constraint encapsulates knowledge about evaluating a given design with respect to itself. The implementation of this evaluation knowledge is discussed in the following sections;
  - *Modification.* Each modification operator corresponds to a constraint and seeks to modify a solution chosen from the design store to reduce the violation of this particular constraint only. Thus each of these modification operators produces local improvements using only a subset of the design criteria. The modification operators encapsulate improvement knowledge specific to the nature of the constraint. The constraints could thus be regarded as agents themselves. We

### 5.3 Constraint Evaluation and Improvement

---

have chosen to implement extremely simple operators; a set which could potentially be extended to include more sophisticated techniques. These improvement operators are described in section 5.3;

- *Crossover and Mutation.* These are operators which are derived from the concepts of crossover and mutation in Genetic Algorithms. They evaluate random recombinations of solutions. These operators serve the same purpose as in a GA. They preserve the diversity of the solution store, and help propagate good traits across solutions; and
- *Destroyers.* These agents selectively delete solutions from the memory based on their evaluations. The primary purpose is to control the size of the store and to concentrate the efforts of the modification operators on more promising solutions.

A schematic diagram (Figure 5-1) illustrates the ATeam for the conceptual design problem.

### 5.3 Constraint Evaluation and Improvement

The conceptual description of the ATeam has emphasized the importance of the evaluation and improvement operations for a design solution. These operations are encapsulated within each constraint. Thus as each constraint is entered in a dynamic design environment, it enters the design world with knowledge about evaluating and improving a potential solution with respect to itself. Each constraint can thus be treated as not only as further defining the problem space, but also as actively driving the design solutions towards the feasible region.

#### 5.3.1 Constraint Evaluation for Qualitative Relationships

The constraint evaluation for the qualitative relationships is now discussed.

#### Evaluating Primitive Relationships

To evaluate whether a particular primitive relationship is satisfied, the two intervals in space must be specified in terms of their starting and end points (minimum and maximum) along the vector defining the relationship. With each relationship, the source is identified as

### 5.3 Constraint Evaluation and Improvement

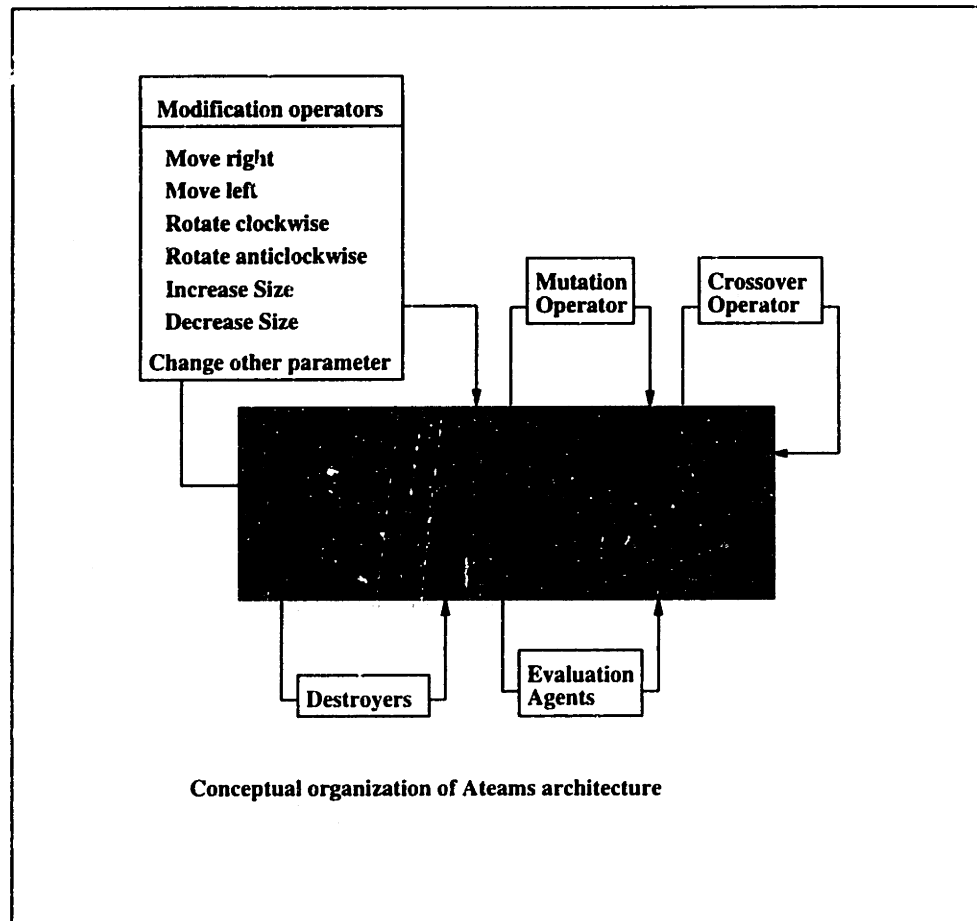


Figure 5-1: A schematic diagram of the ATeam for solving the conceptual design problem.

the first interval in the relationship and the target is the second interval. In a relationship such as  $A$  abuts  $B$ , the projection of  $A$  is taken to be the source interval and the projection of  $B$  as the target interval.

Given a point and an interval, the algorithm checks to see if the relationship is satisfied; the violation of the relationship is quantified by a penalty between 0 and 1. For each relationship, this scheme is illustrated in the figure 5-2. If the relationship is satisfied, the evaluation is 1. Note that the choice of penalty function is somewhat ad-hoc; any other penalty function could be equally used. The only requirement is that the function provide an unambiguous, ordinate measure of violation. For a qualitative relationship, this penalty function could be viewed akin to a set membership function in a fuzzy set. Each

### 5.3 Constraint Evaluation and Improvement

---

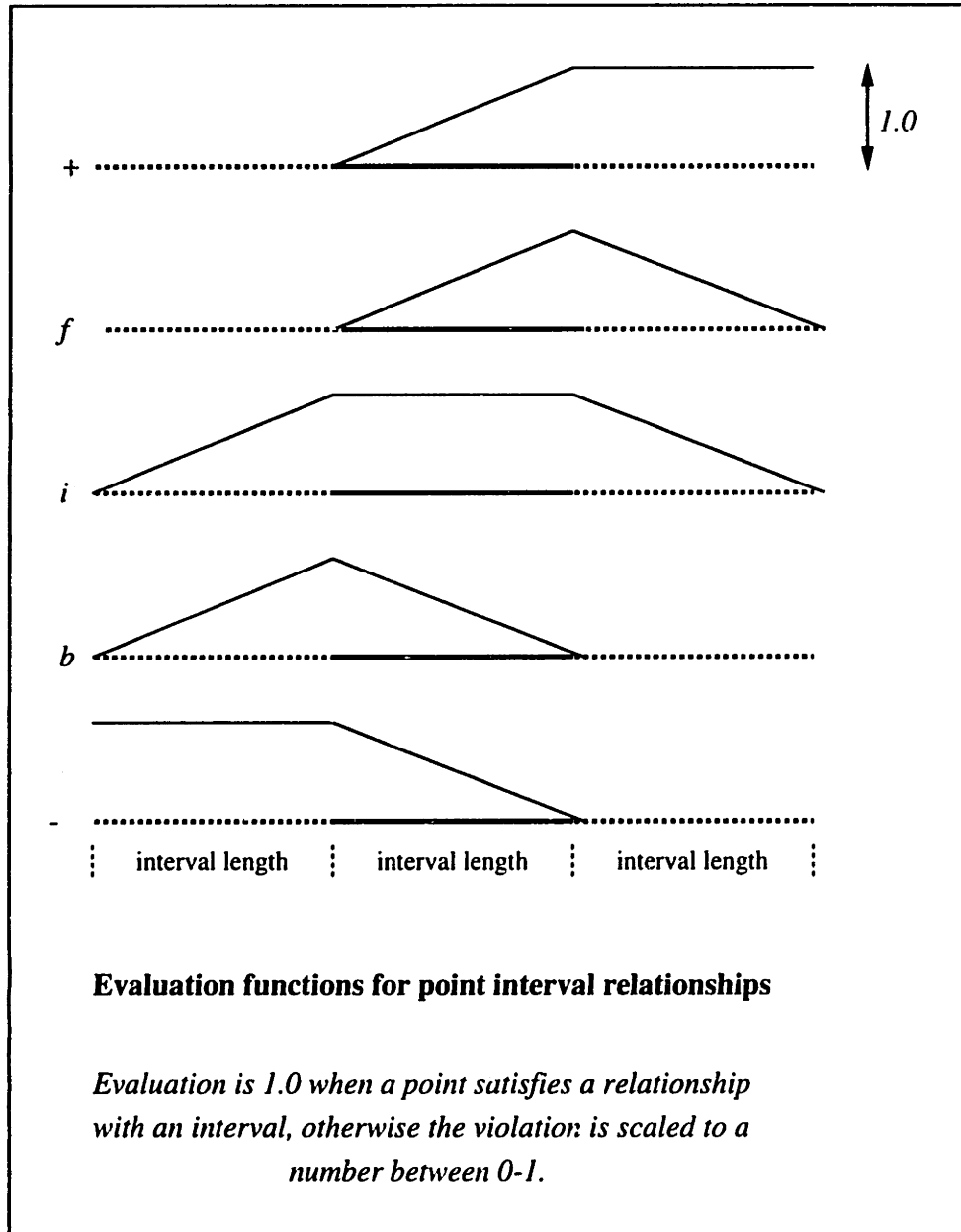


Figure 5-2: Evaluation function for point interval relationships

## 5.3 Constraint Evaluation and Improvement

---

interval interval relationship is a combination of two point interval relationships between the endpoints of the source, and the target treated as an interval. Each one of these is evaluated separately. The degree of satisfaction of the two point interval relationships is multiplied to give a composite measure of the degree of satisfaction of the interval interval relationship. Again, this is a number between 0 and 1.

### Evaluating Higher Level Relationships

A representative degree of satisfaction for the disjunctive class of relationships is obtained by evaluating each component of the disjunction and taking the degree of satisfaction of the best component - closest to 1 - to be its evaluation measure. 3D relationships are evaluated by taking the product of the evaluations of their components. The composite evaluation structures are then tagged along with the design solution and the design is returned to the store.

The evaluation criteria chosen above for the qualitative constraints give a quantification as well as a feel for the degree of satisfaction of the constraints. While they give an unambiguous, ordinal measure, they are yet highly subjective. It is difficult to get a feel of how the composite measure of evaluation of an interval interval relationship relates to its degree of violation. For instance, given a measure of violation of 0.5 for a point interval relationship, it is easy to visualize how the relation is violated. On the other hand, given the same measure for an interval interval relationship, it is difficult to visualize all possible combinations of point interval violations that can result in a composite violation of 0.5. The difficulty is compounded in higher order relationships as the combinations get more complex. However, the ATeam uses the evaluations only to give an indication for the direction of improvement sought in a particular solution. This simple evaluation scheme presented above was found to be quite sufficient.

Evaluating numerical constraints given values for the variables is a simple inorder traversal of the expression tree and deserves no further mention.

### 5.3.2 Improvement

We now turn to another important aspect of the solution scheme: improvement suggested by the modification operators. Recall the claim that the modification operators required by the ATeam could be built in into the constraints. When a design is sent to a constraint to be evaluated, the constraint returns an evaluation structure which is tagged

### 5.3 Constraint Evaluation and Improvement

---

along with the design. The evaluation structure contains some additional information: a set of possible modification operators which might potentially improve the design. Note that we are deliberately reserved about the improvement potential of a modification operator. This is because the modification operators operate independent of each other, and they might well have deleterious effects on some other aspect of the design. The destroyers weed out solutions resulting from poor interactions. Thus the operators can be restricted to a very simple set. For the qualitative spatial algebra, these are:

**{ LEFT, RIGHT, SMALLER, BIGGER, CLOCKWISE, ANTICLOCKWISE }**

Each of these objects specifies a change in an object participating in the relationship along a direction which is stored in the evaluation structure. e.g., **LEFT** means a negative movement along a particular direction specified as a vector in space. This vector is the defining direction of a relationship or its components.

#### Improving Primitive Relationships

Upon evaluation of a relationship, a modification operator is associated with each of the objects involved in the relation. For instance, if the interval interval relationship A (++) B is violated, improvement is possible by either of two modification operators A-RIGHT or B-LEFT. If a relationship is satisfied, no operators are specified.

#### Improving Disjunctions

For disjunctions, each component interval interval relationship is evaluated and some modification operators are associated with it. The algorithm compares the evaluation of all components and retains only the modification operators and evaluation associated with the least violated constraint (which can be looked on as the constraint with the best hope for satisfaction).

#### Improving 3D Relationships

A set of modification operators for 3D relationships is obtained as a union of all modification operators associated with each of its component relationships. No operators can be discarded since each 3D relationship is a conjunction of relationships. In order to improve a 3D relationship, any member of the operator set can be applied.

### 5.3 Constraint Evaluation and Improvement

---

Relationships such as *is – perpendicular*, if violated, can only be improved by rotating the objects. For these, the operators **CLOCKWISE** and **ANTICLOCKWISE** are specified. These operators refer to the objects as a whole and do not have any natural correspondence to the modification operators for interval interval relationships. The rotation is assumed to be about the centroid of the object.

Although the modification operator set suggested by each relationship may not cover all possible ways in which a relationship may be improved, it is not necessary to always have a complete set of operators. This fact is borne out empirically in our examples as even incomplete operator sets do not exceptionally hinder the search process. However, the more complete the set, and the more sophisticated the improvements suggested, the faster the algorithm converges to a solution.

#### 5.3.3 Modification Operators for Algebraic Constraints

This section proposes a scheme to handle improvements for arbitrary algebraic constraints.

Each algebraic constraint is parsed into an expression tree involving operators, constants and variables. Suggesting modification operators for the solution being evaluated is a two-pass process. At each operator node in the tree, the values of the expressions to the left and right node children expression are stored when the tree is first evaluated. For modification, note that the root of a parse tree will always contain an expression as the left child and a constant as the right child. If a constraint is found violated at the root operator, the amount of increase or decrease to be made to the left expression is known. This message is propagated to every node in the tree such that the actual modification made to the variables results in an increase or decrease in the net value of the left-hand-side of the constraint.

Any increase or decrease with reference to an operator means increasing or decreasing the net value of the expression formed by concatenating the children and the operator itself: binary operators in the middle of the two children and unary operators before the child. Therefore to increase the net value of such an expression, it is necessary to know how to change the values of the child sub-expressions.

Such qualitative knowledge can be built into each mathematical operator. For example, consider the operator  $+$ . The only way to increase an arbitrary expression such as  $expression1 + expression2$  is to send the message *increase* to either (or both) of the expressions  $expression1$  and  $expression2$ . With a  $*$  operator, it might be more complicated.

### 5.3 Constraint Evaluation and Improvement

---

A message such as *increase* sent to  $*$  must be interpreted according to the most recent evaluations of its child expressions. For instance, if both children evaluated positive, the message *increase* must be sent to any one or both children to guarantee an increase in  $expression1 * expression2$ . On the other hand, if one child evaluated positive and the other negative, the message sent down to the positive child will be *decrease*, and the message sent to the negative child will be *increase*. The example in Figure 5-3 illustrates this idea. The current implementation includes a limited number of such mathematical opera-

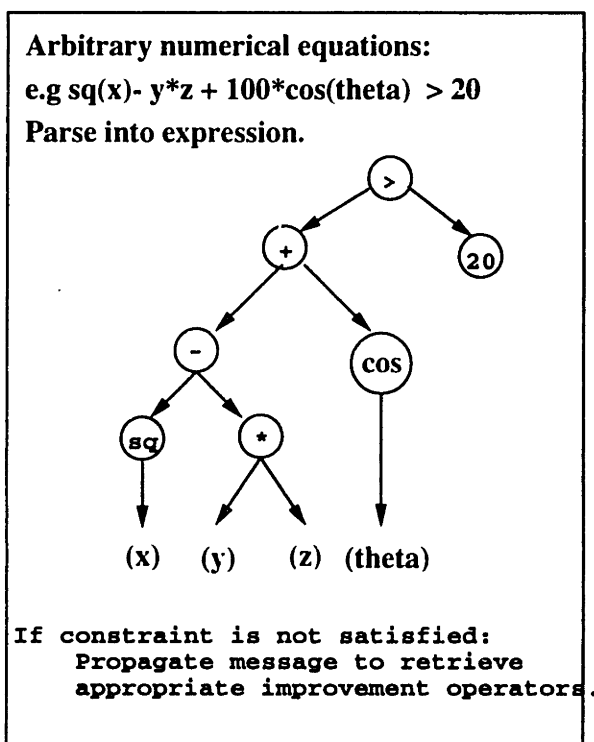


Figure 5-3: Improving numerical constraints

tors. We believe that this scheme is extensible to arbitrary mathematical functions; testing on this issue is pending.

A final note on the modification operators: They merely suggest a direction of change and the parameter to change. When the modification is actually invoked, the message is sent to the object that represents the variable that needs to be changed. The actual amount of change is left to this object to decide.



## 5.4 Implementation

---

### 5.4 Implementation

This section presents details of the implementation of the ATeams algorithm and also the overall structure of the algorithm. The nature of the scheme is inherently object-oriented, and the implementation is fairly simple and elegant. Modeling the system in an object-oriented fashion provides flexibility by allowing the various operators to be incorporated independently into the system when desired.

The object-oriented ATeams implementation reported in this section is credited to Humair[21]. Humair also reports on the relative merits of ATeams *vs* Genetic algorithms as general search techniques.

#### 5.4.1 Solution Representation and Storage Classes

**Dobj** class represents a bounding box for a design object. Each object has a string identifier and an array of nine values containing the configuration variables of the box. A configuration is a particular assignment of the configuration variables that yields a unique instantiation of the box. These variables are the position of the centroid  $(x, y, z)$ , the sizes along each of the axes  $(size_x, size_y, size_z)$ , and the rotations about the global axes  $(\theta_x, \theta_y, \theta_z)$ . This real-valued variable space is discretized according to a user-defined mesh size. When a **Dobj** is initialized, these parameter values may be randomly created or copied over from another object. Each **Dobj** stores the information about the local reference axis system of the object relative to the global frame. The message interface includes a method which returns the starting and end points of the projection of an object along an arbitrary vector. This is useful in the evaluation of point interval relationships as discussed earlier. A second method allows improvement in the values of the configuration variables along a certain direction.

**Design** class is the representation of a candidate design solution. Each **Design** object stores an array of **Dobjs**. It has information about the number of objects in each design, the number of constraints, and an array of **Evaluation** objects, one for each constraint.

#### 5.4.2 Operators

The operators represent agents in ATeams. An operator requests a design from the store of designs and modifies or destroys it accordingly. The class hierarchy of operators is shown in figure 5-4. **Operator** class is an abstract class from which all other operators are

## 5.4 Implementation

---

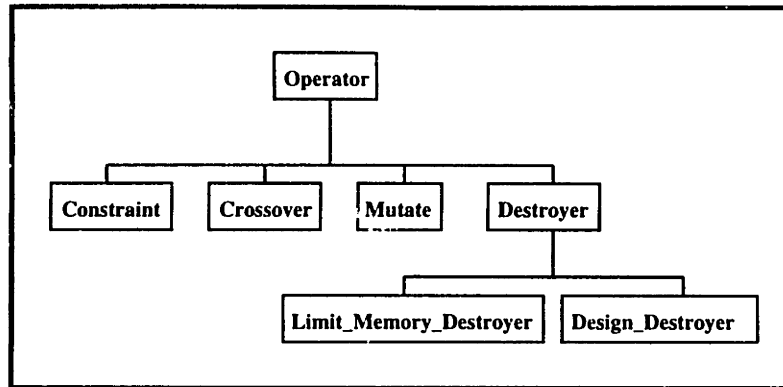


Figure 5-4: Class hierarchy of operators

derived. It is convenient for providing a uniform interface for all operators. **Constraint** class represents the design constraints input to the ATeams. Member methods include evaluating a particular design and improving it. The method `fire` encompasses the entire operation of getting a random design from memory, copying it, improving the copy and evaluating it before adding it in the store.

**Crossover** class is the analogue of the crossover operator used in genetic algorithms. Its interface includes methods to get two random designs from memory and mate them. Mating takes place by picking a random object and swapping all objects after and including this one with the other design. The mated designs are evaluated before putting in the store.

**Mutate** class is the analogue of the mutation operator in genetic algorithms. The relevant method gets a design from memory, copies it, and randomly assigns a value to a configuration variable for a randomly picked object in the design. The design is evaluated before inserting in the store.

**Destroyers** are used to destroy bad designs with a given probability. They are necessary for controlling the size of the population and to make sure that the effects of the other operators are concentrated on the most promising designs. **Design\_Destroyer** probabilistically deletes bad designs from memory based on their overall evaluation. **Limit\_Memory\_Destroyer** is used to keep the size of store within reasonable limit. **Duplicate\_Destroyer** is used to destroy duplicates from the store.

## 5.4 Implementation

---

### 5.4.3 Overall ATeams Algorithm

The algorithm operates on a discretized space for each of the configuration variables involved in the design. The choice of the number of discretizations is arbitrary, but important since it determines the size of the search space.

The overall algorithm proceeds in a randomized manner. At many places in the algorithm, probabilistic steps are taken depending upon certain criteria. The probability of picking up any design from the memory is independent of the evaluation of the design. This probability is given by  $1/n$ , where  $n$  is the number of designs in memory at that particular time. The **Design\_Destroyer** retains all designs with an overall evaluation greater than 0.75, but probabilistically destroys all other designs with a probability of 0.8. The probability that a design destroyer is fired is variable. We have found 0.4 to be a stable value for this probability. After every 1000 iterations, the limiting memory destroyer goes through every design in the memory, and destroys it with the probability  $1 - evaluation^3$ . The choice of this probability was based on the observation that it provides a sharp discrimination between the probability of destruction of good and bad designs owing to the sharp slope of the cubic function near 1 as contrasted to its behavior near 0. When a duplicate destroyer is fired, duplicates are destroyed with probability 1. All through the algorithm, the elitist strategy is maintained, i.e., the designs with the best evaluation in the store is never destroyed.

The **Limit\_Memory\_Destroyer** maintains the integrity of the store by probabilistically destroying the poor solutions after every 1000 iterations. This periodic expansion and contraction to original size of the store is very effective. It helps to concentrate the efforts of the modification operators on the good designs, while guarding against premature convergence.

The overall algorithm can then be described as:

1. *Create initial store (randomly or seed with external solutions).*
2. *Evaluate all solutions and place in memory.*
3. *While termination condition is not met*
  - If necessary fire destroyer to limit memory*
  - Pick operator randomly (modification / destroyer)*
  - If operator is a modification operator,*
    - make change, evaluate new design, place in store*

## 5.5 Summary and Discussion

---

*If operator is a destroyer,  
destroy probabilistically based on evaluation.*

## 5.5 Summary and Discussion

With respect to a design environment, this research has not properly addressed one of the important issues of constraint management. i.e., constraint consistency. Unfortunately, one of the very strengths of the architecture is also a limitation. This chapter elaborated on being able to handle mixed formulations of constraints. But the traditional consistency detection algorithms (usually involving graph closure of some kind) may not be applicable over the mixed formulation. One potential approach is to have a pre-processing stage, when consistency detection is applied over subsets of the constraint formulation, treating them independently. Another approach is to follow a more heuristic-based route, and identify the subset of inconsistent constraints by examining the results at the termination of the algorithm. This thesis does not present answers to this question.

Another interesting issue is termination of the algorithm. It is very difficult to *a priori* determine termination of the algorithm. In the general case when it is not possible to specify the goal design state, it may be necessary to preset the execution time or number of iterations or number of feasible solutions desired. In this respect, the algorithm shares the limitations of the genetic algorithm. Arguably, this may not really be a limitation in a design environment, since it allows the flexibility to terminate the algorithm at any point, and use the best approximation to a solution found thus far. Chapter 8 presents a discussion of how this iterative approach may be turned into a strength.

The algorithm is easily able to handle a dynamic design environment in which constraints (or objects) are incrementally added. A reasonable computation strategy at this point is to simply continue the search with the results of the last run being used as a seed population. In fact, this is an approach similar to the one used by iterative numerical solver based constraint management techniques.

Concluding the chapter, the solution architecture implemented here is quite promising. Chapter 8 presents some interesting results on a test design example.

---

## Chapter 6

# Geometric Representation for Conceptual Design

This chapter addresses the issue of knowledge representation for geometry. The overall objective of this thesis is to automate the generation of form alternatives for design artifacts. This implies that the computer has at least an abstract notion of the geometry of primitive components. In traditional CAD systems, this knowledge has been in the form of generic methods and procedures concerning the basic building blocks. In a CSG-based solid modeler, these building blocks might be cubes, cylinders, spheres, etc. These primitive blocks permit the synthesis of a wide variety of objects. Yet, they have little direct relation to any engineering domain. This chapter discusses abstraction as a complexity-reducing mechanism to represent geometry of engineering artifacts, with a specific focus on civil engineering. This abstraction may be of various types: parameterized shapes, generalization, composition, idealizations, etc. The idea of the form definition scheme discussed in this chapter is to use some or all of these concepts to represent geometric shapes of engineering objects in a higher-level domain representation. The focus of representation is conceptual design, with its requirements for abstract and incomplete descriptions of geometry.

The chapter is organized as follows: Section 6.1 briefly describes the problem of knowledge representation for geometry. It further describes the need for multiple levels of abstraction at various levels of detail, and presents a solution to the problem. Section 6.2 presents the form definition scheme and some interesting implementation issues related to form definition.

## 6.1 Knowledge Representation for Geometry

---

### 6.1 Knowledge Representation for Geometry

If a computer is to reason intelligently about geometry, it must have a representation scheme for the objects it reasons about. Researchers in mechanical engineering have termed these representations as *features*. Features are an attempt to replace simple geometric elements as the primary building blocks with higher-level modeling elements that more directly relate to engineering domains. Many successful applications of feature-based design have been demonstrated ([8], [10], [45]). In the words of Mantyla [28],

The success of feature based modeling techniques is largely determined by whether a useful taxonomy of feature types can be identified and organized in a modeling system, and whether application-oriented data and knowledge bases can be conveniently organized on the basis of this taxonomy.

This chapter demonstrates such a taxonomy in a civil engineering context. Representations of primitive components are used as features which form the basic building blocks in the design. In addition, the chapter addresses several important issues relating to knowledge representation geared towards conceptual design. Some of these issues are now presented.

#### 6.1.1 Abstraction as a Representation Mechanism

The previous discussion highlighted the need for a modeling scheme which relates CAD primitives directly to the domain they seek to model. AI-based techniques could be used to structure geometric information in the desired manner. But how can the underlying differences between AI-based and geometry-based representations be reconciled? The underlying computational paradigm for traditional geometric modeling techniques has been *procedural*. Existing computational theory on geometry is expressed strictly in algorithmic terms. Information required for the algorithms to operate is implicitly held within the structure of the program. AI-based representation schemes, on the other hand, have a distinct separation between the knowledge and the algorithms used to process the knowledge. Recently, researchers in design have recognized the need to develop declarative representation schemes for geometry ([67], [46]).

The object-oriented representation paradigm provides a solution which allows integration of a natural hierarchical decomposition of a problem domain with the procedural computations necessitated by geometric algorithms. The knowledge-based system building shell COSMOS enables association of declarative knowledge with the abstractions in

## 6.1 Knowledge Representation for Geometry

---

the geometric hierarchy. The scheme proposed in this chapter exploits this integration of procedural and heuristic programming provided by COSMOS.

### 6.1.2 Multiple Levels of Abstraction in a Unified Framework

Observations of human designers at work indicate that they typically use multiple modes of graphical representation for an object. These modes may depend on various factors: the stage of the design, the amount of detail available, the current focus of attention for the design, etc. These modes of representation may be non-uniform, e.g., a slab supported at its edges by four beams might have the slab shown as a surface representation, and use solid representations for the beams. To handle such situations in current CAD programs, special purpose mappings would have to be implemented between various modes of representation and display.

To solve this problem, this research uses a representation of geometry based on a non-manifold geometric engine (GNOMES). The non-manifold model used by GNOMES (SGC model, [38]) provides a larger representation domain as opposed to conventional two-manifold geometric models. Current CAD systems utilize the concepts of two-manifold solid modeling, which make the assumption of closed point-sets. The SGC non-manifold model enables representation of mixed dimensional collections of pointsets. Thus the system can model wireframe, surface, and solid models, along with non-manifold conditions in a unified framework of data structures. The representation is structured into a set of geometric abstractions that can be interrogated and managed directly by application programs; these abstractions are borrowed as-is from [71]. Such a representation allows a unified framework for multiple levels of abstraction.

### 6.1.3 Evolving Geometric Descriptions of Objects

Design is an evolving process. The modeling requirements of applications at the conceptual design stages are different from the requirements at the detailed design stage. For example, a beam at various stages of the design may be modeled as a straight-line, an enclosing box with overall dimensions, an I-beam in wireframe mode, or a solid I-beam. Traditional models invariably require the user to prescribe fully defined constructions from a set of available primitives and operators. If we are to break free from the shackles imposed by such restrictions, we must address the following problems:

## 6.1 Knowledge Representation for Geometry

---

- (1) How do we allow a concept representation to evolve as detail accumulates in the design process ? One way to solve this problem is to use the lowest-level primitives available to the solid modeler, i.e., points, lines, etc. If the user were to directly interact with the solid modeler, we would then just update the model. But if the computer is to reason at a higher level about the shape, the representation must be more intelligent. Consequently, if we do not wish the computer reasoning process to commit prematurely to a particular class of shape, we must allow for some form of migration for the shape definition through various abstract levels. Consider the example of a beam being designed. Initially all we may know about the beam is the length. At this point we cannot say whether the beam is an I-beam or a C-beam, neither would we expect the computer reasoning framework to commit to any description so specific at such an early stage. As designer decisions influence the design process, we might be able to confidently assert a decision at some point during the process. In terms of knowledge representation, this poses a problem. We have embarked on a quest for a higher-level domain representation. Yet the incremental definition of an object would seem to necessitate some kind of a classification scheme to identify the current shape from a predefined knowledge hierarchy. We address this problem in section 6.2.
- (2) Even if we were able to commit early on to a certain abstract shape, complete information needed to display the object may be absent. To deal with this problem, we use the concept of a 'minimal representation' at various levels in the domain-dependent abstraction hierarchy. At each level, we identify the minimum information required to display the object. For example, the minimal information to evoke the image of a beam is a length. For a slab, this information would be the planar dimensions.
- (3) How do we manage the communication between the geometry of the object and an intelligent design agent? We describe a geometric abstraction class called **Geometry** which serves the purpose by providing a uniform interface with the spatial reasoning component of the design agent. We further describe the additional intelligent behavior that we can embed inside this class.



## 6.2 Implementation

---

### 6.1.4 Domain Taxonomy

We have discussed the need for a domain taxonomy which enables us to relate the primitives directly to the domain they seek to model. In particular, this thesis focuses on civil engineering shapes at a conceptual design stage. There are three main issues in such a classification:

- (a) To study the kinds of shapes commonly used.
- (b) To decide on a basis for taxonomy of these shapes.
- (c) Having decided on a classification, we must decide on how we can impart intelligence to these abstract classes. Here, we use intelligence in the narrow sense to mean information that can be embedded inside the classes representing these shapes: for example, construction procedures, parameters, how each object should respond to display messages, whether an object has enough information to display itself, or whether it should defer the display to a more abstract class, responses to generalized spatial queries regarding position, orientation, surface area, volume, etc.

Section 6.2 presents our approach to the taxonomy problem.

## 6.2 Implementation

The implementation addresses each of the issues raised in the previous section. The domain taxonomy is based on standard civil engineering knowledge, as presented in [42]. We define spatial classes corresponding to commonly used domain objects. These spatial classes form a layer of abstraction over the actual geometry representation, which is based on the GNOMES model. The classes present a uniform interface to communicate with the application which drives the design process. The facility for incremental specification is primarily provided by the C++ feature of *inheritance*. We describe an implementation to deal with the classification problem presented by an evolving geometry description. We further discuss the behavior encapsulated within each of the classes in the toolkit. The implementation discussed in the rest of this chapter was done by Georgios Margelis as part of his Master's thesis [29].

## 6.2 Implementation

---

### 6.2.1 Taxonomy

In this work, objects are classified into line-forming elements and surface-forming elements, following Schodek [42]. Line-forming elements are further classified into straight and curved and surface-forming elements into planar or curved. The curved-surface elements may be of single or double curvature. This particular hierarchical representation identifies shape information of primary and secondary importance in the design. This is shown in Figure 6-1. Implementationally, each of the classes presents a polymorphic interface for response to messages. Thus the classes have a similar architecture, and redefine the generic interface provided by the root class **Engineering\_object**.

Each of the classes has a set of constructors which allow creation of objects under various circumstances: we discuss some of these in the next section. Each class further possesses methods to propagate and compute attribute values based on internally encapsulated knowledge. The display method encapsulates knowledge about the minimal information required to display an object.

The class **Engineering\_object** is the top level class. It provides the generic virtual interface for all classes. In addition, it contains information about a bounding box, which fully contains each engineering object. Further it provides operators for the union, difference and intersection of engineering objects. These operators perform the Boolean operations between the geometric models that represent each engineering object. These may yield arbitrary geometric models, which are stored in the class **Engineering\_object** under the attribute **GNmodel\* model** (The type **GNmodel** is a type from the GNOMES non-manifold geometric engine and represents a geometric model, see Figure 2-2).

The classes **Straight\_line**, **Line\_rect\_cross\_section**, **Line\_circ\_cross\_section** and the associated solid and hollow section representations allow a progression of shape descriptions for linear elements: beams, columns, pipes, etc. The family of shapes related to the surface elements are used to represent slabs, plates, and shells. Most civil engineering objects can be constructed through some combination of these basic primitives.

### 6.2.2 Evolving Description

The classification described in the previous section allows for an evolving shape description through *external* and *internal* modes of abstraction. The external abstraction refers to the migration of an instance down the class hierarchy as detail is accumulated. The

## 6.2 Implementation

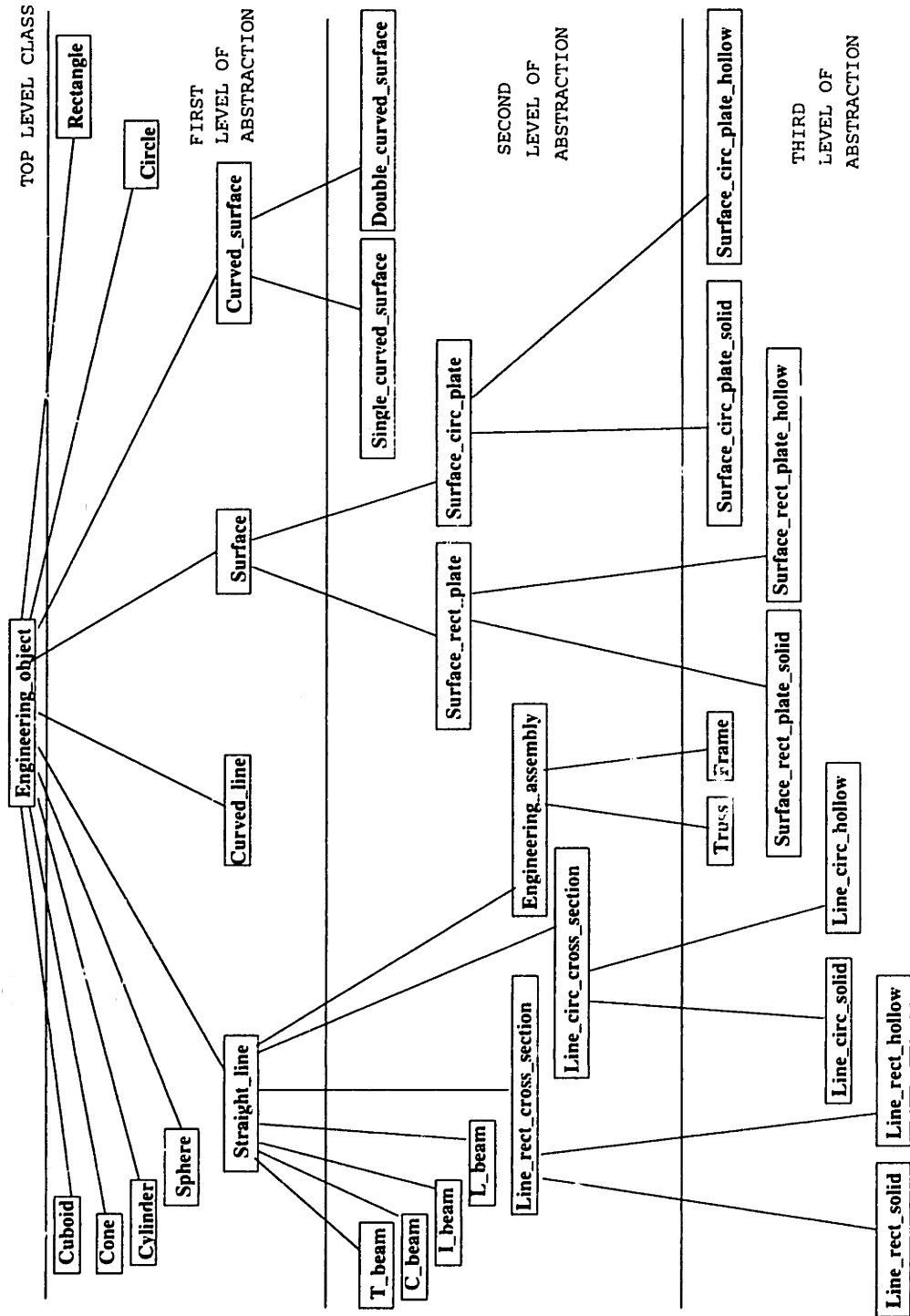


Figure 6-1: Shape classification

## 6.2 Implementation

---

implementation mechanism in C++ to achieve this is quite simple: we merely define copy constructors for each class which copy attributes from the immediate parent class. Note that “migration of an instance” is a misnomer; in reality each upgrade replaces the instance with a new one. A sample fragment of code illustrates this procedure for a beam.

---

```
Geometry* beam = new Geometry("beam1"); // create Geometry called 'beam1'.
beam->create_object(Geometry::STRAIGHTLINE); // created as a straight line
.. set attributes ..
beam->display(); // beam is now displayed as a straight line
beam->upgrade(Geometry::LINERECTCROSSSECTION); // set to a beam of
// rectangular section
.. set attributes ..
beam->display(); // beam is now displayed to have a rect. section
beam->upgrade(Geometry::LINERECTSOLID); // upgrading the geometry to
// a solid section now.
. set attributes ..
beam->display(); // solid rect. beam is displayed now
beam->upgrade(Geometry::I); // set to an I beam
beam->display(); // display shows an I-beam
```

10

---

This solution is admittedly simple, yet partially achieves the need for gradual definition. It is conceptually possible to move across the hierarchy, rather than the strictly up-and-down process that we have laid out. Semantically, this will still translate to a move up the hierarchy till a common parent is reached (losing some detail in the process) and then moving down a new branch in the hierarchy. This would entail some redefinition of geometry.

The display methods further encapsulate some knowledge, which might be considered an *internal* abstraction. We use the concept of a minimal representation to decide whether the object can be displayed or if the display must be deferred to the parent. For example, the display of an I-beam in its entirety will involve the length, width, height, thickness of flange and thickness of web. Yet, if only the length is known, display can be deferred to the **Straight\_line** class. As bounding box dimensions become known, it is possible to compute the parameters defining the I-section, except for thicknesses. Similarly, display modes for **Line\_rect\_hollow** and **Line\_rect\_solid** objects are wireframe and solid respectively. It is thus possible using the GNOMES geometric model to describe multiple levels of abstraction and multiple modes of display.

---

## 6.2 Implementation

---

### 6.2.3 Geometry Interface

We have described the geometric abstractions module which allows the evolution of geometry. But we have not yet discussed the control of flow which allows the geometry updates to be performed.

The class **Geometry** is an abstract geometric class (not necessarily corresponding to any particular geometric model) provided to serve this purpose and to act as the link with the client applications. Figure 6-2 represents the possible ways of communication between the **Geometry** class and the two modules: CONGEN and the constraint manager. CONGEN represents the functional modeler which models the symbolic evolution of the design. The constraint manager determines the alternate geometric configurations which are functionally viable. The **Geometry** class provides a unique identity for the geometry

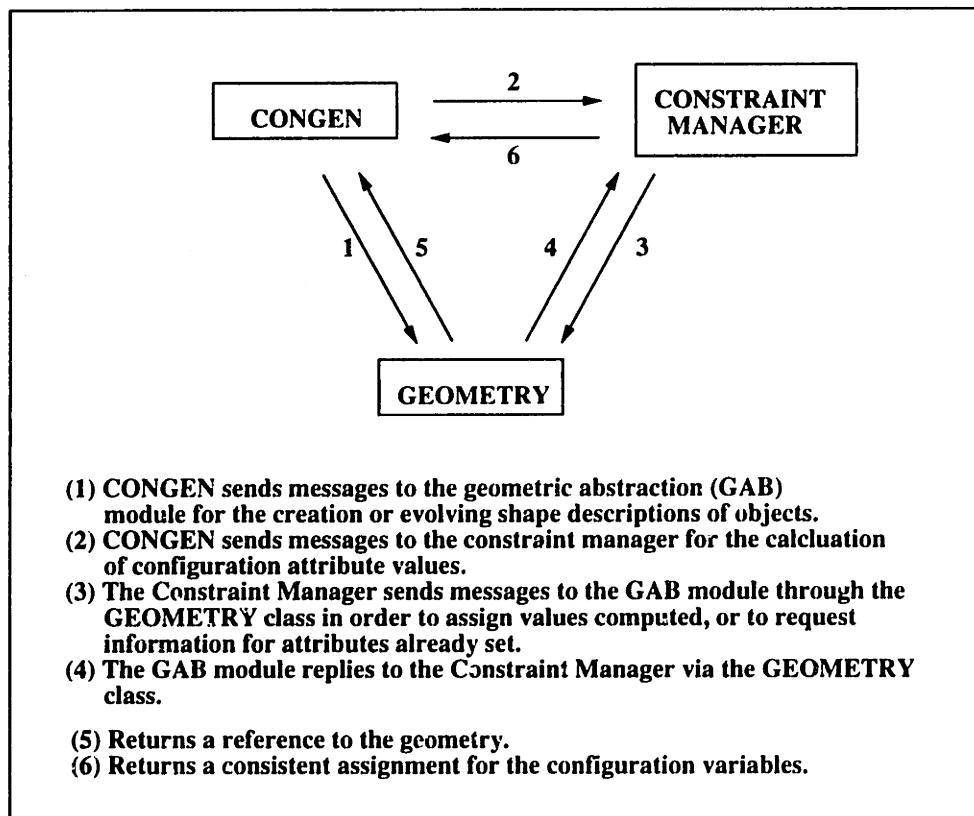


Figure 6-2: Communication between modules

of an object, allowing for referential integrity as different programs access an evolving

### 6.3 Summary

---

geometry. In essence, it serves as a wrapper which forms the communication interface for the evolving geometry of the object. Further, the shape knowledge representation discussed so far only allows the generic shape classification in a domain. The **Geometry** class is a convenient mechanism to further augment this knowledge with project-specific and context-specific knowledge regarding default sizes and orientations. We could also set dependencies for geometric attribute values. Presently, we use rule-based and constraint mechanisms to represent this knowledge.

Currently, our implementation defers the decisions for shape evolution to the user, i.e., the user explicitly identifies an upgrade to switch from a class in the domain hierarchy to a more detailed description. These decisions are often a fallout of the design process. However, the **Geometry** class provides an interface to isolate these upgrade procedures. It is conceivable that an automatic classification procedure could be used to direct this control. Some work on this has been reported in literature [18]; but there are a lot of unresolved issues in implementing such schemes in a statically typed language such as C++. We have finessed this problem in a sense by providing a partial solution and implementation mechanism.

In addition, the **Geometry** class provides mechanisms for dimension-independent spatial queries (adjacency checks, intersections, enclosures), generalized transformations (translations, rotations) and representing spatial relationships. In this respect, the **Geometry** class is similar to the **Space** class described in [71], and incorporates some of the ideas therein.

### 6.3 Summary

In this chapter, we have defined the requirements for geometric representation for conceptual design. We have also addressed some of the implementation issues involved in representing incomplete and evolving geometries. The GNOMES geometric modeler also provides a complete set of geometric editing tools, which allow the user to directly modify geometry.

---

## Chapter 7

# CONGEN Implementation

This chapter describes the implementation of the CONGEN system. Some aspects of this implementation have already been presented previously, whenever such presentation was necessary to explain the concepts involved. This chapter provides greater detail on the elements of the overall framework. Each of the sections also shows screendumps of the user interface, as necessary. Section 7.1 gives an introduction to the implementation framework and some of the base tools involved. Section 7.2 explains details of the process elements. Section 7.3 refers to the C++ classes involved in modeling product information within the CONGEN framework. Section 7.4 deals with geometry representation. Section 7.5 deals with aspects of the constraint management framework. Section 7.6 shows important modules of the user interface, and gives an idea of the overall system functionality from a user perspective.

### 7.1 Overall Implementation Framework

CONGEN is implemented as a layered application over an underlying object-oriented database, EXODUS [7]. EXODUS provides a programmatic interface to client applications. This interface is a language called **E**, which is an extension of C++ to provide database types, and persistence [36]. **E** further provides iterators, collections and transaction support. The **E** counterpart of a C++ class, **dbclass**, is the equivalent persistent type for any class. In CONGEN, all classes which may potentially have persistent objects residing on the database are declared to be **dbclasses**.

The database allows the creation of persistent objects on a *persistent heap*. In CON-

## 7.1 Overall Implementation Framework

---

GEN, these persistent objects are grouped together into *applications*. An application represents all knowledge and user-generated information relevant to a particular project. All application data is handled by a special class called **Data\_manager** (see Figure 7-1). Either the user or the program may instantiate any one of the primitive constructs provided by CONGEN. The **Data\_manager** handles all further interactions with these objects within the application. The user may also define additional classes relevant to the domain. These classes are maintained by the COSMOS object management. COSMOS is one of the underlying tools which is tightly integrated with CONGEN. The following subsections provide an overview of the two main tools used by CONGEN: COSMOS and GNOMES. The focus of discussion is limited to the main classes which provide the interfaces to the underlying functionality provided by these tools.

### 7.1.1 COSMOS

The design document of COSMOS [48] describes the implementation details of COSMOS. COSMOS is a C++ (rather, E) environment for building knowledge-based systems. COSMOS provides a powerful object management system. The object management system allows end-users to define arbitrary data types. The system generates C++ code for all user-defined types, compiles and dynamically links this incremental code with the running system. All class information is stored on the object-oriented database. Subsequently, the object management provides various functionalities to operate on these user defined classes: the user may instantiate any of the classes, access objects by name, access and modify any of the attributes by name, and invoke methods associated with these objects. This is a complete run-time environment built over C++. The functionalities of the object management may be accessed by two main classes, **Class\_Manager** and **Instance\_Manager**, which handle the class information and the instances respectively.

Each of the classes generated also has an interface to the inference engines. COSMOS provides both a forward chainer **Ie** and a backward chainer **BC**. The rule format followed by both these inference classes is very similar. The COSMOS inferencing operates on C++ objects by firing member methods of the object generated. For example **get\_value(attrname,attrtype)** is a message sent to an object to retrieve a *string* representation of the attribute specified. The returned type of the attribute is then used by the inference engines to perform appropriate pattern matches. This utilizes two C++ concepts: encapsulation and polymorphism. Only individual objects know about the value and type of



## 7.1 Overall Implementation Framework

---

their attributes, and convert them to a standard representation. The inference engines thus have a uniform interface to the individual objects. The interface to the inference engines is a set of simple methods which instruct the inference engines to parse rule files, construct inference nets, load objects and run to perform the inferencing. The COSMOS rule format allows arbitrary expressions in the conditions, and allows any of several consequent actions: make an instance, modify an instance, execute a method, fire another chainer (backward or forward), print, display images, etc.

CONGEN retains most of the user interfaces and functionality of COSMOS. This includes the various high-level user facilities: to create, browse and edit classes, to examine generated code, to create and directly edit instances, and to create and edit rules.

### 7.1.2 GNOMES

GNOMES is a non-manifold geometric modeler which has also been integrated with CONGEN. The main classes involved in the interactions of any application program with GNOMES are shown in Figure 2-2. The facilities to directly create/modify geometric models are accessed through the **GNmanager** class. The **GNmodel** class represents a general geometric model, which may further be a **GNcomplex** or a **GNassembly**. The application program which interacts with GNOMES may directly create a general model, and add vertices, edges, faces or volumes to the model. GNOMES also provides a set of basic solid modeling primitives: arcs and lines, rectangles and circles, cuboid, sphere, cone and cylinder. Arbitrary geometric models may be created by Boolean operations performed using combinations of these primitives. These Boolean operators include union, intersection and difference.

The GNOMES user interface (GRAPHITI) also provides a complete set of editing facilities: these include viewing transformations (zoom, rotate and translate camera, set mode of display (wireframe, solid, etc.)), select and pick objects, edit objects, translations, rotations, Boolean operations and geometric queries (distance, volume, area).

GNOMES is completely integrated with CONGEN. All geometric objects are created on the database along with the other application data. This integration allows the GNOMES objects to be directly manipulated by different modules, and also by the user.

## 7.2 Modeling Process Information

### 7.2 Modeling Process Information

A high-level class diagram for the overall CONGEN implementation is shown in Figure 7-1; class declarations for some of the important classes are listed in A. One of the

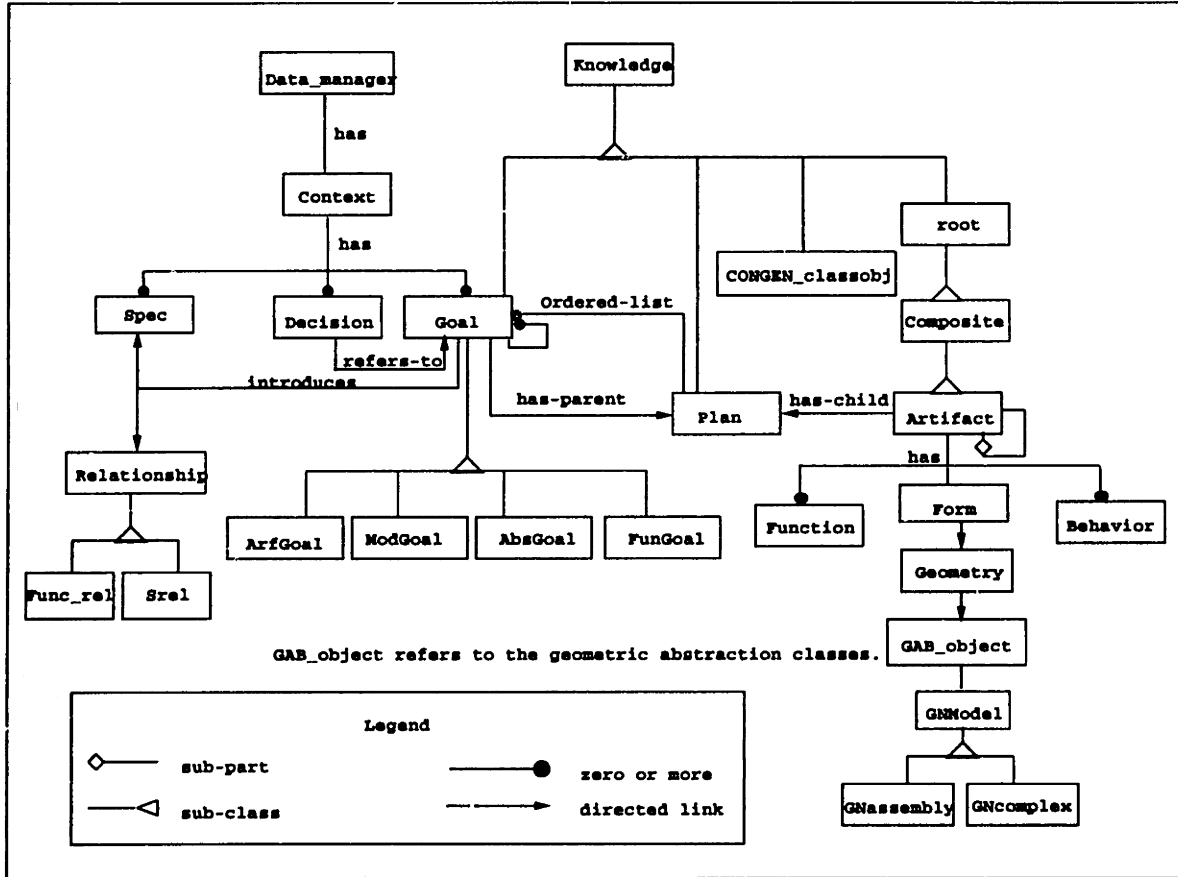


Figure 7-1: CONGEN class hierarchy

important classes representing the information generated during the course of the design process is the class **Context**. A new design **Context** corresponding to a new alternative is created at each decision point during design. Each **Context** consists of all the product information (artifacts, design relationships, decisions, specifications) generated during the design. All retrieval for specifications, artifacts and relationships must be done through the **Context**: some of this information may be duplicated across contexts. The context browser allows the user to examine the design being currently pursued: all decisions, artifacts, etc. are shown in the context tree (Figure 7-2). Clicking on any one of the buttons shown in

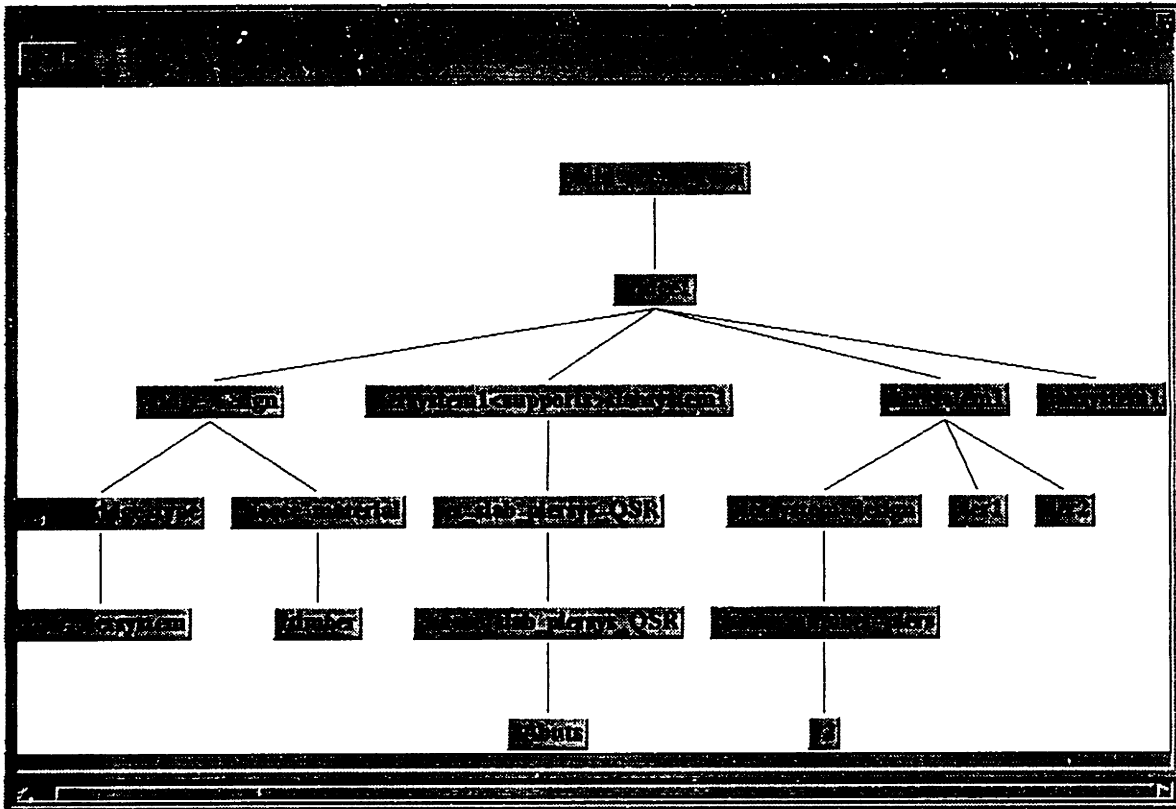


Figure 7-2: Context tree showing a particular design alternative.

the tree brings up the corresponding editor. At any point in the design, the designer may switch from one alternative to another by choosing from a list of contexts.

The class **Goal** represents a design task, which may also be a decision point. When a goal is *expanded*, the choices associated with the goal are checked for validity. This check is conducted by firing a backward chainer on each of the choices listed in the goal. The current context conditions are used to prune the choice. All the valid choices are presented to the user. When the user picks a particular alternative, a new **Context** is created. The effects of the decision are also then asserted by (a) setting the choice for the goal; and (b) firing a forward-chainer on the rulebase defining the *Consequences* of asserting a decision for the goal. Four subclasses have been defined for **Goal**: **AbsGoal**, **ModGoal**, **ArfGoal**, and **FunGoal**, as shown in Figure 7-1. These four subclasses refer to the important types of tasks that a **Goal** may represent: expand the process, modify an artifact (setting an attribute of a particular class), create a new artifact, or invoke a sub-

## 7.2 Modeling Process Information

---

function. The *pursue\_choice* method in the **Goal** class is refined by each of the subclasses. The editor for the **Goal** class is shown in Figure 7-3. The class **Plan** represents an

The screenshot shows a graphical user interface for editing a goal. It contains several labeled input fields and a choice list. The fields are as follows:

Name	choose_bridge_type
Rulebase	bridge_type.rul
Consequences	bridge_type.effects.ru
ParentPlan	bridge_design
Goal intended to:	MODIFY_ARTIFACT
Choice List	suspension, slab_piersystem
Artifact to modify: Classname	bridge
Slotname	type

Figure 7-3: Goal editor

ordered sequence of design tasks as a list of **Goals**. Rulebases may be associated with **Plan** to reorder the sequence of tasks. When a plan is associated with an **Artifact**, it constitutes the link from the product to the process hierarchy.

The class **Decision** refers to the decision made for a goal. Each **Decision** object not only stores the choice made at a point in the design process, but also all the alternatives

### 7.3 Modeling Product Information

---

which were available at the time. Each new decision spurs a new **Context**. The class **SpecFrame** contains all the specifications (represented by class **Spec**) for a particular design alternative. The specifications are values for attributes of objects, as shown in Figure 7-4. The specification may be either *instance-level*: refer to a specific instance, or *class-level*: refer to all instances of the class. New instances of **Spec** are also created whenever the user modifies an object directly by opening an instance editor. This allows the system to record all the direct design decisions and input made by the user.

### 7.3 Modeling Product Information

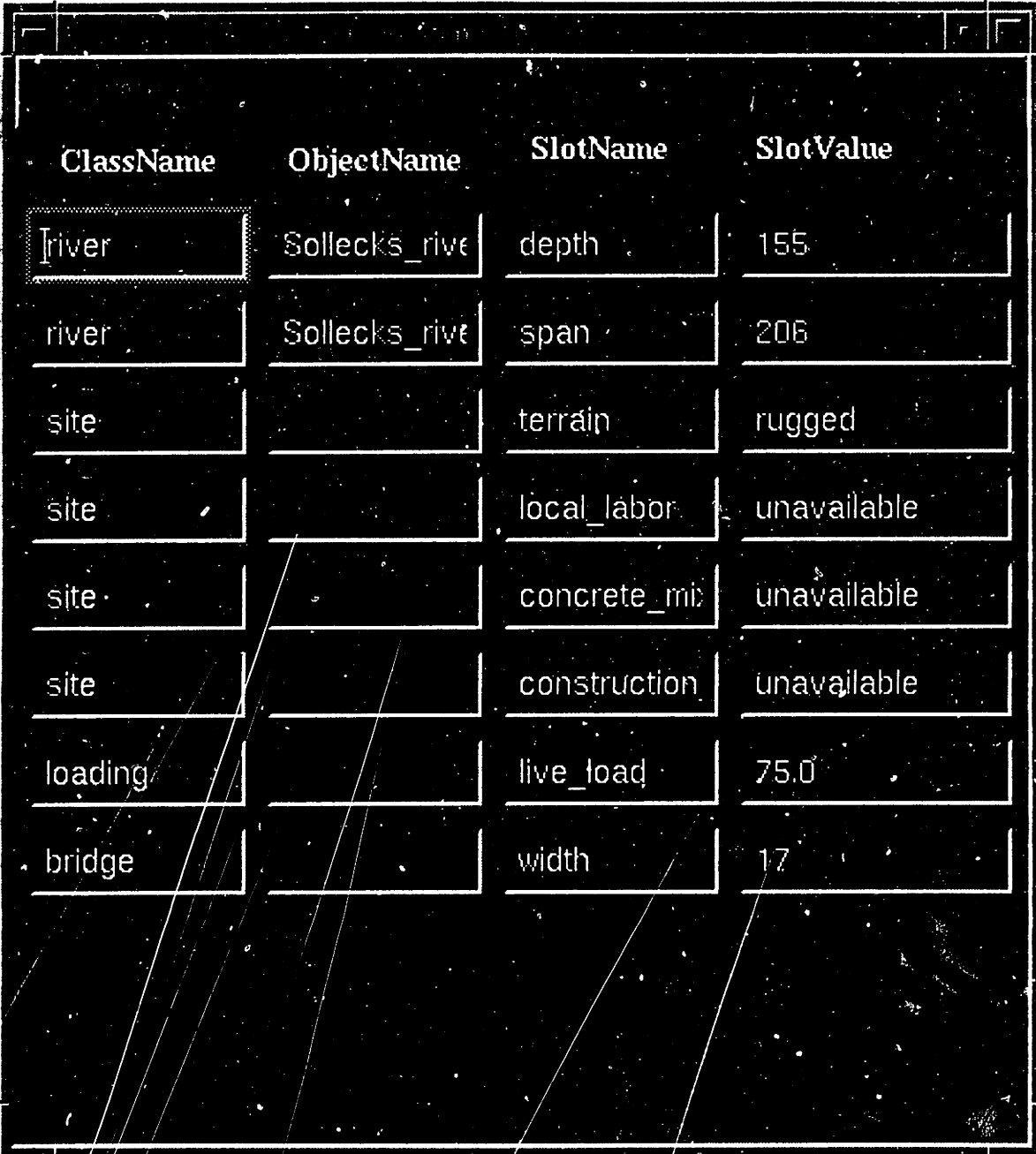
The main class representing product information is the class **Artifact**. The inheritance hierarchy for **Artifact** is shown in Figure 7-1. The class **Knowledge** is the base class for all knowledge defined in CONGEN (including the product and process knowledge). It merely provides a polymorphic interface for all the subclasses. It defines a method to *expand* the object. This refers to the disaggregation followed during the design process: each **Goal**, **Plan** and **Artifact** refines this method further. The **Knowledge** class also declares methods to display the object in various widget trees used in the user interface.

The class **root** is the base class defined in **COSMOS**. It provides the **COSMOS** interface: *get\_value* and *put\_value* methods allow attributes to be accessed by name. **root** stores the user-defined attributes in a list. It also defines an *invoke\_method* which allows member methods to be invoked by name. The class **Composite** provides composite functionalities for a design object. The **Composite** class stores a list of composite relationship objects, which may be relationships with the parts, or with parents in the composite hierarchy. The **Comp.rel** object defines the semantics of the composition relationship (e.g., whether an object is to be deleted if the parent is deleted, etc.).

**Artifact** is the base class for all user-defined classes. An **Artifact** object has a reference to the geometry of the object, and also has a link (a **Plan**) to the process hierarchy. An **Artifact** may be designated as *shared* across design alternatives (i.e., All design **Contexts** refer to the same **Artifact** object.). Any **Artifact** which is not shared is duplicated every time a new **Context** is created. This allows **Context** consistency to be maintained during a potentially iterative design process. The methods defined in the **Artifact** class define behavior which is common to all product elements: maintaining spatial and functional relationships, communicating with the geometry, and various display methods.

All user defined classes in CONGEN are derived from the class **Artifact**. The user

### 7.3 Modeling Product Information



ClassName	ObjectName	SlotName	SlotValue
river	Sollecks_rive	depth	155
river	Sollecks_rive	span	206
site		terrain	rugged
site		local_labor	unavailable
site		concrete_mix	unavailable
site		construction	unavailable
loading		live_load	75.0
bridge		width	17

Figure 7-4: Specification editor

### 7.3 Modeling Product Information

---

needs to merely specify the attribute names and types for a class. To associate behavior with the class, the user may associate rulebases and constraints with the defined class. The user may also associate a design **Plan** with subtasks for the design of an artifact. The parts of an object may be pre-specified for any user-defined class, if they are known at the time of class definition. These parts are treated exactly like other attributes, except that the corresponding objects are created and part links are set when the parent object is created. In the more general case when the parts are not known before-hand, they may be added through rules or directly by the user during the design process using the method `make-part` associated with the class **Artifact**. Figure 7-5 shows the user interface console for creating new classes and associating rulefiles with these classes. Sample generated

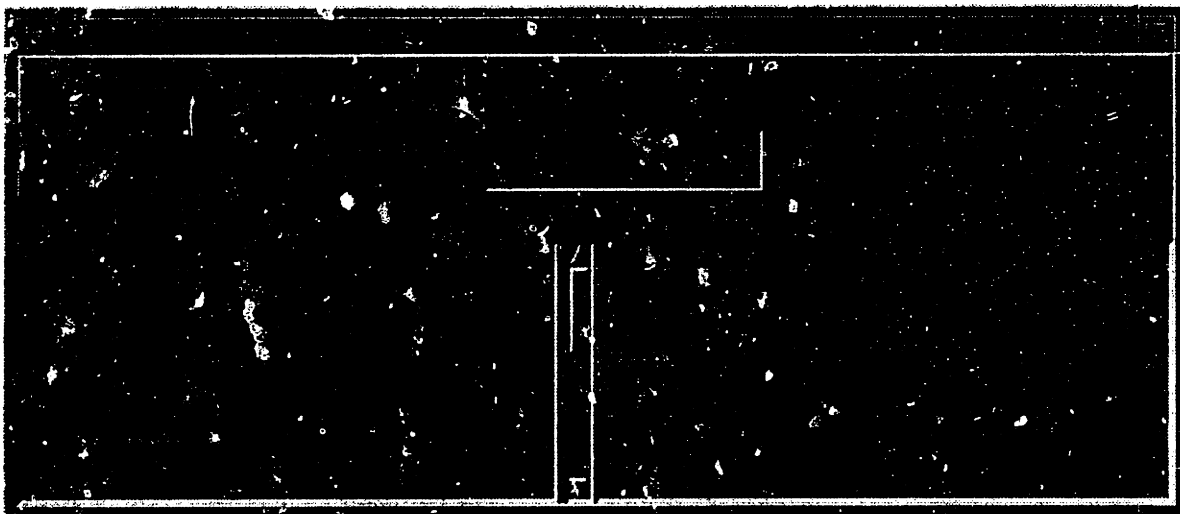


Figure 7-5: Product knowledge: COSMOS console

code for a class **bridge** is shown in Figure 7-6. The instance editor for this class is shown in Figure 7-7.

In CONGEN, all class-related information is stored inside a special class object **CONGEN\_classobj**. Each user-defined class has an instance of **CONGEN\_classobj** created to represent it. This class serves several purposes: it maintains all information about this class, it stores the *extent* of this class (all instances), it also stores all the specifications related to this class. It also serves as a placeholder for all information: specifications and constraints about instances which have not yet been created. e.g., The user may state as input the fact that the bridge width must be 17 feet. This may be done before any instance

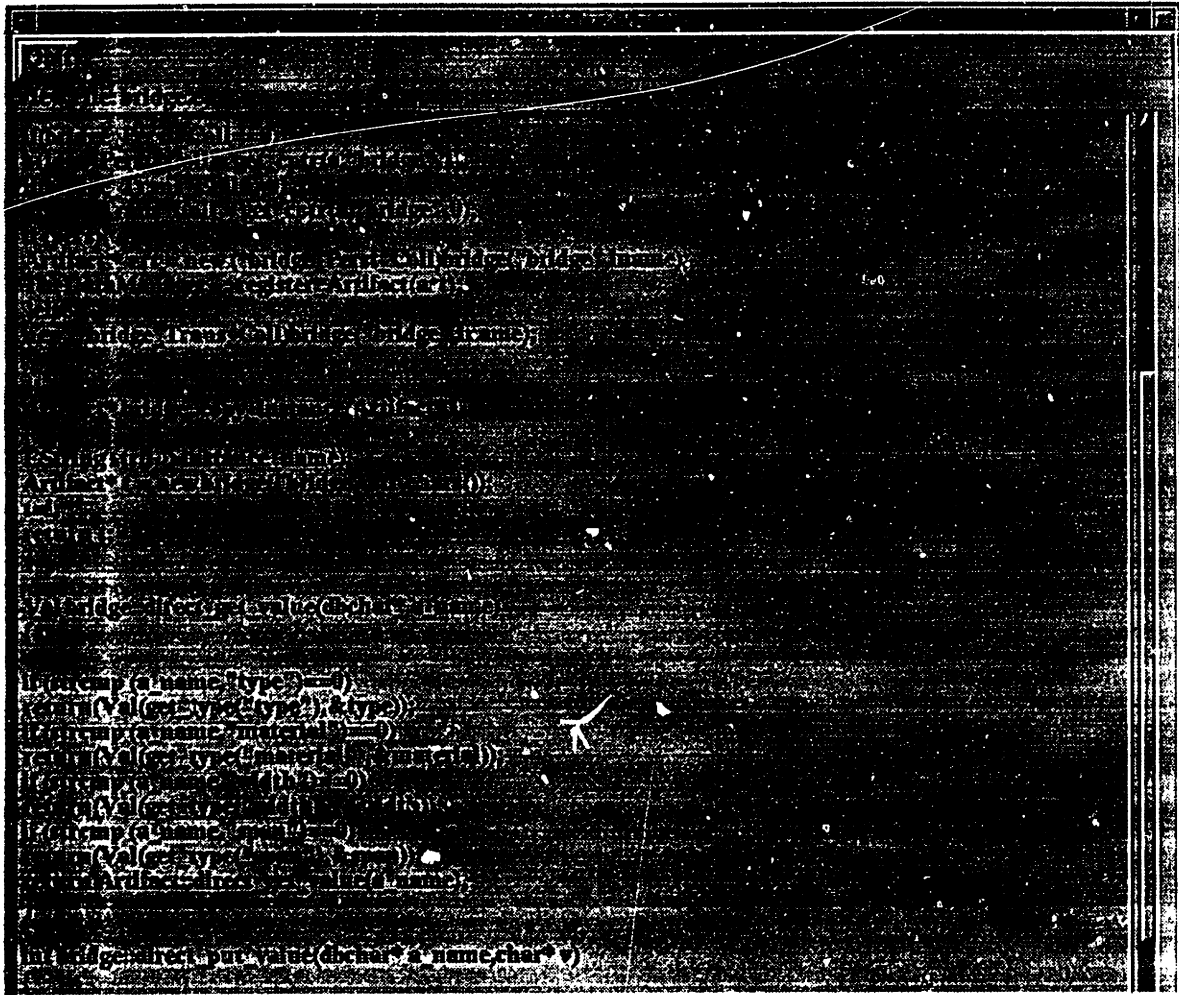


Figure 7-6: Sample generated code from COSMOS

of the bridge is created. The information is then stored in the placeholder and set when the instance of the class **bridge** is created. This class may also be used for reasoning by the different inference modules, when an instance is absent. The **CONGEN\_classobj** class serves one last purpose: it has convenience methods to allow attribute defaults to be set for all instances of a class, without having to name each of these separately.

#### 7.3.1 Defining Design Relationships

In addition to defining the **Artifact** and its subclasses, the product knowledge consists of the various relationship classes. These classes are also derived from **Knowledge**. The



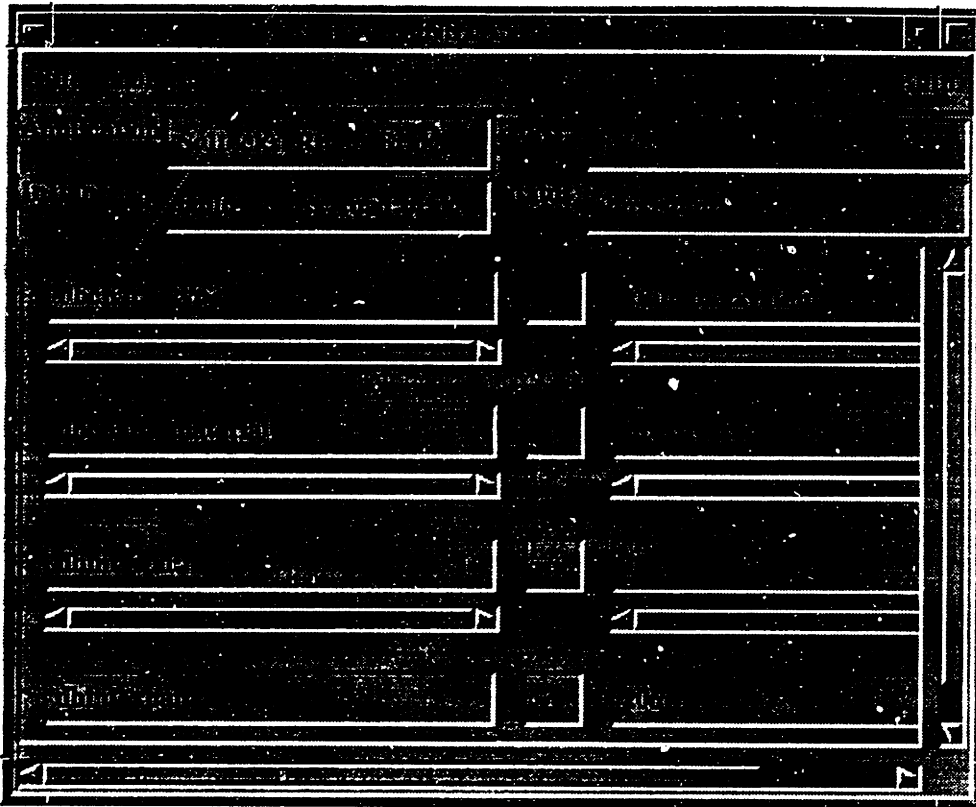


Figure 7-7: COSMOS Instance editor: can be invoked at any time during the design process

class **Relationship** is the base relationship class: it defines a **Role** for each of the objects involved in the relationship. The class **Func\_rel** defines a functional relationship. It refers to the function, the object which needs this function, and the object which provides it. The user may associate a **Plan** with the functional relationship. This allows the choosing of various spatial implementations of the functional relationship. The class **Srel** defines the base spatial relationship, and all associated classes are already explained in Chapter 5.

All functional and spatial relationships may be created through rules, or directly by the user. The corresponding functions invoked are *make\_func\_rel* and *make\_QSR*. Both these functions are invoked as strings: this implies that the arguments to the functions are also sent as strings. Type-checking for the arguments is currently quite primitive: a more robust technique which directly uses the C++ compiler facilities should be implemented eventually. The relationship palate which allows a user to choose spatial relationships is shown in Figure 7-8. On choosing one of these relationships, the corresponding editor is

### 7.3 Modeling Product Information

---

popped up: Figure 7-9 shows such an editor for the **Abuts** relationship. The palate does not represent a complete set of relationships; but it is easily extensible to define additional relationships.

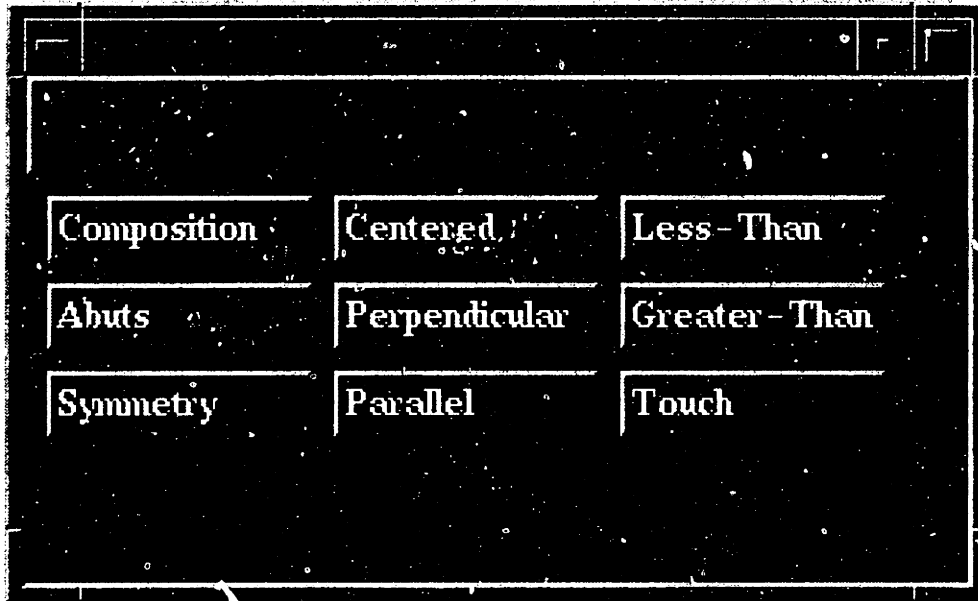


Figure 7-8: Palate for choosing spatial relationships

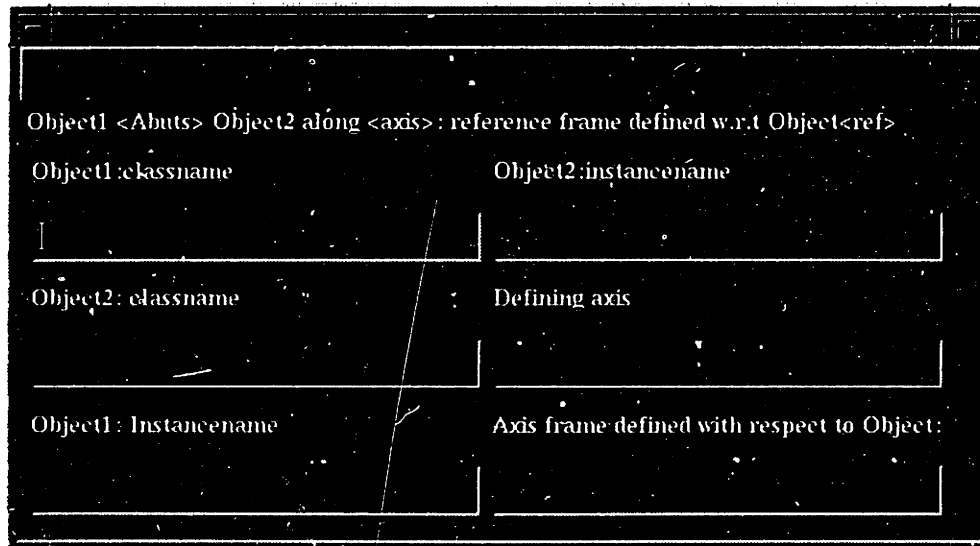


Figure 7-9: Abuts relationship editor

## 7.4 Modeling Geometric Information

---

### 7.4 Modeling Geometric Information

Every design artifact has a reference to its **Geometry**. The **Geometry** class is also derived from the COSMOS **root** class. This allows the inference engines to access and modify the geometry. The attributes stored in the **Geometry** class are the positions, orientations and bounding box dimensions for each design object. The class **Geometry** is a wrapper around the geometric abstractions, which may be evolving. All access to geometric information is through the **Geometry** of each artifact. This allows the actual geometric representation to evolve, while not disturbing any other module which might access the geometry. It is possible to set the geometric information through rules. This allows domain-specific defaults to be set for geometric information.

Also associated with **Geometry** are flags indicating whether a geometric attribute can subsequently be changed during constraint propagation. The **Geometry** class contains methods to create actual geometric models, update these geometric models (as discussed in the previous chapter), and to display the object.

The class **Engineering\_object** is the base class for all the shape abstractions in the geometric definition module (see Figure 6-1). It defines the virtual interface to access geometric attributes of objects by name. For example, an inference engine might set the flange thickness of a T-beam: by sending a message *set\_value("flange.thickness", "7.0")*. The **Engineering\_object** class also defines a virtual method *display* which displays the object. This method is redefined for each subclass. This method (a) checks to see if the object attributes necessary for the display are set; (b) attempts to compute these attributes if necessary; and (c) if they cannot be computed, defers the display to the parent in the hierarchy.

### 7.5 Constraint Representation and Satisfaction

Chapter 5 has already dealt with various implementation aspects of constraints, as necessitated by the discussion therein. All constraints within CONGEN are derived from the base class **Srel**(see Figure 7-1, Figure 4-5). Its subclasses include **QSR** and its derived classes (representing qualitative spatial relationships), **LT** and **GT** constraints (numerical restrictions). The current user interface implementation is limited in its ability to handle numerical constraints: a generalized parser will have to be eventually built. But the underlying constraint computation framework has the complete functionality required to handle

## 7.6 User Interface Components

---

arbitrary numerical constraints.

When the ATeams module is invoked, all the QSRs and numerical constraints relevant to the current context are sent in as constraints to the ATeams constraint satisfaction module. All the objects and variables referred to by these constraints are also retrieved. The constraint satisfaction module first creates a population of designs. To initialize a randomly generated design, the module first checks all the **Geometry** objects for each of the **Artifacts** involved in the constraints and QSRs. The **Geometry** object contains the defaults for the configuration variables for each object (sizes, positions and orientations). If these attributes have been denoted as fixed, all designs use the same values for these variables. The **Geometry** object may also specify ranges for the attributes. For any attributes which are not fixed, a random value in the range is then generated. When the entire population has been initialized, and the number of iterations has been chosen, the algorithm proceeds as described in Chapter 5.

## 7.6 User Interface Components

### 7.6.1 Main Console

The main CONGEN console is shown in Figure 7-10. The Console presents function-



Figure 7-10: CONGEN main console

alities to create/open new applications, edit knowledge, enter specifications, execute the application, and browse the design at any point.

Editing the knowledge consists of entering/modifying process and product knowledge. The process knowledge consists of defining a design process in terms of **Goals** and **Plans**.

## 7.6 User Interface Components

---

The product knowledge is defined by creating new classes to represent elements in the domain. The user interfaces and functionalities for each of these components have already been explained.

There are two main components to execute the application: the symbolic design is driven from the **Synthesizer** and the resulting geometry is displayed on the GNOMES graphical user interface, GRAPHITI. Each of these components is now discussed.

### 7.6.2 Synthesizer

The Synthesizer is the main execution module for the CONGEN system. This user interface is designed to provide the complete set of functionalities required to run an application. The system follows the step-wise hierarchical decomposition process defined by the **Goal - Plan - Artifact** constructs: the user begins the process by clicking on the root **Goal::Expand**. The corresponding method of the **Goal** class is then invoked. Subsequent decisions dictate the decomposition of the design. At any stage in the design process, the user may override the suggestions provided by the system, and directly provide input. This input may be in the form of a decision for a **Goal**, create an artifact, open up various editors (**Goal, Plan, Artifact, Specification**), create and add new relationships, or invoke the constraint propagation engine. Some aspects of this input constitute bottom-up knowledge: the user may even define new knowledge concepts (**Goal, Plan, Artifact**) and operate on this new knowledge. The user interface for the Synthesizer module is shown in Figure 7-11. When the user decides to run the constraint propagation, the **ATeamUI** console is invoked.

#### **ATeamUI**

The **ATeamUI** console user interface is a two-dimensional display, which displays the projection of the design on one of the display planes. This user interface is shown in Figure 7-12. It first asks the user for the number of iterations that the algorithm needs to be run before interruption, and also the display plane. After each set of iterations, the user may (a) examine the current best designs (ranked in order) and their evaluations with respect to each constraint; (b) decide to concentrate the effort on one or more of these designs; (c) continue the algorithm for another set of iterations; or (d) accept a design.

When the user accepts a design, the message is communicated to the the class **Geometry** described in section 7.4. The geometric model for the design objects is then computed

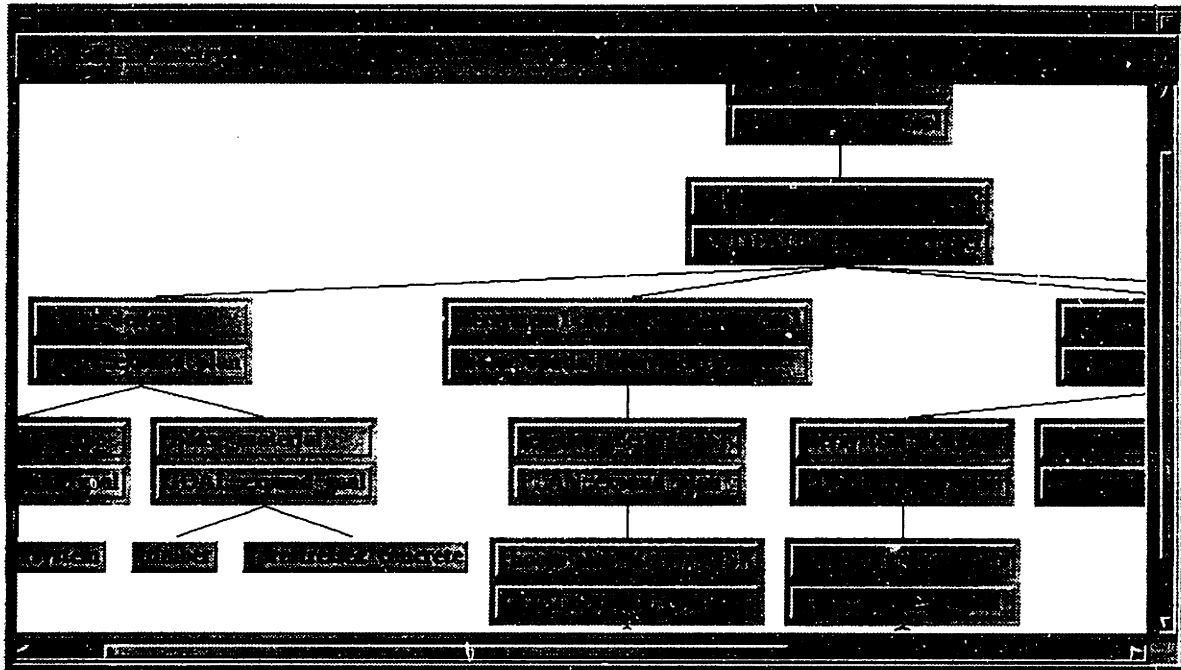


Figure 7-11: Synthesizer

and displayed on GRAPHITI.

### 7.6.3 GRAPHITI

The geometric modeler interface (GRAPHITI) may also be invoked from the main console at any point. The geometry of the evolving design is updated on the geometric modeler display as the design proceeds. Sometimes, displaying the geometry may need the constraint propagation to be run so that the alternative geometric configurations can be computed. The **Synthesizer** has an option: *display-geometry* which displays the geometry of the current design context.

GRAPHITI provides a complete set of geometric editing tools, which may be directly used by the user at any point in the design process. This user interface is shown in Figure 7-13.

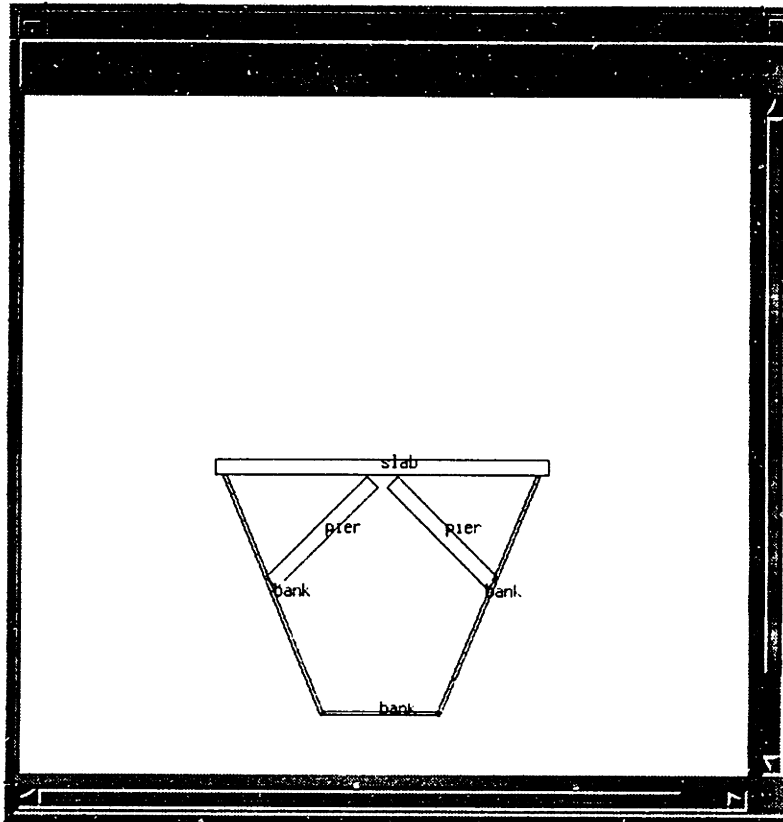


Figure 7-12: Ateams user interface

## 7.7 Summary

This chapter has given a brief description of the implementation and functionalities of CONGEN. These functionalities include knowledge editing and definition facilities (both process and product), various modes of specification, facilities to handle user decisions, and geometric modeling facilities.

The following chapter describes an example which is used to demonstrate these functionalities.



Figure 7-13: GNOME User interface



---

## Chapter 8

# Example and Results

This chapter describes a study of the overall framework, developed through a bridge design example. The steps in modeling the design products and the design process are introduced in sections 8.1 and 8.2 respectively. Section 8.3 runs through the design process, explaining various design decisions, user interactions with CONGEN, and the system behavior. It also provides a discussion of some of the results obtained during the example run-through.

The example is modeled after an actual bridge designed to span the 150 feet deep Sollecks river canyon in the Olympic Peninsula in northwest Washington [66]. The purpose of the bridge is to provide an access road across a river for logging operations in state-owned forests. The bridge is located about 85 miles from the city of Aberdeen, and the bridge site is accessible only by road. The terrain is extremely rugged and remote: neither construction facilities nor local labor are readily available. The bridge span is 200 feet, and the bridge width is specified to be 17 feet.

All the knowledge involved in modeling the design of this bridge is extracted from White et al.[66]. The focus of this chapter is to explain the functionality of CONGEN, using this real-world illustrative example. The following sections describe various elements of the implementation.

### 8.1 Modeling the Design Products

The first step in modeling the bridge design example is to create a new application: **Sollecks\_River\_Bridge**. Once the application is created, all the knowledge modules are activated: i.e., the user is allowed to start defining the product and process elements.

## 8.2 Modeling the Design Process

---

The following classes are developed for this example:

- (1) **loading**. This class models the loading conditions at the site: attributes are *live-load*, *wind-load*, *earthquake-load*, etc..
- (2) **site**. The site conditions of interest are *local-labor*, *concrete-mixing*, *construction-aids*, and *terrain-conditions*.
- (3) **bank**. The river bed is modeled by a series of banks. The attributes defining a bank are *rock-conditions*, *slope*, *length*.
- (4) **river**. This class defines the river bed (limited in this example definition to being piece-wise linear and having five banks as parts), depth and span of the river.
- (5) **bridge**. This is the top-level system and has attributes *span*, *width*, *material*, *type*. The parts of the bridge cannot be pre-defined unless the bridge type is known. It has an associated design plan **bridge\_design**.
- (6) **piersystem** defines a system of piers for the bridge, the no of piers determines the constituents of the piersystem. The piersystem also has a design plan **piersystem\_design**: to choose the number of piers and assert the effects.
- (7) **slabsystem** must at least consist of slab. Other components may be set during the design. It also has a design plan (**slabsystem\_design**) which consists of designing the slab, and setting support elements for the slab, if necessary.

Other classes modeled include some standard domain structural elements: **beam**, **column** and **slab**. These classes are just a representative sample: CONGEN provides the facility to model the domain elements in any other equally appropriate manner.

All the code generated for these classes is compiled into an *application* library, which is then dynamically linked with the rest of the system. In addition to the symbolic attributes, the example also models the geometry of the river canyon.

## 8.2 Modeling the Design Process

A few of the important elements of the process associated with the conceptual design are now presented.

### 8.3 Illustrating the Design Flow

---

- (1) The top level (root) goal is **build\_access\_road**.
- (2) The plan **bridge\_design** is associated with the class **Bridge**. This plan consists of two subgoals: **choose\_bridge\_type** and **choose\_bridge\_material**. Both of these goals define choices for attributes of the class **bridge** and validity conditions for these choices.
- (3) **choose\_bridge\_type** is a **ModGoal** whose purpose is to modify the *type* attribute of the bridge. Rulebases are defined both to prune the set of all choices listed, and to assert the effects of the decision made.
- (4) **choose\_bridge\_material** is a **ModGoal** whose purpose is to modify the attribute *material* of the bridge. Again, both the validity conditions and consequences are listed as rulebases.
- (5) **slabsystem\_design** and **piersystem\_design** are design plans which are used to design the slabsystem and piersystem respectively.
- (6) **choose\_number\_piers** is a subgoal in the piersystem design plan. The number of piers chosen has an enormous effect not only on the geometric configuration, but also on the behavioral properties of the slab and piersystem. We will demonstrate this during the example.

The root goal for the application is first specified. This creates the initial context in which all the input is created. The root goal may be specified by opening the process knowledge editor. The process also includes the initial designer input and specifications. In this example, an instance of **river** is first created to represent the **Sollecks\_river**. Depth and span attributes of this river are specified to be 155 and 206 feet respectively. Site conditions are also specified as input. This includes: *rugged* terrain, *concrete\_mixing* and *construction\_aids\_unavailable*. The desired width of the bridge is entered as 17 feet. The geometry of the river bed is also recorded as input, this determines important configuration properties of the bridge.

### 8.3 Illustrating the Design Flow

The design flow for the example is shown in Figure 8-1. The designer first invokes the **Synthesizer module**. On choosing to expand the goal **build\_access\_road**, the system

---

### 8.3 Illustrating the Design Flow

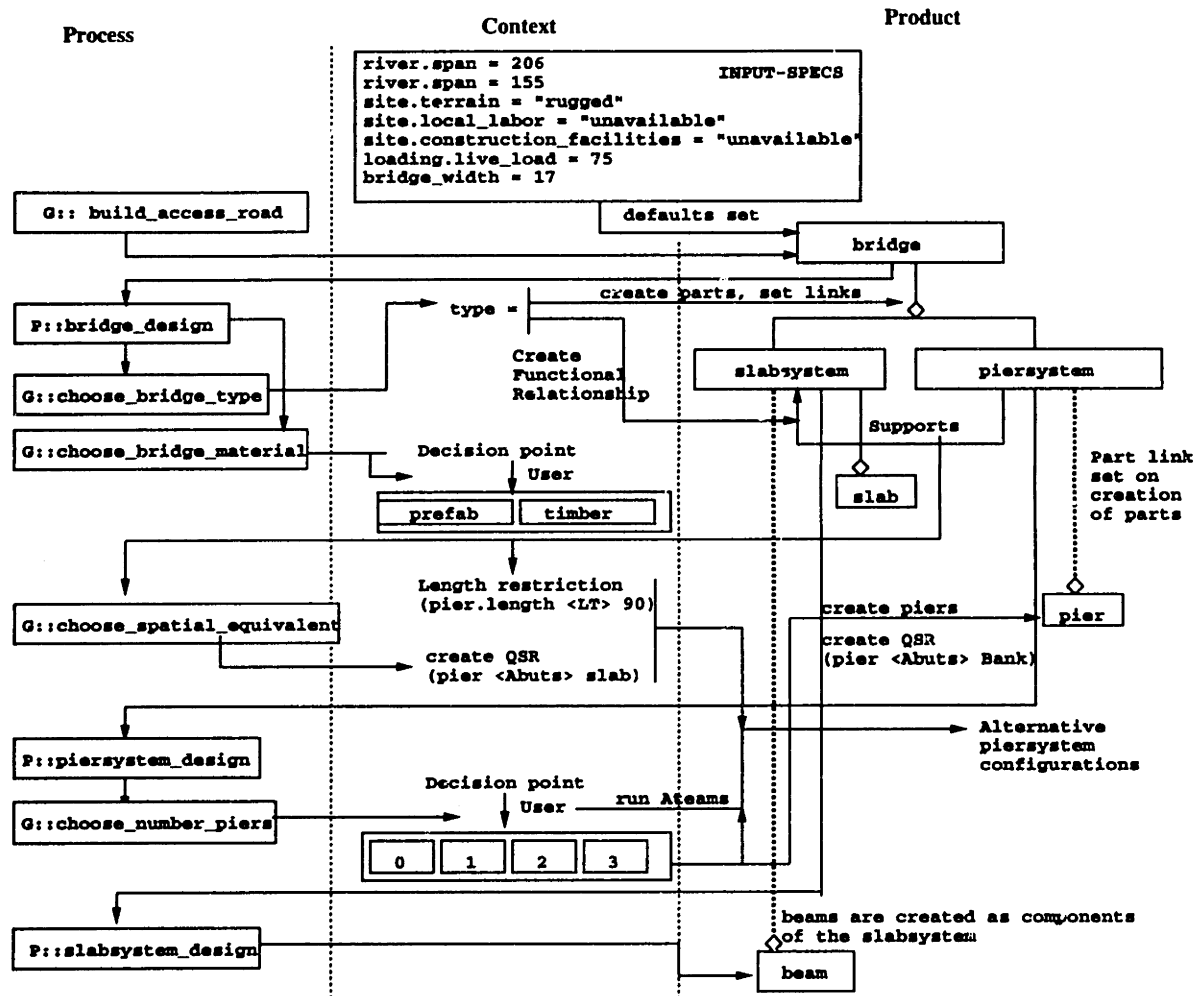


Figure 8-1: Design flow for the bridge example

instantiates the class **bridge**. At this point, all the specifications for the class **bridge** are set by the system. These include setting the span and the width of the bridge. Expanding this artifact brings into the design context all the design knowledge associated with the bridge: the components and the design plan. In this case, the design plan expands to two major tasks: choosing the type of bridge and choosing a material for the bridge.

Any of the design tasks and the knowledge associated with them can be examined at any point by directly clicking on the widget to bring up an editor. Expanding *choose-bridge-type* first, the only available choice at this stage is *slab-piersystem*, as shown in Figure 8-2. The *consequences* rulebase associated with this goal defines the knowledge needed to create a

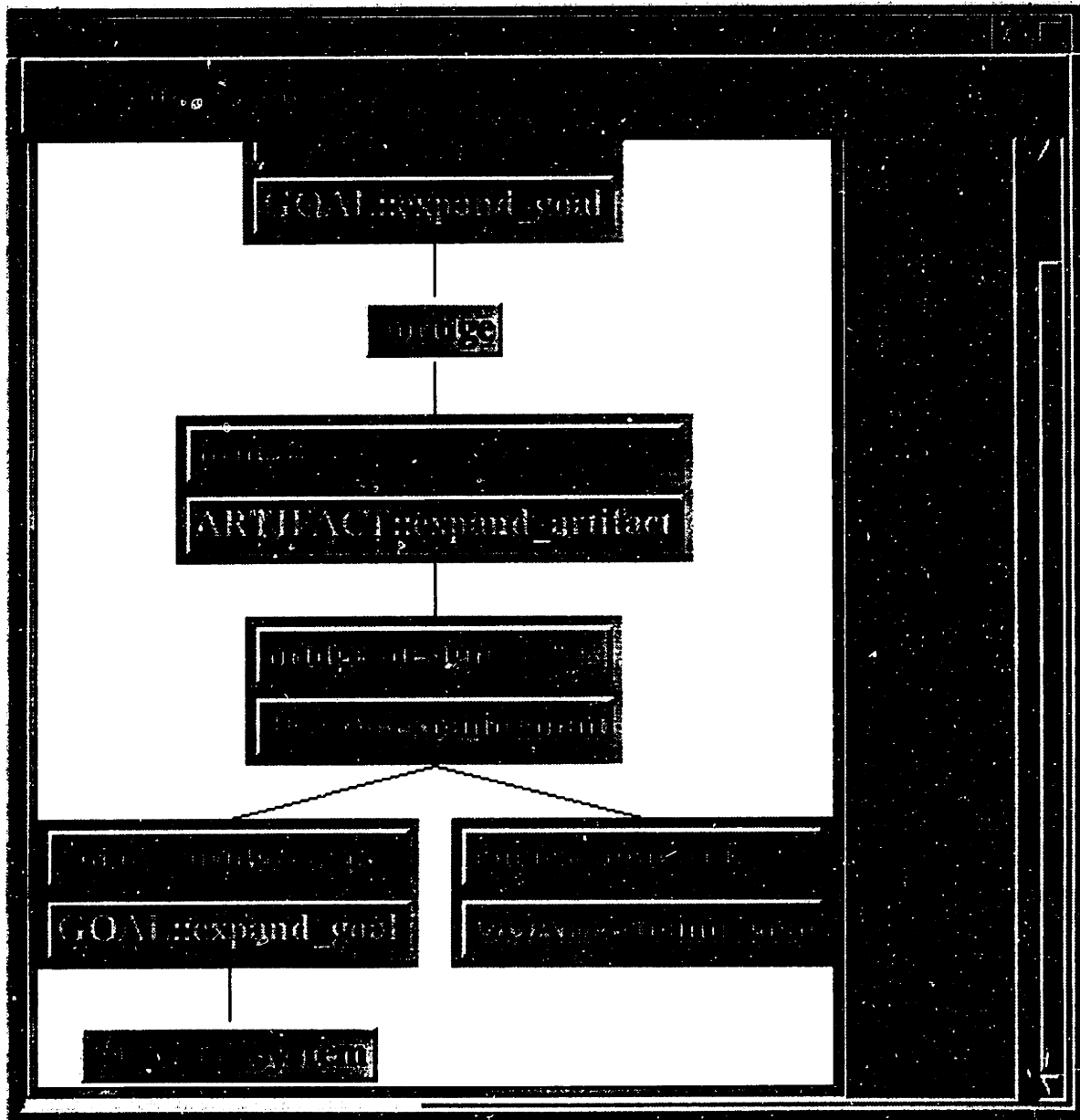


Figure 8-2: Expanding the bridge

### 8.3 Illustrating the Design Flow

---

*slab\_piersystem*. Choosing this option leads to invoking of this knowledge module. Thus the *slabsystem* and *piersystem* instances are first created. Also created by this knowledge is a *functional relationship* between the *slabsystem* and *piersystem*, as shown in Figure 8-3. The functional relationship knowledge, in turn, has a plan associated with it: to choose a spatial equivalent. This is shown in Figure 8-1. The geometry of the design alternative is

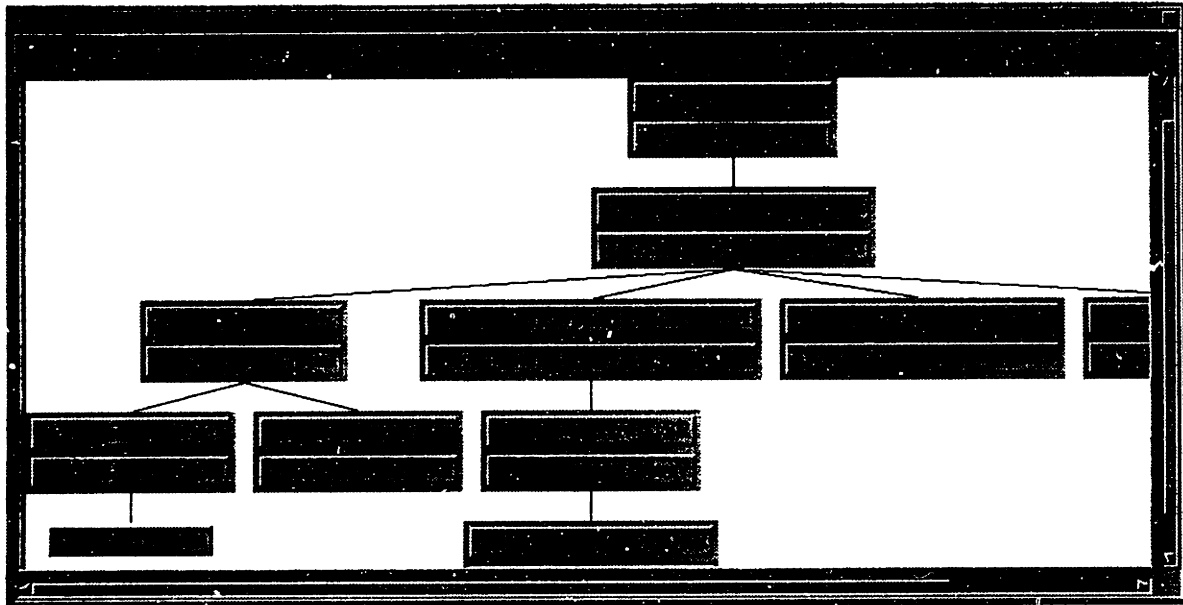


Figure 8-3: Functional relationship is created between the *slabsystem* and *piersystem*.

now reflected in the geometric modeler, shown in Figure 8-4. The bridge at this stage is very incompletely defined: the only known component is the slab. The length and width of the slab have been computed during the design expansion and are the only attributes known at this point, but this information is enough for a minimal display of the slab as shown in Figure 8-4.

Choosing a material has important implications for the structural configuration of the bridge. As already noted, the terrain is rugged and construction facilities are not available. The only materials which are found as feasible choices are *timber* and *prefabricated prestressed concrete*. But the terrain conditions place important restrictions on the length of the prefabricated structural elements that can be transported to site: the roads to the site are heavily winding, and elements longer than 90 feet cannot be transported to the site by truck. The effects of choosing a decision for material are thus to introduce this constraint

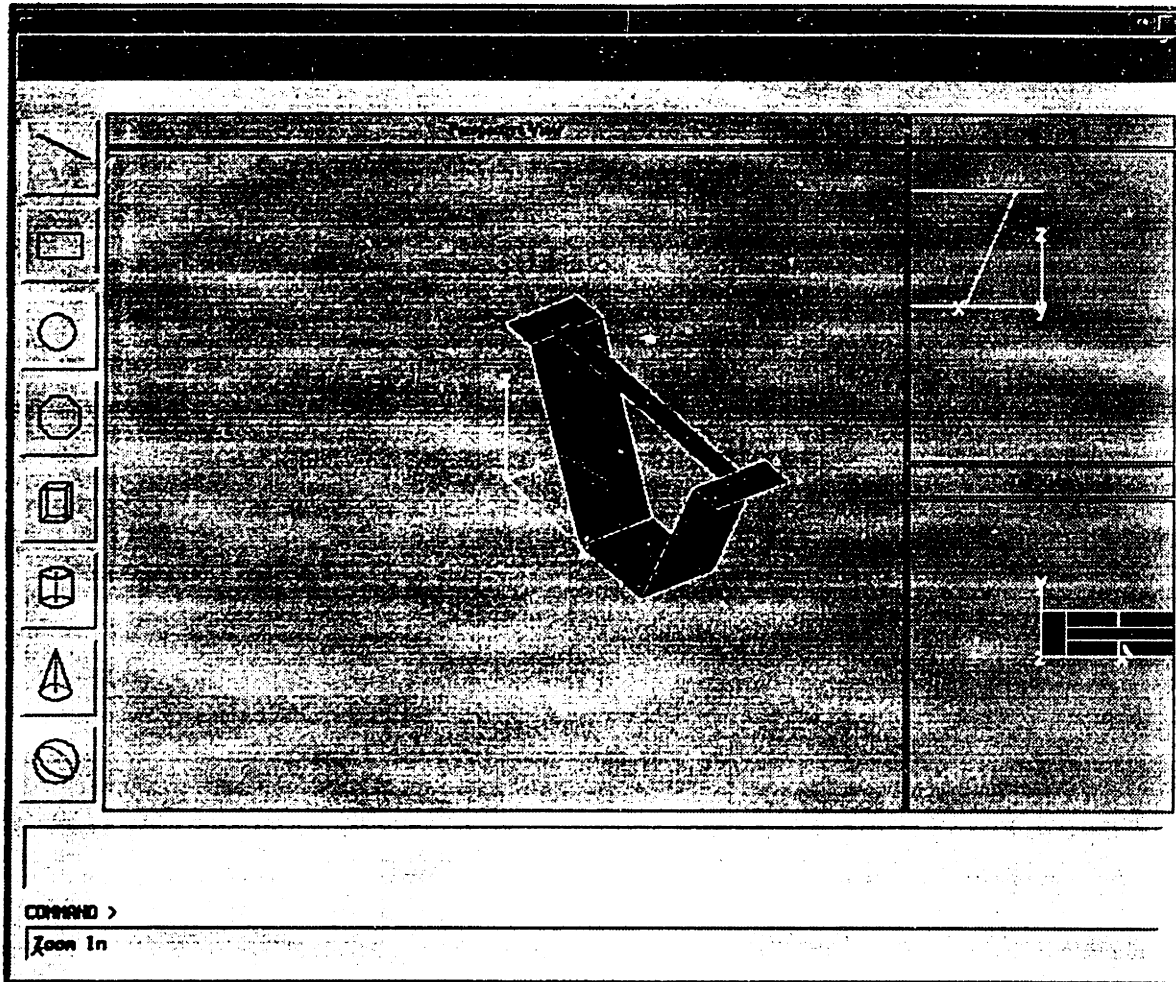


Figure S-4: Bridge geometry: Only the slab geometry is known at this point.

on the lengths of the piers, as shown in Figure S-1.

#### 8.3.1 One Pier Design

Continuing with the design, the spatial equivalent of the *supports* relationship is chosen as the **Abuts** QSR. When the **piersystem** is now expanded, the system prompts the user to choose the number of piers for the piersystem. If the designer chooses one pier, the system propagates the effects of this decision. This propagation includes creating an instance of class **pier** and setting it to be a part of the **piersystem**. An additional effect that is now asserted is a warning that the resulting span length of the slab may be too high

### 8.3 Illustrating the Design Flow

---

(Figure 8-5). Thus constraint checking may be handled either through a distributed rule

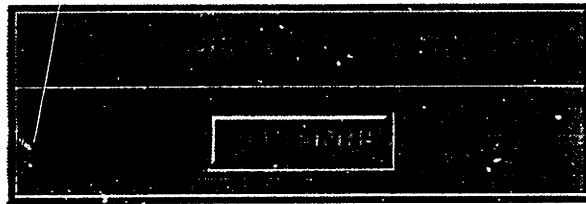


Figure 8-5: Constraint violation notification

mechanism, or by checking the list of constraints currently maintained in the system. An additional constraint violation that would result from deciding to continue with this choice would be due to the resulting length of the pier. This is detected only when the geometric attributes are computed, however.

#### 8.3.2 Two Pier Design

Assuming the designer heeds the warning and decides to choose a two pier design instead, the system would then create a new design alternative corresponding to this choice. At this point, the piers are created and part links are set. Also instantiated are the length restrictions on the piers, and the *< Abuts >* relationship between the piers and the slab. Further, the function of the pier is to transfer load to the bank. Hence the relationship *pier < Abuts > Bank* is also created. Note that this last relationship is a disjunction, since the bank it refers to depends on where the pier is placed. All these effects are shown in Figure 8-1.

Deciding on the configuration for the two-pier system is another major structural and form decision. To proceed with the design, the designer may now choose to compute different alternatives. This is done by invoking the ATeams module. This performs the following actions:

- *Loads the constraints.* The constraints involved at this stage are the following:
  - (1) **Pier1 Abuts Slab1.** This constraint is due to the *supports* relationships between the pier and the slab.
  - (2) **Pier2 Abuts Slab1.** This constraint also arises from the *supports* relationships between the pier and the slab.



### 8.3 Illustrating the Design Flow

---

- (3) **Pier1 Abuts (Bank1 or Bank2 or Bank3)**. This constraint arises from the function of the piersystem, to *transfer load* to the foundation.
- (4) **Pier2 Abuts (Bank1 or Bank2 or Bank3)**. This constraint arises from the need to *transfer load* to the foundation.
- (5) **Pier1.length LT 90.0**. This constraint arises from the choice of material and the resulting transportation problems.
- (6) **Pier2.length LT 90.0**. Similarly, this constraint results from transportation considerations.

Also modeled as a constraint is **Symmetry** of the piers about the center of the river bed. In general, symmetry is a design objective. This constraint need not really be enforced very strictly. The search for the algorithm is conducted with both the pier configurations treated as independent variables.

- *Load the design objects*. This step retrieves all the design objects involved in the relationships, and initializes all configuration parameters which are designated as variable. The population of designs for the ATeams algorithm is then created. In the example, all the **Bank** instances, and also the slab are treated as fixed. The variable parameters for the piers are the lengths of the piers, the X and Z positions of the piers, and the orientations of the two piers in the XZ plane. The system discretizes the parameter space for the configuration variables: 0 to 285 (285 discrete units) for the X positions, 0 to 155 (155 discrete units) for the Z positions, 0 to 142 (142 discrete units) for the lengths, and 18 possible values for each of the orientation parameters. Thus, the search space for the overall system is a total of  $1.28 \times 10^{16}$  possible spatial configurations.

The algorithm is run in sets of 3000 iterations. Each such set of iterations for this example takes about 1 minute (though improvement would certainly result from optimization of the code). At the end of the 3000 iterations, the user may examine the best designs which are currently in the design population. Note that these are not the “optimal” designs. Indeed, at this stage no design can confidently be asserted to be an “optimum”. Rather, the goal is to present alternative configurations which satisfy the constraints to varying degrees. Four of the principal variants found for the example are shown in Figures 8-6 to Figures 8-9. (Note that the actual search is conducted in three dimensions, the display only shows a cross-section in the XZ plane).

### 8.3 Illustrating the Design Flow

---

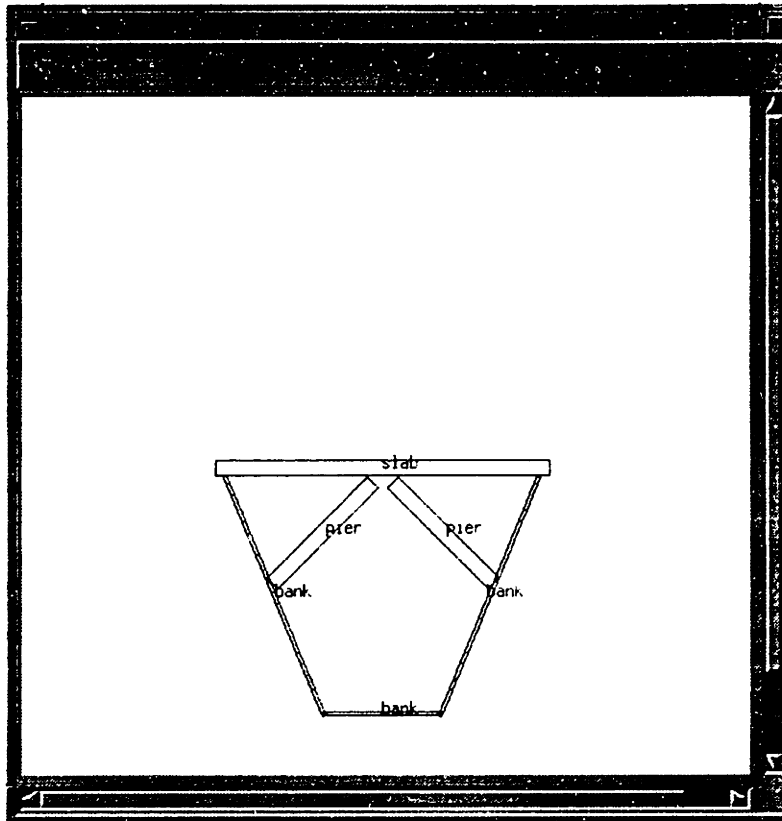


Figure S-6: Results: Alternative 1 (after 3000 iterations)

Each of the four variants shown “approximately” satisfies all the constraints (The user may examine the evaluations at this point). Thus each alternative represents an “approximate” design. This is an important facility from a conceptual design standpoint: the design is at too premature a stage to seek exactly “optimal” solutions. Yet, given this set of approximate solutions, the user may directly operate on any one of these alternatives to improve them. The user can directly instruct the program to apply improvement operators on a single design, instead of the more diffused process of the general algorithm (which operates on a population). Thus an approximate design can be very quickly improved to the desired level of accuracy.

Another interesting approach is to *monitor* the user interactions with the system at the end of each set of iterations. Thus, it is conceivable that the user could identify potentially interesting alternatives, and choose to concentrate the effort of the algorithm on these alternatives. In the same vein, apparently nonintuitive results (from a physical

### 8.3 Illustrating the Design Flow

---

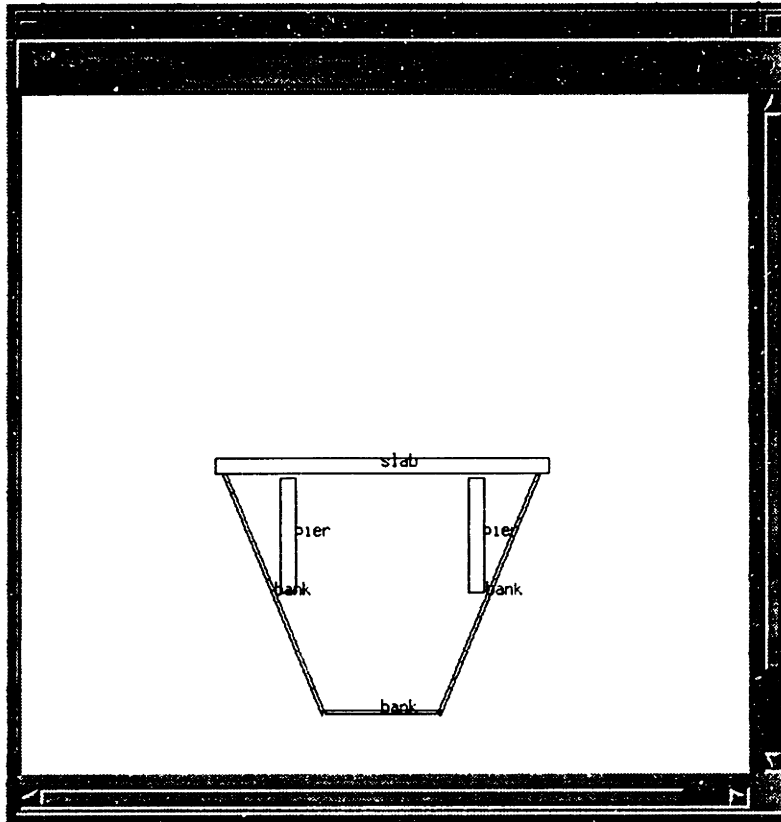


Figure 8-7: Results: Alternative 2 (after 3000 iterations)

perspective) could be put into a *delete list*, and all strains of such variants are wiped out from the population. Thus the user has direct control on the design population and the solutions that the algorithm is currently experimenting with. In the current implementation, when the user attempts to directly improve a particular design, this design is marked as being of apparent interest. To propagate these apparently interesting traits, a duplicate is thrown into the population for every set of improvements performed on the design (each set is 20 improvements). This has proved to be a very effective strategy which allows the designer to monitor and control the flow of the algorithm.

The algorithm is now run for another set of 3000 iterations. Again, the four main variants from above have been carried through for this set, and each of these designs now almost completely satisfies all the constraints. These are shown in Figures 8-10 to 8-13. The first design, shown in Figure 8-10, is the design which is actually carried through for the rest of the example described in this chapter.

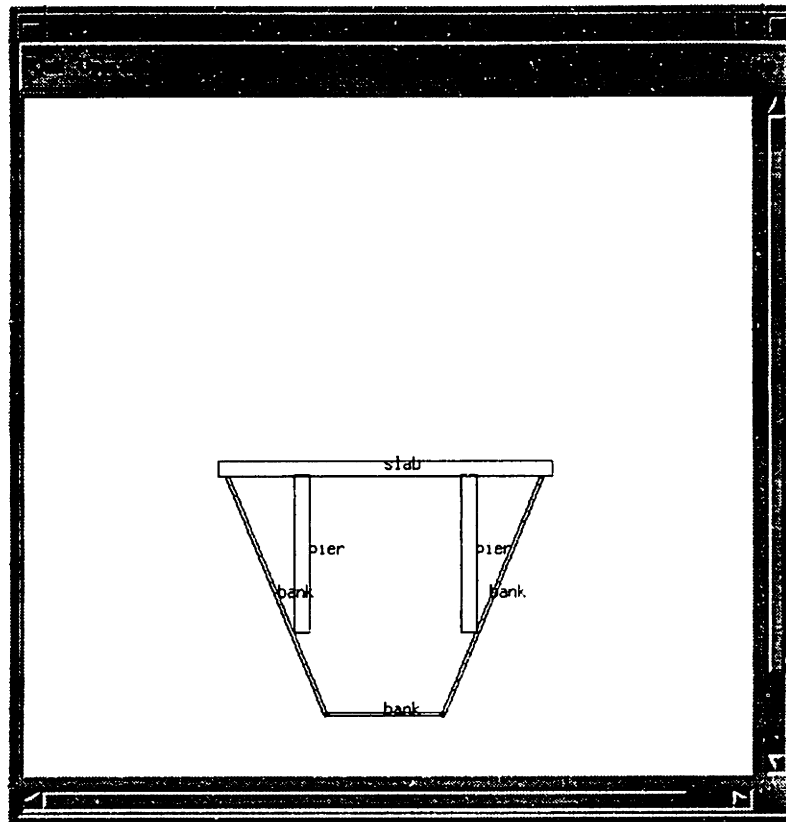


Figure 8-8: Results: Alternative 3 (after 3000 iterations)

#### Three pier design

Figure 8-13 shows a design which is definitely nonintuitive, and might certainly have already provoked the reader's interest. In the example, this design was retained for the following reason: While it may not be immediately obvious that it satisfies the constraints, careful examination reveals that the piers do indeed almost abut both the slab and one of the banks, albeit from the same end ! This reveals an interesting insight: the definition of the *Abuts* relationship merely specified the *touch – contact* in terms of the projections of the two objects. One way to improve this might be to redefine the *Abuts* relationship to also include the plane of contact.

An alternative way of perceiving the situation might be that such unexpected situations occasionally spark off creative thought in a designer. For example, observing that the two piers have a Y-shaped appearance, the designer may at this point thrown in a third pier.

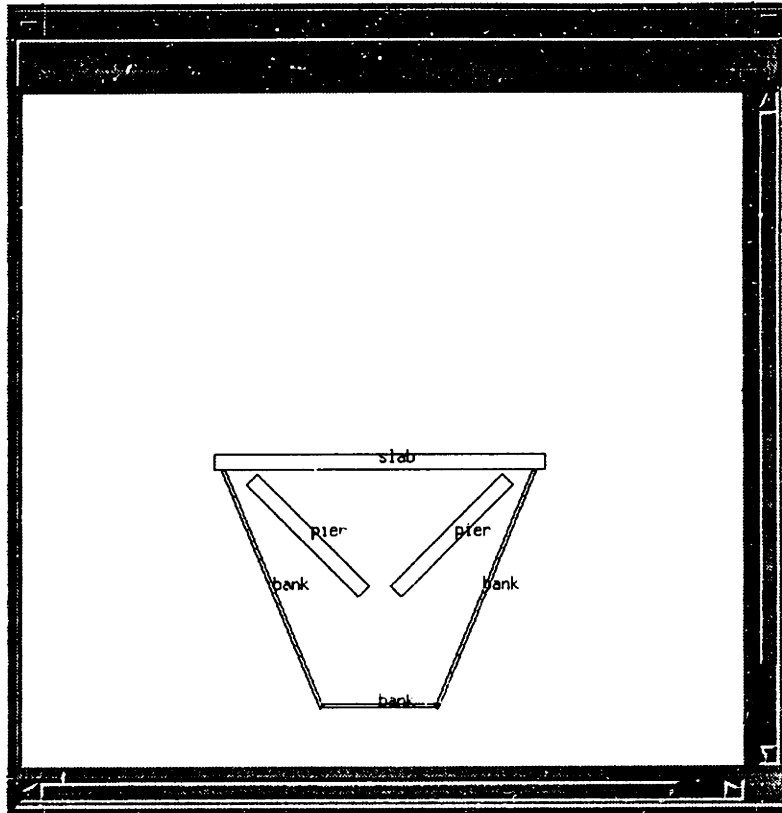


Figure 8-9: Results: Alternative 4 (after 3000 iterations)

Adding a new artifact to the population is simply a matter of adding a randomly initialized object to each design in the population. The designer may add new constraints as well: for example, **Pier1 Abuts Pier3** and **Pier2 Abuts Pier3**. All designs are re-evaluated with respect to this constraint. One of the configurations resulting from running the algorithm again is shown in Figure 8-14. The Y-shaped design implicitly relaxes two of the constraints (**Pier Abuts Bank**). Instead, this transfer of load now takes place onto the third pier.

#### 8.3.3 Design Refinement

This section presents the detailing of the alternative represented by Figure 8-10. The overall bridge system is now shown in Figure 8-15. Pursuing the design further, each of the piers is designed by firing its subplan. The span of the slab supported on each pier determines the number of columns in each pier: Figure 8-16 shows two columns for each of the piers. The geometry of the piers is propagated to the columns. This design process

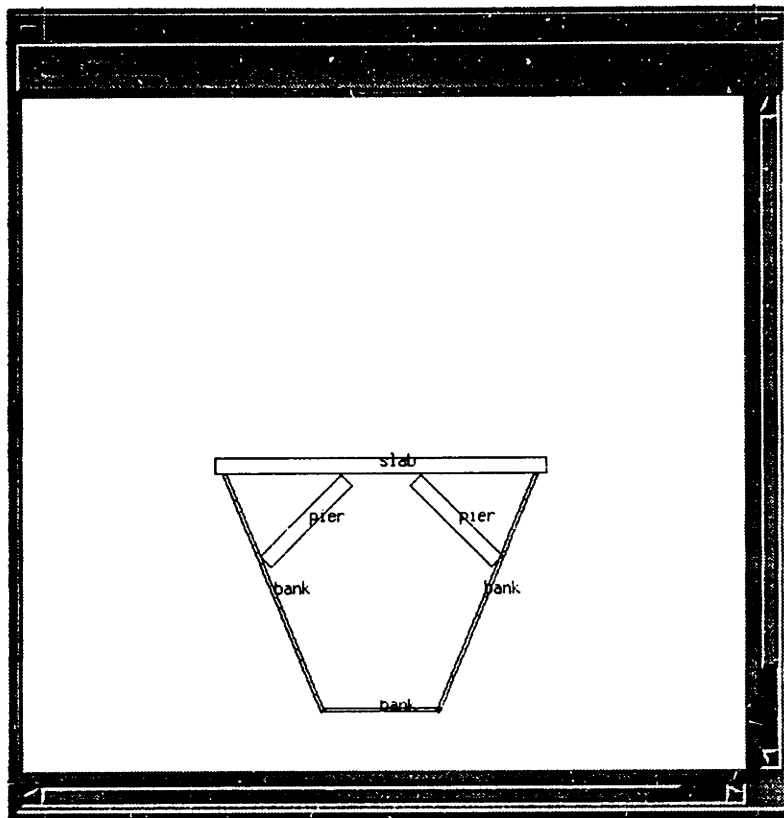


Figure 8-10: Results: Alternative 1 (after 6000 iterations)

moves on to design the slab system to also consist of two beams supporting the slab, and further determines the slab thickness. The resulting conceptual design, shown in Figure 8-17, is now ready for the detailed analysis and design stages; We are currently integrating an analysis package - GROWLTIGER - which was developed at MIT.

## 8.4 Summary

This chapter presented an example of bridge design, which demonstrated the overall functionality of CONGEN. The example illustrates the systematic symbolic decomposition process, and the generation of geometric alternatives at each stage in the design. The experiment demonstrates the main advantages of the approach presented in this thesis. The central idea is to allow the designer to explore with various form alternatives. The symbolic design affects the geometric configuration, and as shown, emergent geometric shapes may

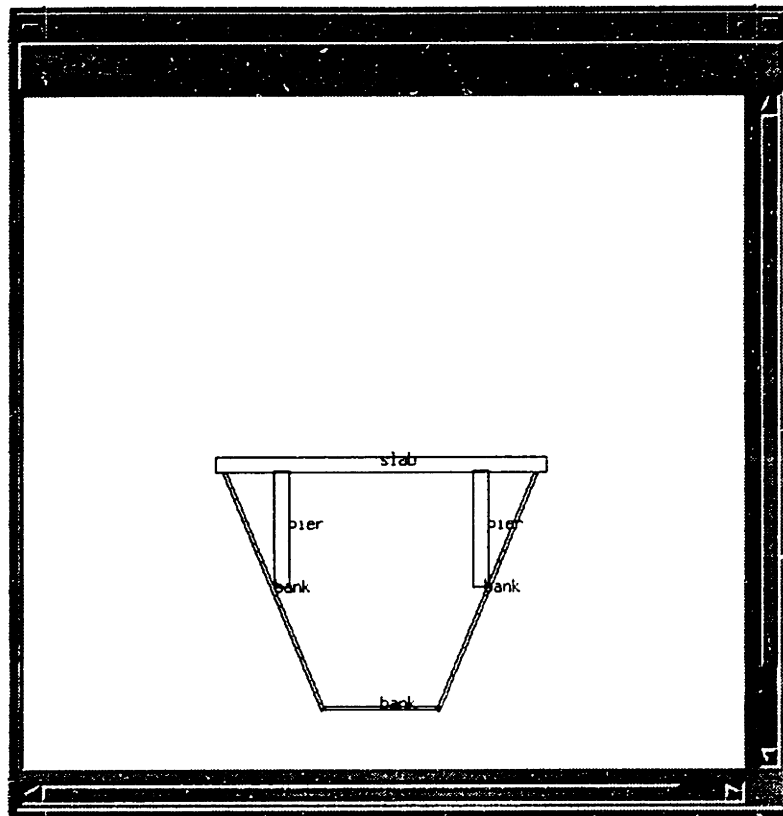


Figure 8-11: Results: Alternative 2 (after 6000 iterations)

be recognized by the user and sometimes effect the further symbolic evolution (In this example, the third pier is added). The system provides support for bottom-up design: new elements and relationships can be added at any point. The systematic decomposition of the form conception into the various elements captures the essential functional intent of the design. Importantly, it also encourages the innovative/creative design process, in terms of experimenting with design alternatives.

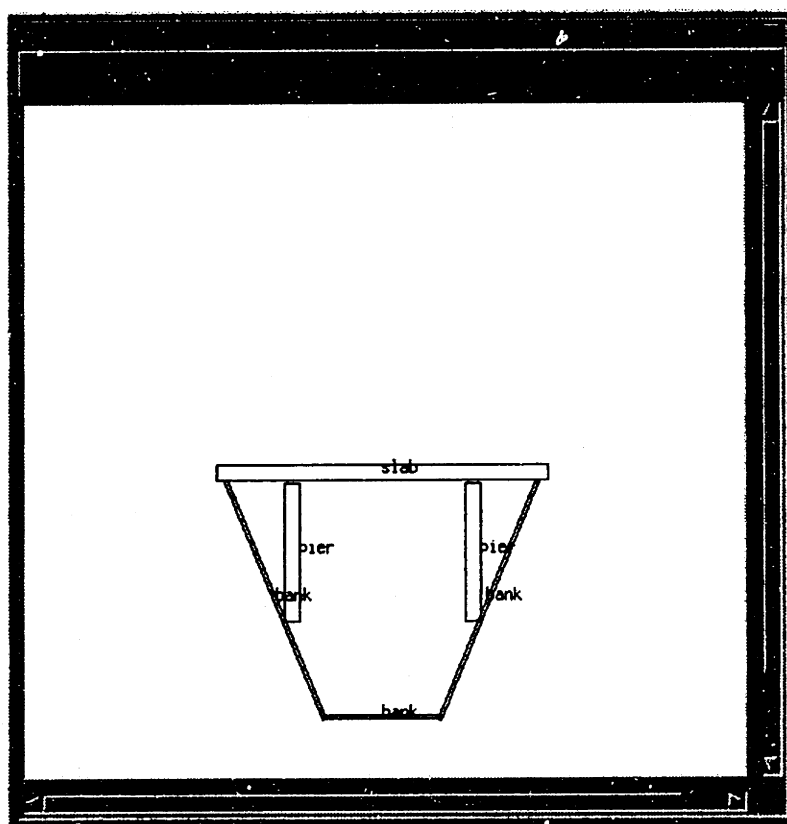


Figure 8-12: Results: Alternative 3 (after 6000 iterations)



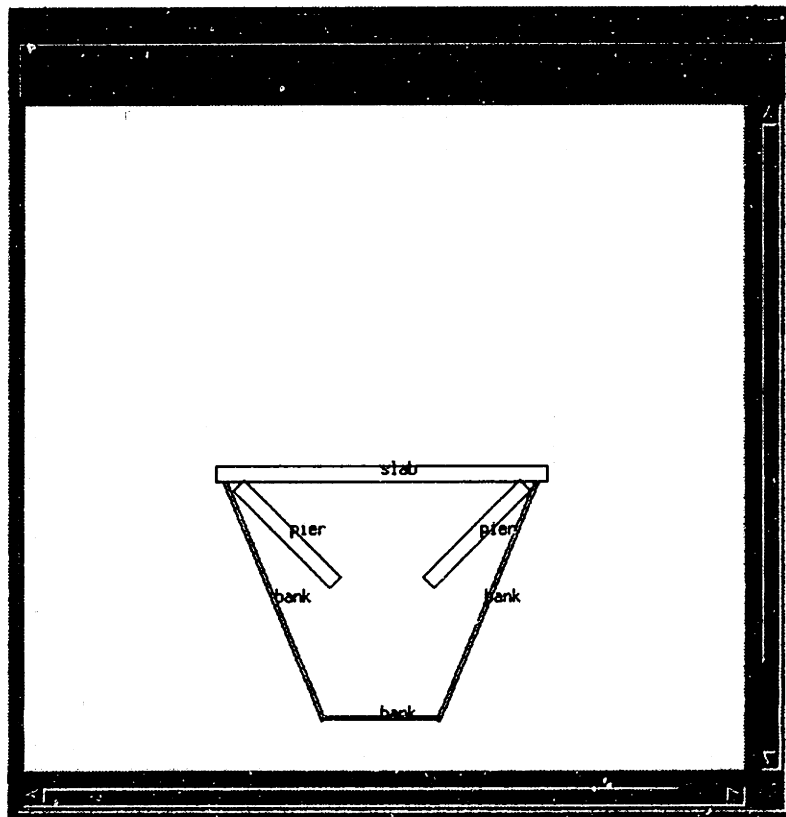


Figure 8-13: Results: Alternative 4 (after 6000 iterations)

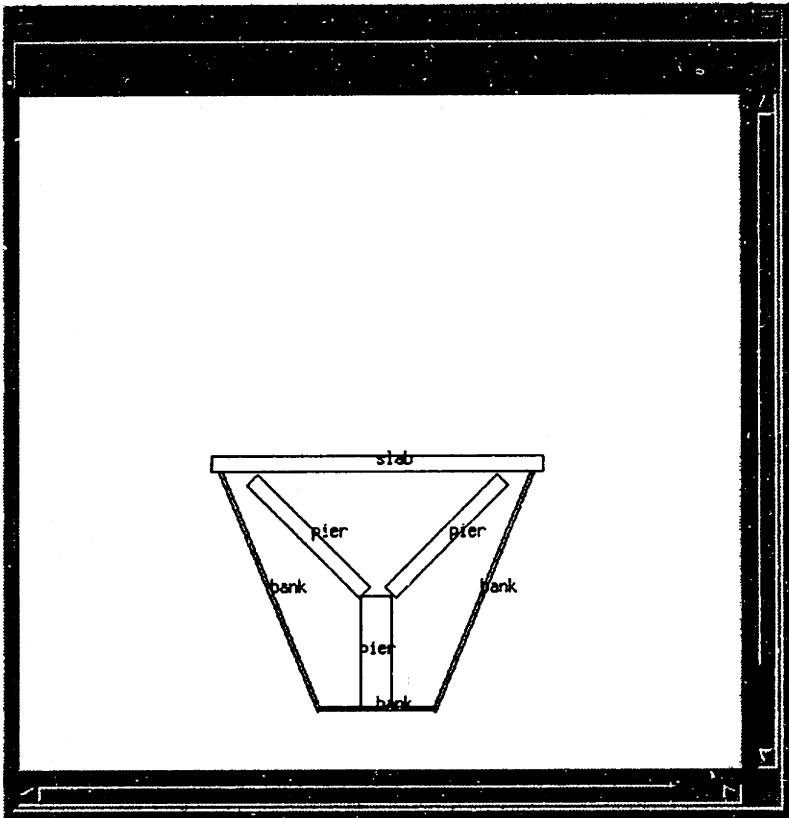


Figure 8-14: Y shaped emergent alternative

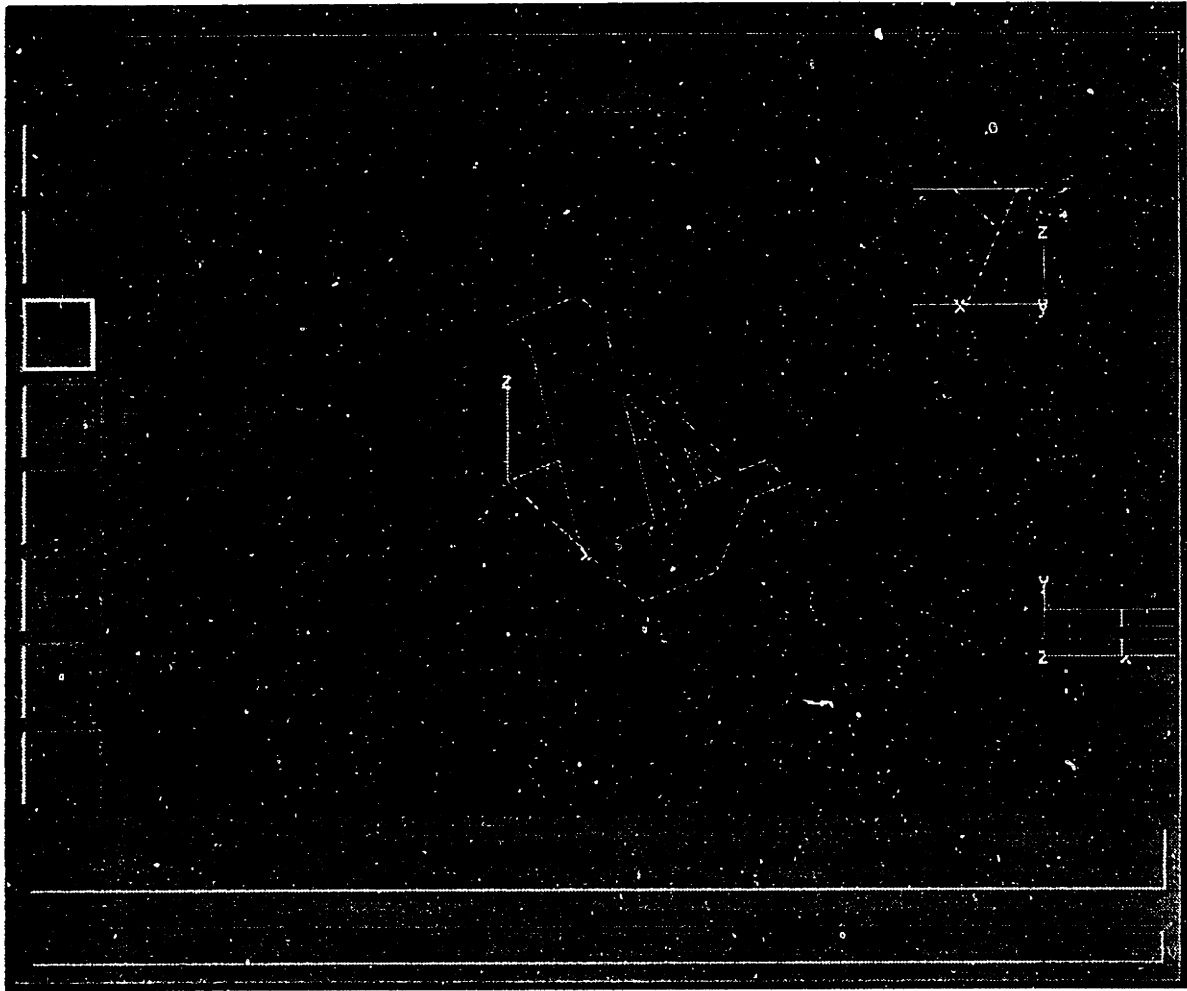


Figure 8-15: Bridge geometry: Piersystem alternative has been accepted.

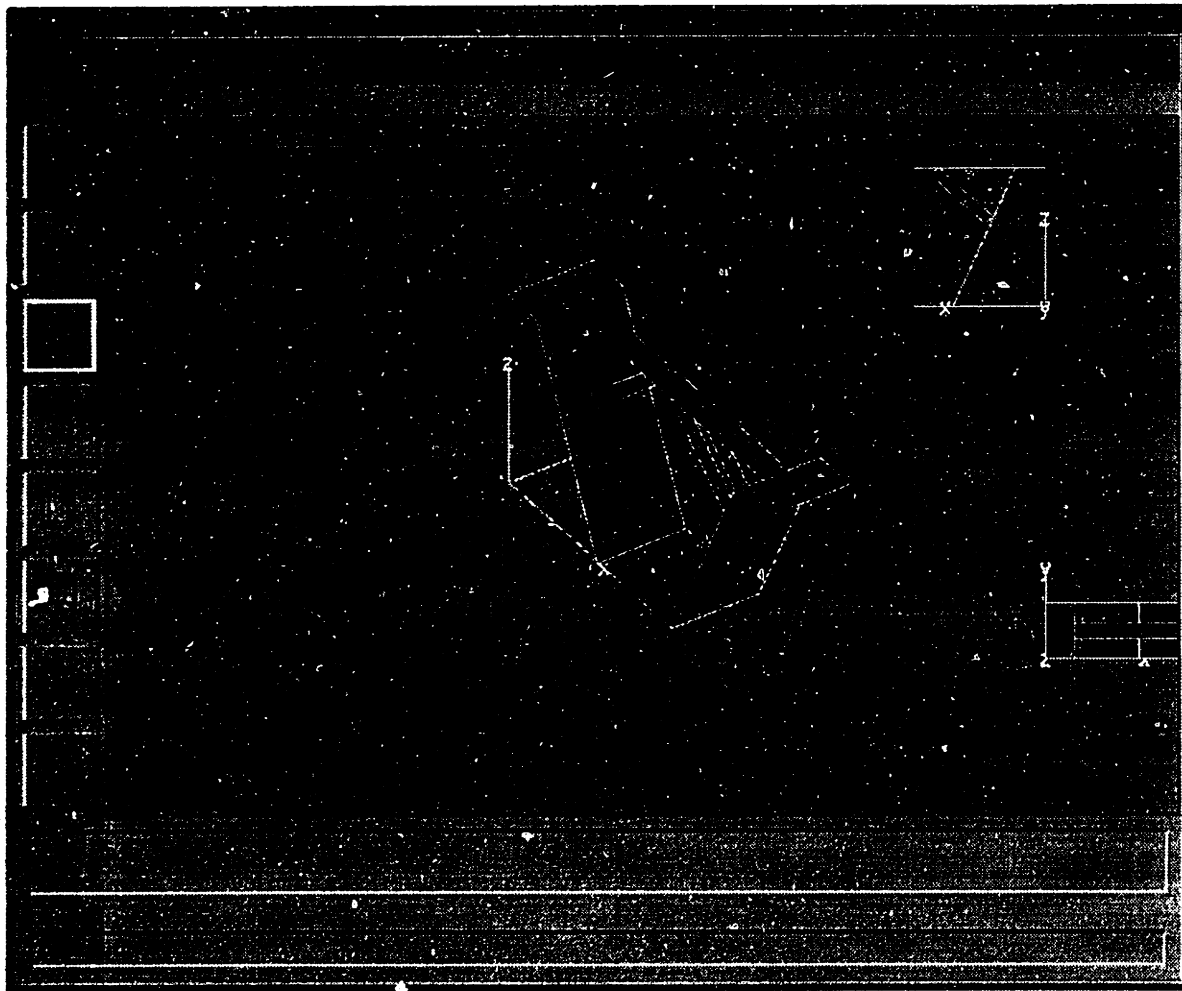


Figure 8-16: Bridge geometry: Each of the piers consists of two columns.

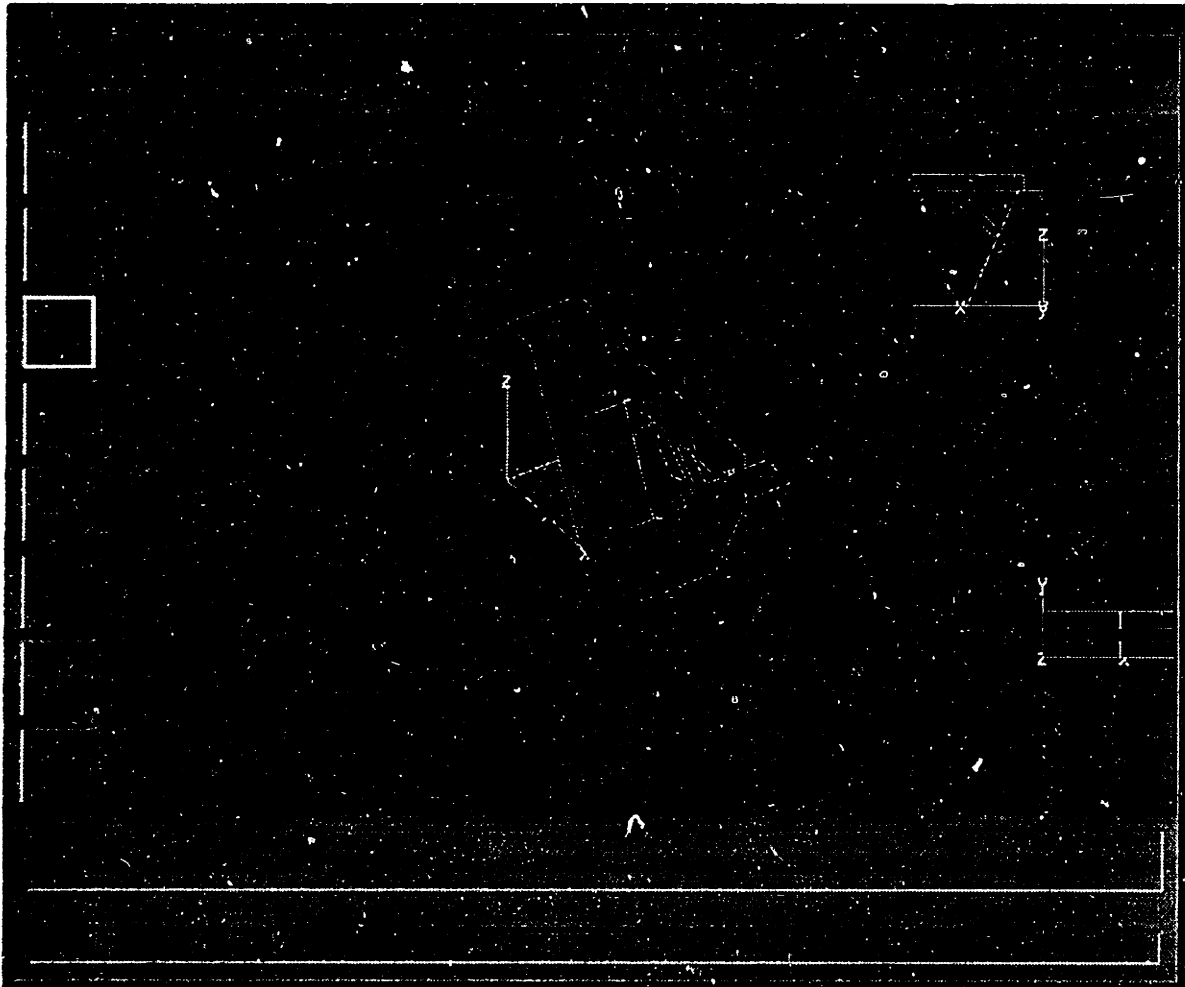


Figure 8-17: Bridge geometry: Final design at the end of the conceptual design stage.

---

## Chapter 9

# Conclusions

This thesis has presented a systematic framework for form conception in engineering design. The central idea is an evolution of form based on a symbolic decomposition of design. The approach is based on a structured representation model for symbolic design and explicitly identifying and decoupling the various elements which constitute form conception. An example demonstrated the feasibility of the approach, and hinted at some of the advantages. This chapter presents some conclusions drawn from the effort to support form conception. Section 9.1 summarizes the overall approach. Section 9.2 identifies the main contributions of this research. Section 9.3 positions the research with respect to various related approaches. Section 9.4 concludes the thesis with some recommendations on future research.

### 9.1 Summary

The requirements of a conceptual design agent are identified in Chapter 1. Support for conceptual design must include not only assisting in the symbolic and functional aspects of design, but also help to translate a broad symbolic description of a high level system into a physically realizable artifact. In terms of system support for conceptual design, the design agent needs to provide representation support and problem-solving facilities to operate on the representation. This representation needs to include various aspects of design: product, process, relationships, design context information, and geometry.

The CONGEN framework was developed in response to these requirements. CONGEN provides an information model to represent the symbolic aspects of design. An *artifact* is the basic product element, which consists of *function*, *form* and *behavior*. Artifacts may

## 9.1 Summary

---

have various relationships: composition, functional, spatial, etc. One of the important process elements is *Goal*, which is a task which may present a decision point during the design process. Design *contexts* are created corresponding to each design alternative. These encapsulate all information relevant to particular alternatives. The user may affect the state/flow of the design at any point by directly modifying attributes, adding elements/relationships, making decisions, etc.

The form conception framework consists of three distinct aspects: generating spatial constraints, instantiating feasible solutions to satisfy these constraints, and presentation of geometry. It is based on the observation that many of the important spatial relationships and constraints in the design process are derived from the functional aspects of design. This thesis has presented a mapping framework based on *localized* knowledge mappings from functional to spatial relationships. It also developed a very general specification framework for constraints. This specification is based on a qualitative interval algebra approach which provides an intuitive, high level scheme for specifying relationships in conceptual design. The specification also enables relating functional relationship to spatial relationships at a local level.

To solve the instantiation problem, an approach using Asynchronous Teams of Agents is proposed. The approach is conceptually simple, yet shown to be highly effective. The algorithm operates on a population of solutions using localized improvement knowledge, which is embedded within the constraints themselves. A generalized spatial reasoning effort is thus enabled by very simple local spatial knowledge. The Ateams approach is inherently modular and extensible to include more sophisticated constraint satisfaction techniques over specific constraint subsets.

The requirements for the geometry definition from a conceptual design perspective are also presented. The geometric abstractions for primitive components permit description of evolving and incomplete geometries. The underlying geometric representation is based on a non-manifold geometric model which supports such descriptions.

The implementation of CONGEN is an integrated system development effort. It has extensive knowledge representation and inferencing ability drawn from COSMOS, and also a complete geometric environment provided by GNOMES, a non-manifold geometric modeler. The CONGEN model and implementation provide extensive *support* facilities: the user may interact with the system at any point to control the design flow.

### 9.2 Contributions

This thesis has addressed the issue of form conception in engineering design. The overall design support framework also addresses many of the requirements for conceptual design presented in Chapter 1. Some of the research contributions are now identified:

- *An approach to form conception.* The primary goal of this thesis was to identify a systematic approach which permits computer support for the process by which the form of design artifacts is conceived. The symbol-form mapping is identified as the crucial issue to be addressed in developing such a support system. Related to this goal, the thesis explicitly identified and decoupled the issues which constitute the problem. Solutions to each of the three issues is presented:
  - (a) *Generating spatial constraints.* Spatial relationships and constraints related to spatial attributes of design objects are identified as a course of the symbolic evolution process. The concept emphasized is a *localized* mapping from function to form based on an explicit representation of functional and spatial relationships. A high-level specification framework based on qualitative interval algebra presented by Mukerjee, is used to meet the representation requirements for the spatial relationships.
  - (b) *Instantiating feasible solutions.* A novel approach to constraint satisfaction based on Asynchronous Teams of Agents is developed. The key advantages of the approach are its ability to: (1) handle arbitrary constraint formulations in a unified framework; (2) use very simple localized knowledge to build a general reasoning mechanism; (3) retain the flexibility to exploit sophisticated techniques over constraint subsets; (4) generate families of solutions; (5) allow the user to directly influence the nature of the solutions generated by the algorithm; and (6) handle dynamic constraint environments in design.
  - (c) *Geometry representation for conceptual design:* The thesis has presented a scheme to enable representation of evolving, incomplete geometries during conceptual design.
- *Integrated product and process representation.* The model allows considerable flexibility in knowledge representation. It accounts for representation of artifact and relationships, modeling the design process, and lays out a set of well-defined interac-



### 9.3 Comparison with Related Research

---

tions with the primitives provided by the model. The model allows the exploration of multiple conceptual design alternatives. It provides a top-down hierarchical decomposition approach with support for bottom-up aspects of design. The model is validated by an extensive implementation.

- *Disaggregation and localization* of knowledge is a key concept advocated in each element of the overall framework, in keeping with the object oriented philosophy.
- *Problem solving support* includes rule-based inferencing (both forward and backward chaining), procedural knowledge representation (as methods of objects) and constraint satisfaction,
- *Geometric modeling facilities.* GNOMES has been completely integrated with CONGEN to provide the basic geometric modeling facilities needed in design. The GNOMES non-manifold geometric engine can be directly controlled by the functional modelers. GNOMES provides a complete set of editing facilities for geometry. An abstraction layer corresponding to domain shapes is built over the basic geometric model.
- *Database support.* CONGEN is built over an object-oriented database, EXODUS. This provides persistence for complex engineering data. The data management also permits scalability of the system: the large amount of data generated in pursuing multiple alternatives is stored on the database, and is available for ready retrieval.
- *An integrated architecture for design.* The interfaces between the various support modules have been designed from an overall system perspective. CONGEN thus provides a powerful set of functionalities in an integrated design environment.

### 9.3 Comparison with Related Research

This thesis has presented an *overall system* approach to developing a support framework for form conception. Little has been reported in literature by way of directly related research; however, many efforts exist which relate to aspects of this work. This section discusses the overall framework, and presents comparative analysis with several related efforts over specific aspects of the work.

This thesis has identified and discussed elements of form conception. This explicit decoupling now permits comparison with some other geometry-related design support techniques available currently. Henderson [19] reports an effort to represent functionality in

### 9.3 Comparison with Related Research

---

product models. It does not attempt to help the support of form conception. Rather it allows the user to identify features on a physical model and records the intent based on a process model. In the absence of any support for symbolic design evolution, the CONGEN framework reduces roughly to the “design-with-features” approach [10, 45]: the design may still be carried out by the user directly instantiating the geometric representation primitives provided. This approach is purely geometry-based, albeit the primitives have some domain significance. Efforts to couple the feature-based design with constraint management techniques also have been reported [27, 28], etc. Many of these approaches are *parametric*. They provide support at a post-conceptual design stage, when the overall components are specified, and the overall topological connectivities are often encoded directly by the user in the form of an instantiation. The focus of these approaches is on *optimizing* the resultant shape. The parametric approaches allow for variational studies in preliminary design, but their success in defining assemblies of objects has been limited. As compared to parametric approaches, this thesis tackles the design problem at an earlier stage, when the components are as yet unspecified, and the connectivities are unknown.

The problem stage prior to the instantiation of a feasible solution is similar to the problem tackled by shape grammars: the design components and the spatial relationships that must be satisfied between them are known at this point. The Qualitative Spatial Relationships are conceptually akin to the rewrite rules in the shape grammars. However, knowledge representation in shape grammar consists of directly encoding the rewrite rules which represent relationships between objects. The research presented in this thesis makes the basis of these relationships more explicit. This permits more flexibility for conceptual design. Moreover, the representation of the spatial relationships is decoupled from the generative capability of the system. This thesis reflects the belief that this is a more robust and general approach.

The instantiation approach presented in this thesis is derived from a specific focus on conceptual design: the need to specify relationships between objects instead of low-level parametric constraints, the need for multiple feasible solutions, and being able to handle arbitrary constraint formulations are all important requirements which are supported by the technique developed. The generality of the technique far surpasses any existing algorithm for constraint satisfaction.

Overall, the work adds value to the current state of research primarily from a notion of *automation* of form alternative generation in an interactive design paradigm. This extends

## 9.4 Future Directions

---

the capabilities of geometry-based approaches in which the *designer* instantiates prototypes at different stages and specifies geometric constraints. The automation of prototype instantiation is consistent with the notion of design evolution in a human-computer interactive environment. In such an environment, the system enumerates the alternatives, performs consistency checking, and propagates the decisions made by the designer. This thesis treats generation of form alternatives analogously to generation of symbolic alternatives (depicting the logical structure of the design artifacts), as a natural part of the design evolution. The form evolution is based on a strong model of the *symbolic* design evolution. This enables the capture of design intent, as well as maintaining consistency and providing visualization support. A functional basis for form has been identified as Functional Modeling [23]. The research presented in this thesis is an attempt to capture such a notion within a coherent framework. Evolution of the form implies capturing the qualitative spatial descriptions at the abstract conceptual design stage and yet providing for support through to the more detailed design stages. The approach differs from other approaches [28] in allowing a uniform framework for dealing with spatial abstractions, be it conceptual design or detailed design. Work done by Mukerjee [32] on qualitative spatial relationships is a step in this direction, enabling the capture of functional intent behind spatial connectivities. The current research further augments this approach with an *automation* of the generation of these relationships, a very general instantiation mechanism and powerful non-manifold geometric modeling support.

## 9.4 Future Directions

The research is neither without limitations nor complete. This section proposes some further directions of inquiry that might prove fruitful.

- *Constraint consistency* is a problem rather inadequately addressed by this research. Unfortunately, the very generalized approach to constraint satisfaction developed in this thesis presents problems for the traditional constraint consistency algorithms (which typically involve propagation and transitivity information). A potential approach might be to integrate the well-developed algorithms (e.g., COPLAN) with a more heuristic approach in some coherent manner.
- *Efficiency of the ATeams search*. The ATeams architecture was motivated by the requirement of a general constraint satisfaction algorithm for conceptual design. The

## 9.4 Future Directions

---

development was guided largely by intuition, yet the approach proved surprisingly effective. A comparison of the ATeams search technique against a comparable approach, genetic algorithms, is reported by Humair [21]. A more theoretical investigation into the nature of the search and its potential convergence behavior needs to be conducted in order to gain additional insight into the constraint satisfaction architecture.

- *Theoretical analysis of the symbol-form mapping.* The symbol-form mapping framework developed in this work needs to be explored from a theoretical knowledge-use-analysis perspective.
- *Behavior representation.* The model of symbolic evolution presented in the thesis is designed to be flexible, but limited to an extent by the inadequate representation of behavior and causal processes. True innovation in the symbolic aspects of design is closely tied in with causal analysis, both for choosing new components in the design, and for evaluating alternative structural configurations generated during the design process. Unfortunately, research in this area is still in a rather premature state.
- *A more thorough integration of rule-based and procedural programming.* This thesis has presented a seamless integration of heuristic and procedural code (provided by COSMOS), as a powerful mechanism for distributed knowledge representation. The current implementation of method invocation from the rules is rather limited. From a C++ programming viewpoint, type-checking for these methods need to be tackled more thoroughly. Solution to this problem would possibly involve accessing the C++ compiler implementations. The current implementation can only fire a set of predefined methods. Extending the method invocation to arbitrary methods needs to be explored in greater detail. For COSMOS, the ability to enter generalized object queries in the conditional expressions would greatly enhance the scope of the system.
- *Stronger domain analysis.* The feasibility of the proposed approach has been demonstrated by running the bridge design example on the prototype system. Testing the example on a larger scale system would require a stronger analysis of domain shapes, and also more extensive function-form knowledge mappings.
- *Testing in a collaborative framework.* The model of design reported in this thesis is designed to be consistent with the SHARED model [71]. Testing of this model in such a distributed product development framework is pending.

---

# Bibliography

- [1] Dixon, J.R., "Research in Designing With Features," *Design Theory 88, Proceedings of the NSF Grantees Workshop in Engineering Design*, RPI, Troy, New York, June, 1988.
- [2] Ahmed, S., Wong, A., Sriram, D., and Logcher, R., "Object-Oriented Database Management Systems for Engineering: A comparison," *Journal of Object-Oriented Programming*, June, 1992.
- [3] Alberts, L. K., Wognum, P.M., and Mars, N. J. I., "Structuring Design Knowledge on the Basis of Generic Components," *AI in Design*, Editor: Gero, J. S., 1992.
- [4] Bardis, L. and Patrikalakis, N., *Topological Structures for Generalized Boundary Representations*, Design Laboratory Memorandum 91-18, Department of Ocean Engineering, M.I.T., 1991.
- [5] Buchanan, S. A. and de Pennington, A., "Computer Definition System: A Computer-Algebra Based Approach to Solving Geometric-Constraint Problems," *CAD journal*, 1993.
- [6] Bylander, T. and Chandrasekharan, B., "Understanding Behavior using Consolidation," in *Proceedings of IJCAI-85*, pp. 23-34., 1985.
- [7] Carey, M.J., DeWitt, D.J., Graefe, G., Haight, D.M., Richardson, J.E., Schuh, D.T., Shekita, E.J., and Vandenberg, S.L., "The EXODUS Extensible DBMS Project: An Overview," *Readings in Object-Oriented Databases*, Zdonik, S., and Maier, D., eds., Morgan-Kaufman, 1990.
- [8] Cutkosky, M., Tenebaum, J., and Miller, D., "Features in Process-Based Design," *ASME Computers in Engineering Conference*, San Francisco, Aug. 1988.

## BIBLIOGRAPHY

---

- [9] Davis, R., "Reasoning from First Principles in Electronic Trouble-shooting," *Int. Jnl. Man-Machine Studies*, 19, pp. 403-423.
- [10] Dixon, J.R., "Research in Designing With Features," *Design Theory 88, Proceedings of the NSF Grantees Workshop in Engineering Design*, RPI, Troy, New York, June, 1988.
- [11] Falkenheimer, B. and Forbus, K. D., "Compositional Modeling: Finding the Right Model for the Job," in *Artificial Intelligence 51*, , Kluever Publishers, 1991.
- [12] Fitzhorn, P. A., "Engineering design is a computable function," *AIEDAM*, Vol. 8, pp. 35-44, Feb 1994.
- [13] Leondes, C. T. and Freidman, G. J., "Constraint Theory, Part I, Fundamentals," *IEEE Transactions on Systems Science and Cybernetics*, Vol. SSC-5, No.1, pp. 48-56, 1969.
- [14] Fromont, B., and Sriram, D., "Constraint Satisfaction as a Planning Process," *AI in Design*, Editor: Gero, J. S., 1992.
- [15] Gero, J.S., "Design Prototypes: a Knowledge Representation Schema for Design," *AI Magazine*, pp. 26-36, 1990.
- [16] Goldberg, R., *A Gentle Introduction to Genetic Algorithms: Optimization, Search and Learning*, 1989.
- [17] Gossard, D.C, Zuffante, R.P., Sakurai, H., "Representing Dimensions, Tolerances and Features in MCAE Systems," *IEEE Computer Graphics and Applications*, pp 51-59, Mar. 1988.
- [18] Hakim, M., "A Representation for Evolving Engineering Design Product Models," Technical Report, Dept. of Civil Engg., CMU, 1992.
- [19] Henderson, M., "Representing Functionality and Design Intent in Product Models," *2nd ACM Solid Modeling Conference*, Montreal, 1993.
- [20] Holland, J. H., *Adaptation in Natural and Artificial Systems*, Univ. Michigan Press, Ann Arbor, 1975.
- [21] Humair, S., *An Approach to Solving Constraint Satisfaction Problems Using Asynchronous Teams of Autonomous Agents*, S.M Thesis, Department of Civil Engineering, M.I.T., Aug. 1994.

## BIBLIOGRAPHY

---

- [22] Iwasaki, Y., and Chandrasekharan, C., "Design Verification through Function and Behavior-Oriented Representations: Bridging the Gap between Function and Behavior," *Proceedings of the Thirteenth International Joint Conference in Design, Pittsburgh., 1992*
- [23] Johnson, A.L., "Functional Modelling: A New Development in Computer-Aided Design," *Intelligent CAD, II*, Yoshikawa, H. and Holden, T. (Editors), IFIP, 1990.
- [24] He, L., *A Non-Manifold Geometry Modeler: An Object Oriented Approach*, S.M Thesis, Department of Civil Engineering, M.I.T., Feb. 1993.
- [25] Light, R. A., *Symbolic Dimensioning in Computer Aided Design*, S.M Thesis, MIT, Feb. 1980.
- [26] Light, R. A. and Gossard, D.C., "Variational Geometry: A New Method for Modifying Part Geometry for Finite Element Analysis," *Computers and Structures*, Vol. 17, no. 5, pp 903-909, 1983.
- [27] Lin, W., and Myklebust, A., "A Constraint Driven Solid Modeling Environment," *2nd ACM Solid Modeling Conference*, Montreal, 1993.
- [28] Mantyla, M., "A Modeling System for Top-down Design of Assembled products," *IBM Journal of Research and Development*, Volume 34, Number 5, Sept. 1990
- [29] Margelis, G., *Geometric Abstractions for Conceptual Design*, S.M. thesis, Intelligent Systems Laboratory, Dept. of Civil and Environmental Engg., MIT, 1994.
- [30] Mitchell, W. J., "Reasoning about Form and Function," *Computability of Design*, Ed. Yehuda. E. Kalay, John Wiley, 1987.
- [31] Mittal, S., Dym, C.L., and Morjaria, M., "PRIDE: An Expert System for the Design of Paper Handling Systems," *IEEE Computer*, July 1986, pp. 102-114.
- [32] Mukerjee, A., "Qualitative Geometric Design," Symposium on Solid Modeling Foundations and CAD/CAM Applications, Editors: Rossignac, J. and Turner, J., ACM Press, 1991.
- [33] Murthi, S.S. , and Addanki, S., "PROMPT: An Innovative Design Tool," *AAAI-87*, pp. 637-642.

## BIBLIOGRAPHY

---

- [34] Murthy, S., *Synergy in Cooperating Agents: Designing Manipulators from Task Specifications*, PhD Thesis, CMU, Sept. 1992.
- [35] Peña-Mora, F., Sriram, D., and Logcher, R., "SHARED-DRIMS: SHARED Design Recommendation-Intent Management System," *2nd IEEE Workshop on Enabling Technologies Infrastructure for Collaborative Enterprises (WET ICE)*, 1993.
- [36] Richardson, J.E., Carey, M.J., and Schuh, D.T., "The Design of E Programming Language," *ACM Transactions of Programming Languages and Systems*, Vol. 15, No. 3, 1993.
- [37] Rodenacker, W., "Methodisches Konstruieren," Springer, Berlin, Heidelberg, New York, 1971.
- [38] Rossignac, J. and O'Connor, M., "Selective Geometry Complex: A Dimension-Independent Model for Pointsets with Internal Structures and Incomplete Boundaries," *Geometric Modeling for Product Engineering*, Ed: Wozny, M., J.
- [39] Rossignac, J. and Requicha, A. A. G., "Constructive Non-Regularized Geometry," *Computer-Aided Design*, v. 23, n. 1, 1991.
- [40] Rossignac, J., *Advanced Representations for Geometric Structures*, Seminar given at IESL, Department of Civil Engineering, M.I.T., December 10, 1992.
- [41] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenzen, W., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [42] Schodek, D. L., *Structures*, Englewood Cliffs, N.J. : Prentice-Hall, 1980.
- [43] Serrano, D., *Constraint Management in Conceptual Design*, PhD Thesis, MIT, Oct 1987.
- [44] Shah, J.J. and Rogers, M.T., "Feature Based Modeling Shell: Design and Implementation," *Design Theory 88, Proceedings of the NSF Grantees Workshop in Engineering Design*, RPI, Troy, New York, June 1988.
- [45] Shah, J.J., "Conceptual Development of Form Features and Feature Modelers," *Research in Engineering Design*, Vol 2, pp.93-108, 1991.



## BIBLIOGRAPHY

---

- [46] Smithers, T., "AI-Based Design versus Geometry-Based Design," *Computer Aided Design* 21(3): 141-150., 1989.
- [47] Sriram, D., *Intelligent Systems for Engineering: Knowledge-based and Neural Networks*, Technical report, IESL, MIT, 1988.
- [48] Sriram, D., et al., "An Object-Oriented Knowledge Based Building Tool for Engineering Applications," IESL Research Report R91-16, Intelligent Engineering Systems Laboratory, M.I.T, 1991.
- [49] Sriram, D., Cheong, K., and Kumar, L., "Engineering Design Cycle: A Case Study and Implications for CAE," *Knowledge Aided Design*, Editor: Green, M., Academic Press, 1991.
- [50] Sriram, D. and Logcher, R., "The MIT DICE Project," *IEEE Computer*, Special Issue on Concurrent Engineering, pp. 64-65, January 1993.
- [51] Sriram, D., Logcher, R., Groleau, N., and Cherneff, J., "DICE: An Object-Oriented Programming Environment for Cooperative Engineering Design," *AI in Engineering Design*, Vol. III, Editors: Tong, C. and Sriram, D., Academic Press, 1992.
- [52] Sriram, D., Wong, A., and He, L., "An Object-Oriented Non-manifold Geometric Engine," *CAD Journal*, to be published.
- [53] Steele, G. J., "Constraints," AI Memo no. 50, MIT AI lab., Nov. 1980.
- [54] Steele, G. J., *The Definition and Implementation Of a Computer Programming Language Based on Constraints*, PhD Thesis, MIT, Aug 1980.
- [55] Stefik, M. and Bobrow, D.G., "Object-Oriented programming: Themes and Variations," *AI Magazine*, 1986.
- [56] Stiny, G., "Introduction to Shape and Shape Grammars," *Environment and Planning B: Planning and Design*, 7:343-351.
- [57] Sutherland, I., *Sketchpad-A Man Machine Graphical Interface*, PhD Thesis, MIT, 1963.
- [58] Talukdar, S. T. and Desouza, P. S., "Scale Efficient Organizations," *Proceedings of the IEEE International Conference on Systems Science and Cybernetics*, 1992.

## BIBLIOGRAPHY

---

- [59] Tomiyama, T. and Yoshikawa, H., "Extended General Design Theory," in *Design Theory for CAD, Proceedings of the IFIP WG5.2 Working Conference 1985, Tokyo*, Yoshikawa, H. and Warman, E.A (eds.), pp. 95-130., North-Holland, Amsterdam, 1986.
- [60] Tong, C. and Sriram, R. D., *Introduction to Artificial Intelligence in Engineering Design*, Vol. 1, Academic Press Incorporated, 1992.
- [61] Umeda, Y., Takeda, H., Tomiyama, T., and Yoshikawa, H., "Function, Behavior and Structure," *Applications of Artificial Intelligence in Engineering V*, Vol. 1, Design, pp. 177-193, 1990.
- [62] Van Beek, P., "Approximation Algorithms for Temporal Reasoning," *In Proceedings of the Tenth IJCAI*, Detroit, 1989.
- [63] Vescovi, M., Iwasaki, Y., Fikes, R., and Chandrasekharan, B., "CFRL: A Language for Specifying the Causal Functionality of Engineered Devices." *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 1993.
- [64] Vilain, M., Kautz, H., and van Beek, P., "Constraint Propagation Algorithms for Temporal Reasoning: A Revised Report," *Proceedings of AAAI*, 377-382, 1986.
- [65] Weiler, K., *Topological Structures for Geometric Modeling*, Phd. Thesis, Rensselaer Polytechnic Institute, Aug. 1986.
- [66] White, R., Gergely, P. and Sexsmith, R., *Structural Engineering*, John Wiley & Sons, Inc., NY, 1972.
- [67] Woodbury, R.F., and Oppenheim, I.J., "An Approach to Geometric Reasoning," *Intelligent CAD, I*, Proceedings of the IFIP TC 5/WG 5.2 Workshop on Intelligent CAD, North-Holland, 1987.
- [68] Wong, A. and Sriram, D., "Geometric Modeling Facilities for Product Modeling," *Intelligent Engineering Systems Laboratory*, M.I.T, 1993.
- [69] Wong, A., Sriram, D., et. al., *Design Document for the GNOMES Geometric Modeler*, IESL Technical Report, Intelligent Engineering Systems Laboratory, M.I.T. December 1991.

## BIBLIOGRAPHY

---

- [70] Wong, A. and Sriram, D., "SHARED: An Information Model for Cooperative Product Development," *Research in Engineering Design*, Fall 1993.
- [71] Wong, A. and Sriram, D., *Shared Workspaces for Computer-aided Collaborative Engineering*, Intelligent Engineering Systems Laboratory, Dept. of Civil and Environmental Engineering, Technical Report No: IESL 93-06, March 1993.
- [72] Wong, A. and Sriram, D. "An Extended Object Model for Design Representation," submitted to *IEEE Transactions on Knowledge and Data Engineering*.
- [73] Zamanian, K., *Modeling and Communicating Spatial and Functional Information About Constructed Facilities*, PhD Thesis, Department of Civil Engineering, CMU, Dec 1991.

---

## Appendix A

# Sample CONGEN class declarations

---

```
/* *****  
 * data_manager.h: Definition of class data_manager.  
 *  
 * Author: Gorti Sreenivasa Rao  
 *   Intelligent Engineering Systems Laboratory  
 *   Massachusetts Institute of Technology  
 *   Date: June 23, 1993  
 * *****/  
#ifndef data_manager_h  
#define data_manager_h  
  
// Class to define the data managers (goal, plan, artifact) hashtables  
// for a particular application. Defines lookup functions, creates instances,  
// serves as the in-memory manager for persistent data, reads in application  
// data as necessary  
  
// Tasks to be performed in this class:  
// Lookup: context, goal, plan, artifact.  
// Create artifact, set default values.  
  
#include "congen_classnames.h"  
#include "GPHash.h"  
#include "ccString.h"
```

10

20

---

```
declare(PHash Table,Plan);
declare(PHash Table,Goal);
declare(PHash Table,Context);
```

```
dbclass Data_manager{
```

30

```
private:
```

```
    PHash Table(Plan) *ThePlanHashTable;
    PHash Table(Goal) *TheGoalHashTable;
    PHash Table(Context) *TheContextHashTable;
    collection<Plan> * all_plans;
    collection<Goal> * all_goals;
    collection<Context> * all_contexts;
    // Note that we have a redundant storage mechanism above, but while the
    // collections store all the objects, the hashtables store the pointers for
    // quick retrieval by name.
    Context * curr_context ;
    // the function create_Goal will have to be overloaded for creation of
    // ArfGoal, and the rest of them...
    Goal* create_ArfGoal(char*,GoalEditor* = NULL);
    Goal* create_ModGoal(char*, char* = NULL, char* = NULL, GoalEditor* = NULL);
    Goal* create_AbsGoal(char*,GoalEditor* = NULL);
    Goal* create_FunGoal(char*,GoalEditor* = NULL);
    Plan* create_Plan(char*, PlanEditor* = NULL);
    void remove_Plan(char*);
    void remove_Goal(char*);
    Context* create_root_Context(Goal*);
    void remove_Context(char*, char* = NULL);
    dbint no_goals, no_plans, no_contexts ;
```

40

50

```
public:
```

```
    ccString application_name;
    Data_manager(char* str);
    ~Data_manager();
    void print();
```

60

```
    inline int get_no_of_goals(){return no_goals;}
    inline int get_no_of_plans(){return no_plans;}
    inline int get_no_of_contexts(){return no_contexts;}
    inline int get_no_of_artifacts();
```

---

```
void register_Artifact(Artifact*);
void unregister_Artifact(char*,char*);

// public, since these functions interface with the c functions of
// Collmanager.e (src/COSMOS/obm directory)
Goal* lookup_Goal(char*); // direct lookup
Plan* lookup_Plan(char*); // direct lookup
Context* lookup_Context(char*, char* = NULL); // direct lookup
root* lookup_Artifact(ccString,char* = NULL, Context* = NULL);

Context* create_Context(Context*, char* = NULL);
void set_curr_context(Context*);
inline Context* get_curr_context(){return curr_context;}
void set_alias_for_context(Context*,char*);

Goal* get_rtgoal();
const collection<Goal>* get_all_goals(){return all_goals;}
const collection<Plan>* get_all_plans(){return all_plans;}
const collection<Context>* get_all_contexts(){return all_contexts;}

friend class Congen;
friend class PlanEditor;
friend class GoalEditor;
};
#endif
```

---

---



---

```

/*****
* context.h: Definition of class Context.
*
* Author: Gorti Sreenivasa Rao
*       Intelligent Engineering Systems Laboratory
*       Massachusetts Institute of Technology
* Date: Feb 25, 1992
*****/

#ifndef context_h
#define context_h                                10

#include "congen_classnames.h"
#include "E/collection.h"
#include "ccString.h"
#include "GPHash.h"
#include "GPList.h"

declare (PHashTable,Artifact);
dbclass QSR;
declare(PList, QSR);                            20
class AteamUI;
/* Declaration of class Context */
dbclass Context
{
private:
    Goal * root_goal ;
    Goal * curr_goal ;
    Artifact * curr_artifact;
    // convenience mechanism for allowing us to place a new artifact in the
    // component hierarchy, as a component of curr_artifact                30

    SpecFrame* specs ;
    PHashTable(Artifact)* TheArtifactHashTable ; //note that the classobjects
    // in SpecFrame also keep the list of all the objects, the hashtable
    // is only for quick access..
    Declist * declist ;
    collection<Prule>* plans; /* Keeps track of goal ordering for various
        plans, as per this design context */

    PList(QSR)* all_qsr; // ALL the spatial relationships in this context.  40

```

---

---

```

// Maybe these should be stored with the artifacts themselves ?

void set_rtgoal(Goal* gl){root_goal = gl ;}

public:
dbenum { UNEXPANDED, EXPANDED, INPROCESS, COMPLETED, FAILED };
dbint status;
dbint id ;
ccString id_name ;

Context(Goal* head = NULL , int s = UNEXPANDED);           50
Context(const Context*, ccString);
~Context();
void copy_artifacts(const Context* ctx);

void set_curr_goal(Goal* gl){curr_goal = gl;}
inline Goal * get_curr_goal(){return curr_goal;}
inline Goal * get_root_goal(){return root_goal;}
void set_curr_artifact(Artifact* art){curr_artifact = art;}
inline Artifact * get_curr_artifact(){return curr_artifact;}
Decision* get_decision(Goal*);                             60
void add_decision(Goal*, ccString) ;
inline Declist* get_declist(){return declist ;}
Prule* get_prule(Plan*);
inline SpecFrame * get_specs(){return specs ; }

void add_qsr(QSR*);
PList(QSR)* get_qsr(){return all_qsr;}
int expand() ;
void display();
void print();                                             70
inline root* lookup_Artifact(ccString,char* = NULL);
void register_Artifact(Artifact* );
void unregister_Artifact(char*, char*);

// Functions to get the input and output from the Ateams algorithm
void setup_Ateams(AteamUI*);
void cleanup_Ateams();

friend class Congen;
friend dbclass Data_manager;                             80
};

```

---



---

```
/* extern Context* lookup_context(ccString, ccString); */
```

```
#endif context_h
```

---

---



---

```

/*****
* artifact.h: Definition of class Artifact, used to represent the
*           artifacts (design products).
*
* Author: Gorti Sreenivasa Rao
*         Intelligent Engineering Systems Laboratory
*         Massachusetts Institute of Technology
* Date: Feb 25, 1993
*****/
#endif artifact_h                                10
#define artifact_h

#include "xheaders.h"
#include "congen_classnames.h"
#include "GPList.h"
#include "knowledg.h"
#include "ccString.h"
#include "composite.h"

dbclass MetaClass;                                20
dbclass Geometry ;
dbclass Func_rel;

declare(PList,Func_rel);
dbclass Artifact:public Composite
{
private:
    int shared_mode ; // if it makes no sense to copy the artifact over
    // contexts, the shared mode is set to one at time of creation
protected:                                       30
    Geometry *geom;
    ccString parent ; /* name of the parent Goal object */
    ccString subplan ; /* name of the sub-Plan object */
    ccString rulebase;

    void set_parent(ccString gl ){parent = gl ;}
    void set_subplan(ccString pl){ subplan = pl ;}

    PList(Func_rel) * part_rels; // all the part relationships are stored here

```

---

---

```

public:
enum { UNEXPANDED, EXPANDED, INPROCESS, COMPLETED, FAILED };
dbint status;
Artifact(char*, char*, ArtifactEditor * = NULL );
~Artifact();

void duplicate(Artifact*);
inline void set_geometry(Geometry* g1) {geom = g1;}
void set_id(int);
char* get_classname(); 50
char* get_instancename();
inline Geometry* get_geometry(){return geom;}
inline char* get_id(){return id_name;}
inline int is_shared(){return shared_mode; }
inline void set_shared_mode(int val){shared_mode = val;}
Goal* get_parent() ;
Plan * get_subplan();
void expand(Widget,Widget) ;
void place_in_component_hierarchy(Artifact*);
void make_functional_relationship(char*, Artifact*, Artifact*, char*); 60

void show_geometry();
Widget display( Widget, Widget );
void display_in_synthesizer(Widget, Widget) ;
void display_hierarchy(Widget, Widget );
virtual Val direct_get_value(dbchar*);
virtual int direct_put_value(dbchar*,char*);
friend class ArtifactEditor;
friend dbclass MetaClass ;
friend dbclass Goal; 70
};
#endif

```

---

---

```

/*****
* goal.h: Definition of class Goal.
*
* Author: Gorti Sreenivasa Rao
* original code by: Kevin K. W. Cheong
*      Intelligent Engineering Systems Laboratory
*      Massachusetts Institute of Technology
*      Date: Nov 10, 1992
*****/

#ifndef goal_h
#define goal_h
10

#include "knowledg.h"

/*
* Declaration of class Goal
*/
dbclass Goal : public Knowledge {

protected:
20
    ccString rulebase ;
    ccString effects_rulebase ; // asserting consequences of a decision
    ccString parent ; /* Name of a Plan object */

    void set_parent( char* );
    void set_rulebase(char * rules){rulebase = rules ;}
    void set_effects_rulebase(char * rules){effects_rulebase = rules ;}
    Goal( char *, GoalEditor * = NULL );
    ccString purpose ; // To indicate which subclass of the goal it is
    virtual char* get_art_class(){return NULL;}
    virtual char* get_art_slot(){return NULL;}
    virtual void set_choice(char*){}
    virtual void clear_choices(){}
    virtual void assert_effects();

public:
    Plan *get_parent() ;
    int check_truth(char*,char*,char*);
    Widget display( Widget, Widget );
    virtual void place_in_component_hierarchy(Artifact*){}
    virtual ~Goal();
40

```

---

---

```

virtual PList(ccString)* get_choices(){return NULL;}
virtual Knowledge* get_child(){return NULL;}
virtual void clear_knowledge();
virtual void expand(Widget,Widget){}
virtual void display_in_synthesizer(Widget,Widget){}
virtual void display_hierarchy( Widget, Widget ){}
virtual void fire_forward_chainer(char*){}
virtual void pursue_choice(ccString, Widget){}
friend class GoalEditor;
friend dbclass Plan;
friend class Congen;
    //friend friend int read_goal(ifstream&);
};

/* Interfacing functions to the Goal instance manager */
/* extern Goal *lookup_Goal( char * ); */

/*****
/* Goal associated with creating components */
*****/

dbclass ArfGoal : public Goal
{
private:
    PList(ccString) *choices; /* ALL possible choices defined by expert */
    void set_choice(char* possible);
    void clear_choices();
public:
    ArfGoal(char*, GoalEditor* = NULL);
    ~ArfGoal(){clear_knowledge();}
    virtual void clear_knowledge();
    PList(ccString)* get_choices(){return choices;}
    void expand(Widget,Widget);
    void display_in_synthesizer(Widget,Widget);
    void display_hierarchy( Widget, Widget );
    Knowledge* get_child();
    void pursue_choice(ccString, Widget);
    void fire_forward_chainer(char*){};
friend class GoalEditor;
friend class Congen;
};

```

---

```

/*****/
/* Goal associated with slot of a particular class */
/* Does not extend to components, though */
/*****/
dbclass ModGoal : public Goal
{
private:
    ccString Art_class;
    ccString Art_slot ;
    PList(ccString) *choices; /* ALL possible choices defined by expert */
    void set_choice(char* possible);
    void clear_choices();
    char* get_art_class(){return Art_class;}
    char* get_art_slot(){return Art_slot;}
public:
    ModGoal(char*, char* = NULL, char* = NULL, GoalEditor* = NULL) ;
    ~ModGoal(){clear_knowledge(); }
    virtual void clear_knowledge() ;
    PList(ccString)* get_choices(){return choices ;}
    void expand(Widget,Widget);
    void display_in_synthesizer(Widget,Widget);
    void display_hierarchy( Widget, Widget );
    Knowledge* get_child(){return NULL;}
    void pursue_choice(ccString, Widget);
    void fire_forward_chainer(char*){};
friend class GoalEditor;
friend class Congen;
};

/*****/
/* Goal associated with further process hierarchy steps */
/*****/
dbclass AbsGoal : public Goal
{
private:
    PList(ccString) *choices; /* ALL possible choices defined by expert */
    void set_choice(char* possible);
    void clear_choices();
public:

```

90

100

110

120

---

```

AbsGoal(char* , GoalEditor* =NULL) ;
~AbsGoal(){clear_knowledge();}
PList(ccString)* get_choices(){return choices ;}
virtual void clear_knowledge() ;
void expand(Widget,Widget) ;
void display_in_synthesizer(Widget,Widget);
void display_hierarchy( Widget, Widget );
Knowledge* get_child();
void pursue_choice(ccString,Widget);
void fire_forward_chainer(char*){};
friend class GoalEditor;
friend class Congen;
};
130

/*****
/* Executes an external function */
*****/
140
dbclass FunGoal : public Goal
{
private:
ccString func_name;
void set_function( char* nam){ func_name = nam ; }
public:
FunGoal(char* nam, char * func, GoalEditor *ed = NULL): Goal(nam,ed)
{func_name = func ; purpose = "EXTERNAL_FUNCTION";}
~FunGoal(){}
char *get_function() { return func_name; }
150
void expand(Widget,Widget){}
void display_in_synthesizer(Widget,Widget){}
void display_hierarchy( Widget, Widget );
void clear_knowledge(){}
void fire_forward_chainer(char*){}
void pursue_choice(ccString,Widget){}
friend class GoalEditor;
friend class Congen;
};
160

#endif goal_h

```

---

---

```

#ifndef FUNC_REL
#define FUNC_REL

#include "relationship.h"
#include <E/dbStrings.h>

// class which models a functional relationship in CONGEN
// Note that representation of function is simple-minded
// here, we just use the "reln_type" attribute in class
// Relationship as a descriptor for the function.
// More work needs to be done on representing function,
// and managing creation of required_by, satisfied_by links,
// through the design process somehow.
10

dbclass Func_rel : public Relationship{
protected:
    Role* needed_by;
    Role* satisfied_by;
    dbchar spatial_map[30]; // name of a subplan
    // to choose the spatial equivalents for the function
20

public:
    Func_rel(char*, Artifact*, Artifact*);
    Func_rel(Func_rel&);
    virtual ~Func_rel();
    virtual char* get_classname(){return "Func_rel";}
    inline Role* get_source(){return satisfied_by;}
    inline Role* get_target(){return needed_by;}

    void set_subplan(char*);
30
    inline char* get_subplan();
    virtual void display_in_synthesizer(Widget,Widget) ;
    virtual void display_hierarchy(Widget,Widget) ;
    virtual Widget display(Widget,Widget) ;
    virtual void expand(Widget, Widget) ;
};

#endif FUNC_REL

```

---



---

```

/*****
* decision.h: Definition of class Decision.
*
* Author: Gorti Sreenivasa Rao
*       Intelligent Engineering Systems Laboratory
*       Massachusetts Institute of Technology
* Date: March 22, 1993
*****/
#ifndef decision_h
#define decision_h                                10

#include "xheaders.h"
#include "ccString.h"
#include "congen_classnames.h"
#include "root.h"

/*
* Declaration of class Decision
*
* The list of decisions is maintained in the context, and they
* bind a choice to a goal. Note that while each artifact
* stores its parent goal, the only reverse link between a goal and
* the corresponding artifact chosen in the current context is
* provided by the decision. The name artifact choice, perhaps
* is a misnomer, since the choice may be the name of a plan.
* as in the case of an AbsGoal..
*/
dbclass Decision {
private:
    Goal *goal;                                    30
    ccString choice ;
    Widget walt[10]; /* Widget for each valid alternative */

    int which_decision(ccString);
    static int NOT_VALID ; /* class variable */

public:
    Decision(Goal*, ccString) ;
    Decision( const Decision &d ) ;
    ~Decision();

```

---

---

```
ccString valid_alternatives[10] ; /* Run time alternatives */
void set_valid_alternative(char* ) ;
Goal* get_goal();
void set_choice(char* str){choice = str;}
char* get_choice() { return (char*)choice; }
Widget display( Widget, Widget,ccString);
friend dbclass Context ;
};

//

dbclass Declist: public root{
private:
    collection<Decision>* decisions ;
    int no_decisions ;
public:
    Declist();
    ~Declist();
    Declist(const Declist&);
    void add_decision(Goal*, ccString);
    void remove_decision(Goal*);
    Decision* get_decision(Goal*);
    int direct_put_value(char* , char* , char* = NULL);
    char* direct_get_value(char* , char* , char* = NULL);
};

#endif decision_h
```

---

```

#ifndef _spatial
#define _spatial

#include "GNvector.h"
#include "relationship.h"
#include "artifact.h"
class Ref_frame;

class Evaluation;
class Srel: public ATconst 10
{
    // This ABSTRACT class defines the
    // essential spatial relationship
    // Defines the axis for the relationship and a
    // reference frame which defines this axis
    // Note that this forms the base class for
    // all primitive
    // relationships, actual 3-D relationships
    // would be represented by the composite
    // classes 20

protected:
    enum{X,Y,Z};
    int axis; // 0 for X, 1 for Y, 2 for Z
    Ref_frame* ref ;

public:
    Srel(int, Ref_frame* = NULL);
    virtual char* get_classname() = 0 ;
    virtual Role* get_source(){return NULL;}
    virtual Role* get_target(){return NULL;}
    virtual Evaluation* evaluate(Design*){return NULL ;} 30
    virtual void improve(Design*){}
    virtual ~Srel(){}

};

#endif

```

---

---

```

#ifndef _PRIM_QSR
#define _PRIM_QSR

#include <iostream.h>
#include "qsr.h"

class Evaluation ;

                                     // Subtypes of the QSR class define the
                                     // many kinds of primitive relationships
                                     // that can occur. These types are follows:           10
                                     // Point-Interval types (PI_base forms the
                                     // base class) p, f, i, b, m (point-interval
                                     // types: p is plus) interval-interval types
                                     // (II_base forms the base class) pp, fp, ip,
                                     // if, ii, bi, bf, bp, mp, mf, mi, mb, mm.
                                     // These are pretty horrible classes, we do
                                     // not expect the user to deal with them at
                                     // this level of abstraction. Higher level
                                     // operators can be built up from this
                                     // primitives                                           20

class PI_base: public QSR
{
public:
    PI_base(int, Ref_frame*, Role* = NULL, Role* = NULL);
    ~PI_base(){}
    inline virtual Evaluation* check_rel(double,Dobj*, Dobj*, Design*){return 0;}
    inline virtual char* get_classname() {return "PI_base";}
    inline virtual Evaluation* evaluate(Design*){return NULL;}
};                                                                                       30

class PI_plus: public PI_base
{
public:
    PI_plus(int, Ref_frame*, Role* = NULL, Role* = NULL);
    ~PI_plus(){}
    Evaluation* check_rel(double,Dobj*, Dobj*,Design*);
    inline virtual char* get_classname() {return "PI_plus";}
};                                                                                       40

```

---

---

```
class PI_front: public PI_base
{
public:
    PI_front(int, Ref_frame*, Role* = NULL, Role* = NULL);
    ~PI_front(){}
    Evaluation* check_rel(double,Dobj*, Dobj*,Design*);
    inline virtual char* get_classname() {return "PI_front";}
};
```

50

```
class PI_in: public PI_base
{
public:
    PI_in(int, Ref_frame*, Role* = NULL, Role* = NULL);
    ~PI_in(){}
    Evaluation* check_rel(double,Dobj*, Dobj*,Design*);
    inline virtual char* get_classname() {return "PI_in";}
};
```

60

```
class PI_back: public PI_base
{
public:
    PI_back(int, Ref_frame*, Role* = NULL, Role* = NULL);
    ~PI_back(){}
    Evaluation* check_rel(double,Dobj*, Dobj*,Design*);
    inline virtual char* get_classname() {return "PI_back";}
};
```

70

```
class PI_minus: public PI_base
{
public:
    PI_minus(int, Ref_frame*, Role* = NULL, Role* = NULL);
    ~PI_minus(){}
    Evaluation* check_rel(double,Dobj*, Dobj*,Design*);
    inline virtual char* get_classname() {return "PI_minus";}
};
```

80

---

```

class II_base:public QSR
{
    //base class for the interval–interval
    // relationships

protected:
    FI_base * rel1;
    PI_base * rel2;

    // rel1 represents the relationship of min of
    // source w.r.t the interval represented by
    // target rel2 represents the relationship of
    // max of source w.r.t target
public:
    II_base(Role*, Role*,int, Ref_frame* );
    ~II_base();
    inline virtual char* get_classname() {return "II_base";}
    virtual Evaluation* evaluate(Design*);
};

```

90

```

class II_pp: public II_base
{
public:
    II_pp(Role*, Role*,int, Ref_frame* );
    ~II_pp(){}
    inline virtual char* get_classname() {return "II_pp";}
    virtual Evaluation* evaluate(Design*);
};

```

100

```

class II_fp: public II_base
{
public:
    II_fp(Role*, Role*,int, Ref_frame*);
    ~II_fp(){}
    inline virtual char* get_classname() {return "II_fp";}
    virtual Evaluation* evaluate(Design*);
};

```

110

```

class II_ip: public II_base

```

120

---

---

```
{
public:
  II_ip(Role*, Role*,int, Ref_frame*);
  ~II_ip(){}
  inline virtual char* get_classname() {return "II_ip";}
  virtual Evaluation* evaluate(Design*);
};
```

130

```
class II_if: public II_base
{
public:
  II_if(Role*, Role*,int, Ref_frame*);
  ~II_if(){}
  inline virtual char* get_classname() {return "II_if";}
  virtual Evaluation* evaluate(Design*);
};
```

140

```
class II_ii: public II_base
{
public:
  II_ii(Role*, Role*,int, Ref_frame*);
  ~II_ii(){}
  inline virtual char* get_classname() {return "II_ii";}
  virtual Evaluation* evaluate(Design*);
};
```

150

```
class II_bi: public II_base
{
public:
  II_bi(Role*, Role*,int, Ref_frame*);
  ~II_bi(){}
  inline virtual char* get_classname() {return "II_bi";}
  virtual Evaluation* evaluate(Design*);
};
```

160

```
class II_bf: public II_base
{
```

---

```
public:
  II_bf(Role*, Role*,int, Ref_frame*);
  ~II_bf(){}
  inline virtual char* get_classname() {return "II_bf";}
  virtual Evaluation* evaluate(Design*);
};
```

170

```
class II_bp: public II_base
{
public:
  II_bp(Role*, Role*,int, Ref_frame*);
  ~II_bp(){}
  inline virtual char* get_classname() {return "II_bp";}
  virtual Evaluation* evaluate(Design*);
};
```

180

```
class II_mp: public II_base
{
public:
  II_mp(Role*, Role*,int, Ref_frame*);
  ~II_mp(){}
  inline virtual char* get_classname() {return "II_mp";}
  virtual Evaluation* evaluate(Design*);
};
```

190

```
class II_mf: public II_base
{
public:
  II_mf(Role*, Role*,int, Ref_frame*);
  ~II_mf(){}
  inline virtual char* get_classname() {return "II_mf";}
  virtual Evaluation* evaluate(Design*);
};
```

200

```
class II_mi: public II_base
{
public:
```



---

```
II_mi(Role*, Role*,int, Ref_frame*);  
~II_mi(){}  
inline virtual char* get_classname() {return "II_mi";}  
virtual Evaluation* evaluate(Design*);  
};
```

210

```
class II_mb: public II_base  
{  
public:  
II_mb(Role*, Role*,int, Ref_frame*);  
~II_mb(){}  
inline virtual char* get_classname() {return "II_mb";}  
virtual Evaluation* evaluate(Design*);  
};
```

220

```
class II_mm: public II_base  
{  
public:  
II_mm(Role*, Role*,int, Ref_frame*);  
~II_mm(){}  
inline virtual char* get_classname() {return "II_mm";}  
virtual Evaluation* evaluate(Design*);  
};
```

230

```
#endif
```

---

---

```

#ifndef _QSR
#define _QSR

#include <iostream.h>
#include "spatial.h"

class Dobj;           // the class Dobj refers to a spatial
                    // design object corresponding to an artifact
                    // (the spatial component of an artifact,
                    // separated for practical purposes of
                    // efficiency)                                10

class Evaluation;

                    // This file defines the basic QSR class,
                    // and the composite QSR base classes.
                    // The base classes are instantiable directly:
                    // we can thus allow the user to create
                    // arbitrary combinations, as opposed to
                    // using the predefined combinations.                                20

class QSR:public Srel
{
protected:
    Role* source;
    Role* target;
public:
    QSR(Role* = NULL, Role* = NULL, int = Srel::X,
        Ref_frame* = NULL);
    virtual ~QSR(){}                                30
    inline virtual char* get_classname() {return "QSR";}
    inline Role* get_source(){return source;}
    inline Role* get_target(){return target;}
    virtual void improve(Design*);
    inline virtual Evaluation* evaluate(Design*){return NULL;}
};

                    // We now define the two kinds of composite
                    // QSR classes. The disjunctions and the
                    // composite three-D classes                                40

```

---

---

```

class QSR_disjunc : public QSR
{
    // Still along only one axis, as in
    // the primitive QSRs

protected:
    QSR ** comp ;           // components of the disjunction, stored
                           // as an array
    int comp_num ;         // no of elements in the disjunction

public:
    QSR_disjunc(Role*, Role*, int, Ref_frame*);           50
    virtual ~QSR_disjunc() ;
    inline virtual char* get_classname() {return "QSR_disjunc";}
    virtual Evaluation* evaluate(Design*);
};

class QSR_3D : public QSR
{
protected:
    QSR * comp[3];         // one in each of the three directions

public:
    QSR_3D(Role*, Role*, int, Ref_frame*);           60
    virtual ~QSR_3D(){}
    inline virtual char* get_classname() {return "QSR_3D";}
    virtual Evaluation* evaluate(Design*);
};

#endif

```

---

---

```

#ifndef _COMP_QSR
#define _COMP_QSR

#include <iostream.h>
#include "qsr.h"

class II_ji;
                                // this file defines some sample higher level
                                // QSRs which use disjunctions and the
                                // three-D relations
                                10

                                // First the one-D relationships

class Centered: public QSR{
                                // Relationship to specify symmetry of one
                                // object w.r.t another
    II_ji* rel;                  // source must be <in> target
public:
    Centered(Role*, Role*, int, Ref_frame*);
    virtual ~Centered();
    virtual Evaluation* evaluate(Design*);
};
                                20

class Overlap: public QSR_disjunc{
public:
    Overlap(Role*, Role*, int, Ref_frame*);
    virtual ~Overlap(){}
};
                                30

class Overlap_front: public QSR_disjunc{
public:
                                // This class covers the case of overlap
                                // from the front,
                                // NOTE: it includes the flush-contact
                                // case as well
    Overlap_front(Role*, Role*, int, Ref_frame*);
    virtual ~Overlap_front(){}
};
                                40

```

---

---

```
class Overlap_back: public QSR_disjunc{
public:
    // This class covers the case of overlap
    // from the back,
    // NOTE: it includes the flush-contact
    // case as well
    Overlap_back(Role*, Role*, int, Ref_frame*);
    virtual ~Overlap_back(){}
};
```

50

```
class Inside: public QSR_disjunc{
public:
    // what we actually mean when we say inside,
    // if, ii, bi
    Inside(Role*, Role*, int, Ref_frame*);
    virtual ~Inside(){}
};
```

60

```
class Touch_contact: public QSR_disjunc{
public:
    Touch_contact(Role*, Role*, int, Ref_frame*);
    virtual ~Touch_contact(){}
};
```

```
class No_contact: public QSR_disjunc{
public:
    No_contact(Role*, Role*, int, Ref_frame*);
    virtual ~No_contact(){}
};
```

70

// Now the Three D relationships

```
class Abuts: public QSR_3D{
public:
    Abuts(Role*, Role*, int, Ref_frame*);
```

80

---

```
virtual ~Abuts(){}  
};
```

```
class Intersects: public QSR_3D{  
public:  
    Intersects(Role*, Role*, int, Ref_frame*);  
    virtual ~Intersects(){}  
};
```

90

```
class Contains: public QSR_3D{  
public:  
    Contains(Role*, Role*, int, Ref_frame*);  
    virtual ~Contains(){}  
};
```

```
        // Ternary relationships
```

100

```
class Between: public QSR{  
    Role* center_obj ;           // is between source and target  
    QSR_3D* comp[2];  
public:  
    Between(Role*, Role*, Role*, int, Ref_frame*);  
    virtual ~Between() ;  
    virtual Evaluation* evaluate(Design*) ;  
};
```

110

```
class Connects: public QSR{  
    Role* connector;  
    QSR* comp[6];                // cannot directly use any of the threeD  
                                // relationships, unless we define new ones..  
public:  
    Connects(Role*, Role*, Role*, int, Ref_frame*);  
    virtual ~Connects() ;  
    virtual Evaluation* evaluate(Design*) ;  
};
```

120

---

**#endif**

---

---

```

#ifndef _EVAL
#define _EVAL
#include "GNvector.h"

class Dobj ;

class mod_operator
{
    // This simple "class" stores the modification
    // operators suggested by the evaluation of a
    // design. Move the Dobj indicated by obj LEFT
    // or RIGHT along axis, rotate it clockwise or
    // anticlockwise, make the object SMALLER
    // or BIGGER
    public:
    enum{LEFT, RIGHT, CLOCK, ANTICLOCK, SMALLER, BIGGER,
        UNDEFINED, INCREASE, DECREASE};
    int op;
    Dobj* obj ;
    GNvector axis_vec ;           // axis along which the modification
                                // is defined, it is defined here for
                                // convenience, especially useful for 3D QSRs
    //defining the exact amount of change for
    char var[50];                // numerical constraints, the name of
    double change;              // the variable and the change requested

    mod_operator(int, Dobj*, GNvector&);
    mod_operator(int, Dobj*, char*, double = NULL);
    mod_operator(mod_operator&);
    ~mod_operator(){obj=NULL;op=0;change=0;}
};

class Evaluation
{
    // This simple "class" stores an evaluation
    // as recorded by a QSR object.
    int count ;                  // Number of elements in the array
    public:

```



---

```
double eval;
mod_operator** mod_op_arr;    // array of possible modifications,
                               // recorded during the evaluation

Evaluation();
~Evaluation();
Evaluation(Evaluation&);
void delete_ops();
inline int get_no_ops(){return count ;}
void add_op(mod_operator*) ;
inline mod_operator* get_random_op();
};
```

50

```
#endif
```

---