

Application Development in  
A Knowledge-Based Conceptual Generator

by

Johanes Chandra Sugiono

B.S., Northeastern University (1992)  
Boston, MA

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

Master of Science in Civil and Environmental Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

© Massachusetts Institute of Technology 1995. All rights reserved.

Barter Eng

1995

A / 1 S

Author.....  
Department of Civil and Environmental Engineering  
December 8, 1994

Certified by .....  
Duvvuru Sriram,  
Senior Research Scientist, Thesis Supervisor

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

**Application Development in  
A Knowledge-Based Conceptual Generator**

by

**Johanes Chandra Sugiono**

Submitted to the Department of Civil and Environmental Engineering  
on December 16, 1994, in partial fulfillment of the  
requirements for the degree of  
Master of Science in Civil and Environmental Engineering

**Abstract**

Engineering design process can be decomposed into several stages. One of the most important stage is conceptual design. Conceptual design provides a strong foundation on which the subsequent design stages are executed. CONGEN (CONcept GENERator), a conceptual design agent provides the capability of integrating knowledge which supports the decision making process, as well as a constraint management satisfaction scheme and a visualization tool to help the designer overcome the limitations of traditional CAD systems.

In implementing a design application on top of CONGEN, the structural engineering domain is chosen. Structural engineering design provides a good testbed for application development in CONGEN. The existing knowledge and the preliminary computational support provides challenge to CONGEN as the choice platform.

The contribution of this study is to provide complete documentation support for CONGEN. This helps the users to successfully implement various applications. It starts by introducing the users to the basic concepts and structure of CONGEN and gradually moves toward the development of a Cabin Design application utilizing all the capabilities of CONGEN.

The issues addressed by this thesis are the formulation of CONGEN basic concepts in answering the challenges of conceptual design implementation, the step-by-step approach in developing the application, and lastly, the evaluation of CONGEN capabilities and its future directions. In addition, this thesis also supports the basic premise that CONGEN is a flexible system for developing design applications independent of any knowledge domain.

Thesis Supervisor: Duvvuru Sriram

Title: Senior Research Scientist

---

# Acknowledgments

Firstly, I sincerely thank God for the chances that He has given me throughout my life. The works that He has done for me has been amazing and miraculous.

I would like to thank my advisor, Professor Duvvuru Sriram for his support throughout this study and giving me another shot for studying something totally new.

Thank you to Prof. Robert Logcher for the discussions on developing the application. His quest for new ways and directions in Information Technology makes me believe that the changes are only for the better future.

I would like to thank Jen Diann Chiou for his assistance in developing the knowledge base in an area I've never known before.

I also express my gratitude to Prof. Jayachandran, Gorti Sreenivasa-Rao, and Murali Vemulapati for the discussions and support in finishing this thesis.

I also would like to express my thanks to PT. PAL INDONESIA for giving me the opportunity to study at M.I.T. for two years and supporting me financially during my years in USA.

Lastly, to all my KMK friends, the ones who believe or even do not believe in my finishing this thesis, thank you all for the prayers.

---

## Dedications

To my ever-loving, supportive parents and sisters in Indonesia. Thanks for believing in me. For without them, I would not be here at all. It is my turn to take care of you all now.

To my beloved grandparents, I love you both. You are the best grandparents in the world!!

To my guiding light and inspiration, Diany Pranata. I would not have finished this thesis if I never met you here. You made me believe in miracles. We did it!

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>15</b>
1.1	Motivation . . . . .	17
1.2	Objectives . . . . .	18
1.3	Roadmap of the Thesis . . . . .	18
<b>2</b>	<b>Background</b>	<b>21</b>
2.1	Object Oriented Concepts . . . . .	21
2.2	DICE Project . . . . .	23
2.3	COSMOS . . . . .	25
2.4	GNOMES . . . . .	26
2.5	COPLAN . . . . .	27
2.6	CONGEN . . . . .	28
2.7	How to Use this Documentation . . . . .	28
2.8	Summary . . . . .	31
<b>3</b>	<b>CONGEN Application Concepts</b>	<b>32</b>
3.1	CONGEN – CONcept GENerator . . . . .	32
3.2	Knowledge and Design Concepts . . . . .	34
3.3	CONGEN Product Concepts . . . . .	37
3.4	CONGEN Process Concepts . . . . .	38
3.5	Integrating the Concepts . . . . .	40
3.6	CONGEN Application Structure . . . . .	42

## CONTENTS

---

3.7	Summary . . . . .	44
<b>4</b>	<b>Tutorial I: Getting Familiar with CONGEN</b>	<b>46</b>
4.1	Starting a CONGEN Session . . . . .	46
4.2	File Menu . . . . .	48
4.2.1	CONGEN Application Management System . . . . .	51
4.3	Knowledge Menu . . . . .	53
4.4	Specifications Menu . . . . .	57
4.5	Execute Menu . . . . .	58
4.6	Browsers Menu . . . . .	60
4.7	Summary . . . . .	64
<b>5</b>	<b>Tutorial II: A Simple CONGEN Application</b>	<b>65</b>
5.1	The Notation . . . . .	65
5.2	The Problem . . . . .	66
5.3	The Implementation . . . . .	67
5.3.1	Preparation . . . . .	67
5.3.2	Creating a new application . . . . .	68
5.3.3	Creating classes and rulesets . . . . .	69
5.3.4	Making the application . . . . .	75
5.3.5	Creating goals and plans for the application . . . . .	75
5.3.6	Executing the Synthesizer . . . . .	77
5.3.7	Executing the Geometric Modeler (GNOMES) and show the geometry	80
5.4	Other Solutions to the Problem . . . . .	82
5.5	Summary . . . . .	84
<b>6</b>	<b>Tutorial III: Building a Box</b>	<b>87</b>
6.1	The Problem . . . . .	87
6.2	The Implementation . . . . .	91
6.2.1	Preparation . . . . .	91
6.2.2	Creating a new application . . . . .	92

## CONTENTS

---

6.2.3	Creating classes and rulesets . . . . .	92
6.2.4	Making the Application . . . . .	97
6.2.5	Creating goals and plans for the application . . . . .	98
6.2.6	Executing the Synthesizer . . . . .	102
6.2.7	Executing the Geometric Modeler (GNOMES) and show the geometry	108
6.3	Other Solutions to the Problem . . . . .	109
6.4	Summary . . . . .	112
<b>7</b>	<b>Tutorial IV: CABIN DESIGN Application</b>	<b>114</b>
7.1	The Problem . . . . .	114
7.1.1	Process Flow . . . . .	115
7.1.2	Knowledge Acquisition . . . . .	118
7.1.3	Vocabulary . . . . .	121
7.1.4	Geometry . . . . .	124
7.1.5	Analysis . . . . .	129
7.2	The Implementation . . . . .	130
7.2.1	Preparation . . . . .	130
7.2.2	Creating a new application . . . . .	131
7.2.3	Creating classes and rulesets . . . . .	132
7.2.4	Making the Application . . . . .	140
7.2.5	Creating goals and plans for the application . . . . .	140
7.2.6	Executing the Synthesizer . . . . .	153
7.2.7	Executing GNOMES and displaying the geometry . . . . .	159
7.3	Summary . . . . .	160
<b>8</b>	<b>Summary and Future Work</b>	<b>164</b>
8.1	Summary . . . . .	164
8.2	Future Work . . . . .	167
<b>A</b>	<b>REFERENCE MANUAL</b>	<b>176</b>
A.1	Reserved Keywords . . . . .	176

## CONTENTS

---

A.2 Available Methods . . . . .	177
A.3 Dynamic Methods . . . . .	179
A.4 Application Script . . . . .	183
<b>B COSMOS Knowledge Base Rule in CONGEN</b>	<b>185</b>
B.1 COSMOS Rule Grammar . . . . .	185
B.1.1 Explanation of the Grammar . . . . .	186
B.1.2 Comment Block . . . . .	187
B.1.3 Rule . . . . .	187
B.1.4 Rule Name . . . . .	187
B.1.5 Rule Priority . . . . .	188
B.1.6 Condition Block . . . . .	188
B.1.7 Test Expression . . . . .	188
B.1.8 Arithmetic Expression . . . . .	189
B.1.9 Example of Condition Block . . . . .	189
B.1.10 Action Block . . . . .	190
B.1.11 Example of Action Block . . . . .	191
B.1.12 Inline Comment Block . . . . .	191
B.2 Helpful hints in building and running rules in COSMOS . . . . .	192
<b>C Installation, Configuration &amp; Troubleshooting</b>	<b>195</b>
C.1 Installation . . . . .	195
C.1.1 How to Obtain CONGEN . . . . .	195
C.1.2 Requirements . . . . .	195
C.1.3 Pre-installation . . . . .	196
C.1.4 Compiling CONGEN . . . . .	196
C.1.5 Environment Variables . . . . .	196
C.1.6 Database Environment Variable . . . . .	197
<b>D Tutorial 2 &amp; 3 Listings</b>	<b>198</b>
D.1 TUTORIAL 2 - SIMPLE SLAB . . . . .	198



## CONTENTS

---

D.1.1	Main Listing of the rulefile Tutorial-2.rul = Alternative 1 - One Goal, One Plan rulefile. . . . .	198
D.1.2	Alternative 2 - One Goal, One Plan, Two Subgoals . . . . .	200
D.2	TUTORIAL 3 - BOX APPLICATION . . . . .	202
D.2.1	Main Listing . . . . .	202
D.2.2	Alternative - tut3_long.rul . . . . .	209
<b>E</b>	<b>CABIN DESIGN elements</b>	<b>213</b>
E.1	Cabin Artifact Vocabulary . . . . .	213
E.1.1	Structural Member Superclass . . . . .	213
E.1.2	Joint . . . . .	214
E.1.3	Linear Members . . . . .	215
E.1.4	Area Members . . . . .	216
E.1.5	Beams . . . . .	216
E.1.6	Columns . . . . .	217
E.1.7	Piers . . . . .	218
E.1.8	Truss_member . . . . .	219
E.1.9	Truss_system . . . . .	220
E.1.10	Slabs . . . . .	221
E.1.11	Walls . . . . .	223
E.1.12	Wall_opening . . . . .	223
E.1.13	Strip_footing . . . . .	224
E.1.14	Mat_found . . . . .	225
E.1.15	Spread_found . . . . .	226
E.2	Examples of CABIN Rulefiles . . . . .	228
E.2.1	cabin_eff.rul . . . . .	228
E.2.2	set_columns.rul . . . . .	229
E.2.3	set_girders.rul . . . . .	235
E.2.4	set_girders_eff.rul . . . . .	236
E.2.5	set_south_wall_eff.rul . . . . .	238

## CONTENTS

---

E.2.6	set_foundation_eff.rul . . . . .	241
E.3	CABIN calculation rulefiles . . . . .	261
E.3.1	Loading.rul . . . . .	261
E.3.2	Beam.rul . . . . .	264
E.3.3	Column.rul . . . . .	266
E.3.4	Slab.rul . . . . .	267
E.4	CABIN script . . . . .	269
E.4.1	create_cabin.c . . . . .	269

---

# List of Figures

1-1	The organization of this thesis. . . . .	20
2-1	The roadmap of this chapter. . . . .	22
2-2	The collaborative product development environment. . . . .	24
3-1	The organization of this chapter. . . . .	33
3-2	The architecture of CONGEN and the supporting modules within its frame- work . . . . .	34
3-3	The integrated CONGEN's product and process concepts. . . . .	35
3-4	CONGEN Application Architecture. . . . .	45
4-1	The roadmap of this chapter. . . . .	47
4-2	CONGEN Main Console. . . . .	48
4-3	QUIT Warning window. . . . .	48
4-4	NEW Application window. . . . .	49
4-5	RETRIEVE Application window. . . . .	49
4-6	DELETE Application window. . . . .	50
4-7	Design Decomposition Hierarchy EDITOR window. . . . .	54
4-8	GOAL EDITOR window. . . . .	55
4-9	Product Knowledge window. . . . .	55
4-10	CLASS EDITOR window. . . . .	56
4-11	Rule List Selection window. . . . .	57
4-12	Rule File EDITOR window. . . . .	57

## LIST OF FIGURES

---

4-13	Specifications EDITOR window. . . . .	59
4-14	GNOMES Geometric Modeler window. . . . .	60
4-15	Synthesizer window. . . . .	61
4-16	GNOMES Geometric Modeler window with top and bottom covers of an application. . . . .	62
4-17	Artifact Browser window. . . . .	63
4-18	Rulefile Browser window. . . . .	63
5-1	The roadmap of this chapter. . . . .	66
5-2	Creating Tutorial.2 application. . . . .	69
5-3	Slab Attributes entered. . . . .	70
5-4	CLASS EDITOR (PUBLIC) window after attributes have been entered. . .	71
5-5	<i>createslab</i> rule in the RULE EDITOR window. . . . .	72
5-6	Saving <i>createslab</i> Rule in the RULE EDITOR window. . . . .	73
5-7	create_slab Goal shown. . . . .	76
5-8	DDH Editor after everything has been entered and saved. . . . .	78
5-9	SYNTHESIZER window after finished creating the Slab. . . . .	78
5-10	XTERM window after finished creating the Slab. . . . .	79
5-11	GNOMES showing the Slab geometry. . . . .	81
5-12	Other alternative solution to Tutorial.2. . . . .	83
5-13	Other alternative solution to Tutorial.2. . . . .	84
5-14	The technique of combining a dummy plan and a goal to fire a ruleset. . . .	85
6-1	The roadmap of this chapter. . . . .	88
6-2	Tutorial 3 Application structure. . . . .	90
6-3	Slab3 class attributes. . . . .	93
6-4	Assembly class attributes. . . . .	94
6-5	Assembly class Artifact Defaults. . . . .	95
6-6	The create_box root goal for the application. . . . .	98
6-7	The create_box_plan for the application. . . . .	99

## LIST OF FIGURES

---

6-8	The create_northsouth goal for the application. . . . .	100
6-9	The create_northsouth_plan for the application. . . . .	101
6-10	The create_north goal for the application. . . . .	101
6-11	The DDH EDITOR window after everything has been entered. . . . .	102
6-12	The SPECIFICATION EDITOR window after everything has been entered. . . . .	103
6-13	The SYNTHESIZER window with the created instance of Assembly. . . . .	104
6-14	The SYNTHESIZER window after the top cover slab has been created. . . . .	106
6-15	The XTERM window showing all the rulefiles fired. . . . .	107
6-16	The SYNTHESIZER window after every path has been traversed. . . . .	107
6-17	The DDH EDITOR window after every path has been traversed. . . . .	108
6-18	The DDH BROWSER window after <i>create_south.Slab?</i> is pressed. . . . .	109
6-19	The GNOMES window with the box. . . . .	110
7-1	The roadmap of this chapter. . . . .	115
7-2	Cabin Architectural plan. . . . .	116
7-3	Cabin Design Application structure. . . . .	119
7-4	The New Structural Member hierarchy . . . . .	122
7-5	The abstraction of GAB geometry classes in CONGEN. . . . .	127
7-6	Cabin class expanded without using the Cabin-part - notice all the parts linked to Cabin class filling the window. . . . .	133
7-7	Cabin class expanded using the Cabin-part. Cabin-part contains all the instances shown in the preceding figure. . . . .	134
7-8	Cabin class attributes. . . . .	135
7-9	The create_cabin goal description. . . . .	142
7-10	The set_beam_column_grid goal description. . . . .	143
7-11	The set_column goal description. . . . .	143
7-12	The set_beams goal description. . . . .	144
7-13	The set_girders goal description. . . . .	144
7-14	The set_supporting_beams goal description. . . . .	145
7-15	The set_trussys goal description. . . . .	145

## LIST OF FIGURES

---

7-16	The <code>set_trusses</code> goal description. . . . .	146
7-17	The <code>set_purlins</code> goal description. . . . .	146
7-18	The <code>set_walls</code> goal description. . . . .	147
7-19	The <code>set_north_wall</code> goal description. . . . .	147
7-20	The <code>set_south_wall</code> goal description. . . . .	148
7-21	The <code>set_east_wall</code> goal description. . . . .	148
7-22	The <code>set_west_wall</code> goal description. . . . .	149
7-23	The <code>set_foundation</code> goal description. . . . .	149
7-24	The <code>cabin_design_plan</code> description. . . . .	150
7-25	The <code>set_beam_column_grid_plan</code> description. . . . .	151
7-26	The <code>set_beams_plan</code> description. . . . .	151
7-27	The <code>set_walls_plan</code> description. . . . .	152
7-28	The <code>set_trussys_plan</code> description. . . . .	152
7-29	The Synthesizer first pass of the Cabin Design application. . . . .	154
7-30	The Synthesizer beam-column grid pass of the Cabin Design application. . . . .	155
7-31	The Synthesizer pops out the editor for the artifact Cabin of the Cabin Design application. . . . .	156
7-32	The DDH EDITOR window. . . . .	157
7-33	The CONTEXT BROWSER window for the Context: <code>set_columns.4</code> . . . . .	158
7-34	The SYNTHESIZER window for the Context: <code>set_trussys.set_trussys_plan</code> . . . . .	159
7-35	The CONTEXT BROWSER window for <code>set_beam_column_grid.set_beam_column_grid_plan</code> . . . . .	160
7-36	The Geometric Modeler for Cabin configuration with 6 columns, 2 girders, 1 supporting beam, and two trusses. . . . .	161
7-37	The GNOMES window showing configuration with 4 columns, wall openings, and mat foundation. . . . .	162
7-38	The GNOMES window showing configuration with 6 columns, 3 trusses and 4 purlins. . . . .	163
A-1	The HELP window with the <code>create_cabin</code> script entry. . . . .	184

---

# Chapter 1

## Introduction

One of the most challenging tasks performed by engineers is the process of designing products. This task requires large amounts of domain-specific knowledge, experience, and problem solving skills. In addition to the above, the notion of the geometric structure of artifacts comprising the product plays an important role in design. The design process is evolutionary and iterative in nature with increasing details being developed as the design progresses.

The process of solving a design problem typically involves six stages [52]:

1. *Problem Identification* - Describing of which functions the artifact should perform.
2. *Specification Generation* - Providing the constraints of the artifact - spatial, geometrical, and interaction with other artifacts.
3. *Concept Generation* - Synthesizing preliminary design solutions which satisfy key constraints.
4. *Analysis* - Analyzing the design alternatives generated by the former step in detail.
5. *Evaluation* - Narrowing down feasible design alternatives accepted by the designer's intention.
6. *Detailed Design* - Refining the best possible design so that all applicable constraints or specifications are satisfied.

---

The process of design itself is dual natured [11]. The heuristic nature of the design makes the process a very good candidate for the knowledge-based application. The parameters in designing a system are usually complex and must be selected according to the intuition, judgment, and previous expert knowledge. Therefore, a knowledge-based design support application is very suitable in answering the design process challenges.

In addition, as design is an open-ended problem, a vast design alternative space may be produced which demands further process of selecting the best overall design. Sometimes, the experience of an expert designer is not sufficient to select the best design judged from every required criteria. The ultimate choice of a design made by an expert is usually limited by the expert's knowledge. To extend the expert's capability in selecting the best design satisfying all the criteria, a design support tool is needed.

At the conceptual design stage, the important task is to identify and design the artifact to meet the designer's abstract functionality requirements. Within this scope, a system that can support conceptual design from the initial stages is required. This system will help the designers performing the sequential tasks of general arrangement of artifacts to the detailed geometric structure of the artifacts. Current CAD tools require a complete information of the artifact being created in the design process, including the knowledge and the geometric abstractions.

However, traditional CAD tools pose limitations to the designer's capability such as the following [17]:

- CAD systems do not have the ability to capture the essential functional intent of the design because they are very limited in representing the required design detail.
- CAD systems' ability is focused towards representing the geometric aspects of the artifact instead of supporting the conception.
- CAD systems dictate detailed geometric representation which limits the freedom of conceptual design.
- The design domain geometric requirements and the CAD geometric primitives sometimes differ greatly.



## 1.1 Motivation

---

Based on the limitations and the required capabilities, CONGEN (CONcept GENerator) was developed. CONGEN is a knowledge-based conceptual design support system to help designers. CONGEN is able to represent the conceptual design space efficiently and prune the alternatives according to the required specifications. Ultimately, CONGEN helps produce satisfying initial product designs [17].

### 1.1 Motivation

The structural design process is initiated by a need for a safe and a rigid building structure. It ends with an efficient and effective design satisfying all specifications and constraints. The functional description of the design refers to the characteristics of individual artifacts, such as columns, beams, shear walls, and foundations. The first stage of the design process is the conceptual design - generating solution alternatives based on a set of requirements. It is very crucial to provide the flexibility of accommodating the needs of the users in this stage, and support the decision process.

The main motivation of this work is to develop a structural engineering design system within CONGEN's integrated knowledge-base engineering system framework. Another important factor is the lack of documentation for CONGEN application development. In order to successfully develop a full-blown application, the users must understand the basic concepts and familiarize themselves with CONGEN by developing simple applications first and gradually move towards building a real-world design application.. To achieve this goal, we acknowledge a need to provide the user with complete documentation in CONGEN implementation and CONGEN application development framework.

In addition, this thesis addresses the issue of extending and evaluating CONGEN, primarily in the development of a real life application within its framework. CONGEN is expected to fulfill the following Computer Aided Engineering goals:

- Shortening the design process time, saving time and money.
- Offering rapid response to changing environmental conditions such as budgeting, business, social, and political.

## 1.2 Objectives

---

- Enabling effortless adaptation to architectural changes.
- Providing flexibility of dealing with new design changes generated during the design process.
- Superior control of design errors and enhancements by applying expert knowledge to the design criteria.

## 1.2 Objectives

The primary objective of this study is to develop a complete design application in the area of structural engineering using CONGEN. The knowledge from the structural engineering area itself is too vast to be covered in this application. In realizing this obstacle, we decided to focus the application toward developing preliminary structural engineering design elements. However, the basic concepts presented in this application can be applied to more complex building structures.

To accomplish the above primary objective, we decided on the following strategy:

1. Acquire expert knowledge in the area of structural engineering from texts and interviews with a domain expert.
2. Develop a real-life application of integrated structural design utilizing CONGEN's features.
3. Provide complete documentation on the implementation of CONGEN.
4. Evaluate and suggest future enhancements to CONGEN.

## 1.3 Roadmap of the Thesis

This thesis is organized as follows:

- Background of all the modules in CONGEN are discussed in Chapter 2.
- Basic concepts of CONGEN required to build an application within its framework and a detailed overview of the internal structure of CONGEN are presented in Chapter 3.

### 1.3 Roadmap of the Thesis

---

- Chapter 4 gives a flavor of CONGEN's user interface and the basic interaction scheme between CONGEN and the user in the process of application development.
- A simple application development example is the focus of Chapter 5.
- Chapter 6 broadens the scope of the simple application into more complex application structure within CONGEN with utilization of the product-process knowledge structure.
- Chapter 7 provides implementation details of a cabin design application. It also lists the evolving ideas and the procedures of knowledge acquisition.
- Conclusions resulting from this study and recommendations for future work on CONGEN are the subjects of Chapter 8.
- Appendices overview COSMOS capabilities - which supports forward and backward chaining, rulefiles and classes defined in the applications listed in this study, as well as the CONGEN installation procedures.

### 1.3 Roadmap of the Thesis

---

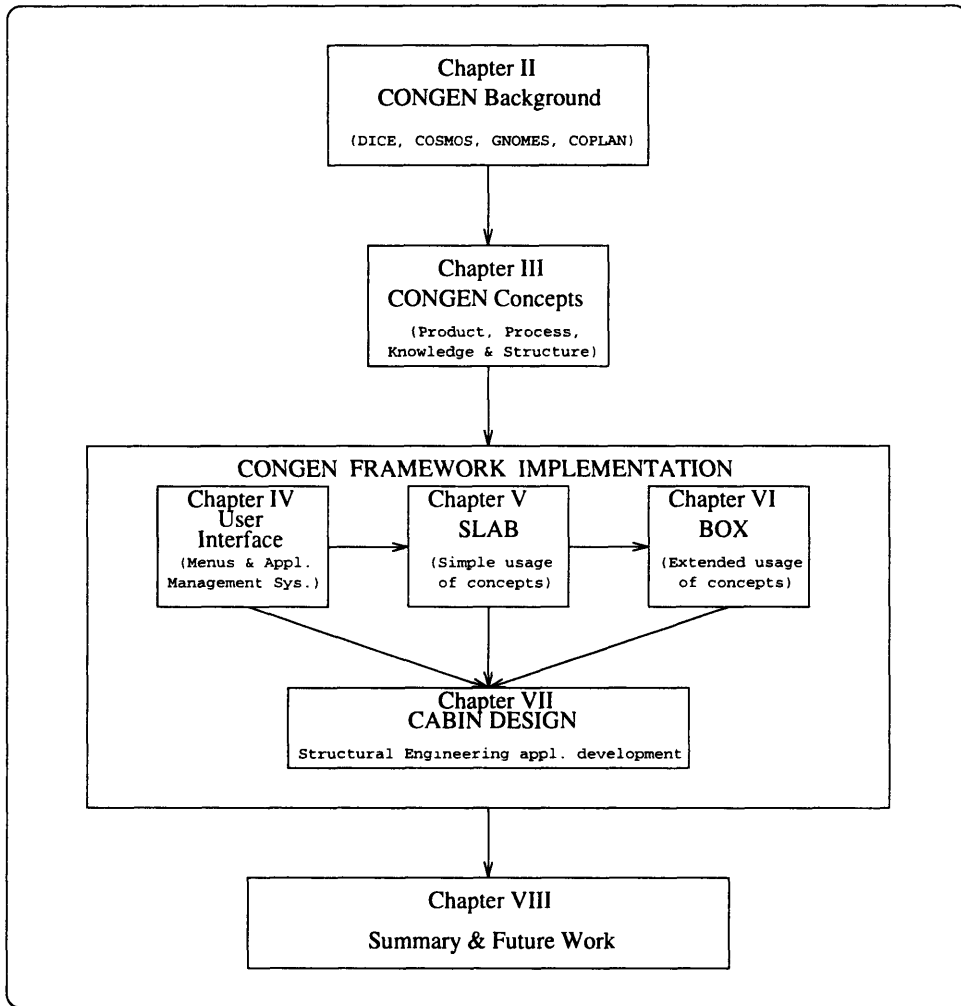


Figure 1-1: The organization of this thesis.

---

## Chapter 2

# Background

This chapter presents a background of the concepts and modules incorporated in CONGEN. Starting with the object-oriented concepts, the chapter continues with an overview of the DICE project, the birthplace of CONGEN and its modules. The subsequent sections provide overviews of the modules used in CONGEN: COSMOS, GNOMES, and COPLAN. Finally, a guide to use this documentation on CONGEN is provided in the last section.

### 2.1 Object Oriented Concepts

The object oriented paradigm is a philosophy of programming which involves the use of objects and messages. Objects are entities that combine the properties of procedures and data, since they can perform computation and save their own local state [48]. Messages are sent between objects to inform the target object to perform specific operations according to the logic of the application

Objects become unique when they each have different object ids. A unique object contains attributes which distinguish it from other objects. The attributes consist of data and methods that shape the object's behavior and properties. The object's methods are summoned via message passing between the objects.

The object oriented programming concepts have many advantages, but the major ones are that it allows reusability, is flexible in changing problem specifications, allows easy

## 2.1 Object Oriented Concepts

---

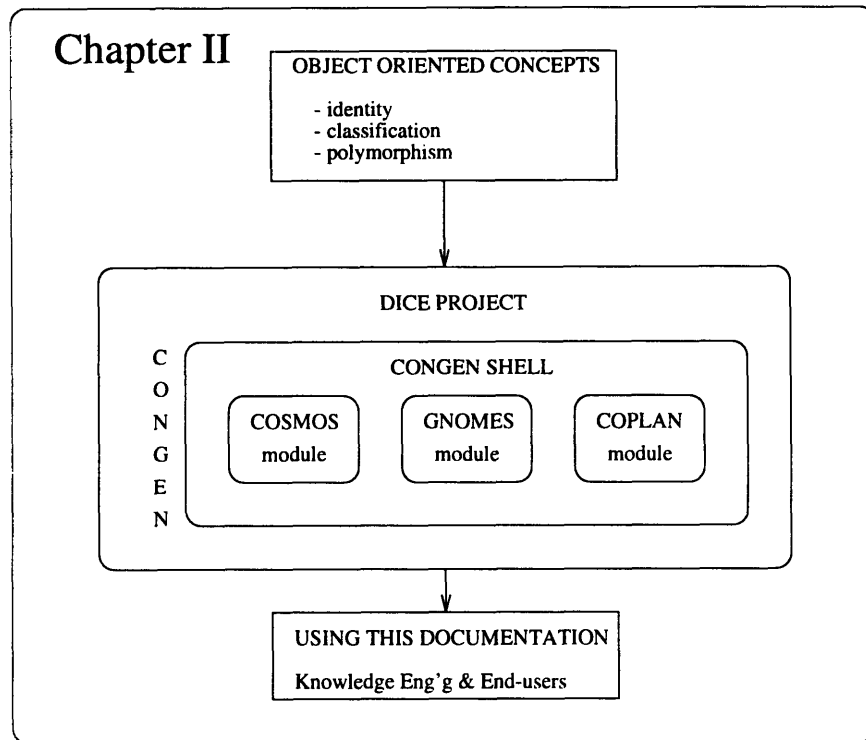


Figure 2-1: The roadmap of this chapter.

expansion and can easily model real-world concepts.

An object-oriented application design usually has the following behaviors [35]:

1. *Identity.* Objects in the design are unique entities which have a special identity assigned to each object.
2. *Classification.* A group of objects with the same properties and behavior are defined as a class. A class also functions as the template of the objects it refers to. A unique object which belongs to a class is called an instance of the class. Classes, however can have relationships with one another in several ways:
  - (a) *is-a* - describes the relationship when one class is a subtype of the other, for example: a Whale is a kind of Mammal.
  - (b) *part-of* - describes the relationship when one class is a composition of the others, for example: Wheels are part-of a Bicycle.

## 2.2 DICE Project

---

3. *Inheritance.* This behavior allows the sharing of behaviors and properties based on the *is-a* relationship. A class inherits properties of another class (parent class) when it becomes a subclass of the parent. Moreover, the subclass can modify the properties inherited from the parent with its own unique properties.
4. *Polymorphism.* Methods with the same name in two different classes inherited from the same parent may have different behaviors. For example, the class Mammal has a method Movement. The Mammal's subclass Whale and Cow inherit the Movement method. Whale's Movement method and Cow's Movement method behaves differently, because a Whale moves in the water while a Cow moves on the ground.
5. *Reusability.* Classes can easily be extended in the future by the Inheritance and Polymorphism mechanisms.

The object-oriented methodology is a powerful modeling tool in a complex engineering information system because of its capability in modeling real-world system. Code reusability and system extendibility provide key advantages for using object-oriented methodology for developing engineering information systems.

## 2.2 DICE Project

The engineering product process involves several stages. The success or failure of the project, however, lies in the proper collaboration between various engineering disciplines. The coordination task between the designers, the engineers, and the builders poses a big challenge to the success of the process.

Recent studies have shown that a collaborative effort during the entire life cycle of the product results in reduced development times, fewer engineering changes, and better overall quality [45]. Figure 2-2 depicts a computer-based collaborative engineering development environment that is being developed at MIT to address the collaborative engineering paradigm. This environment called DICE (Distributed and Integrated environment for Computer-aided Engineering). DICE project has the following goals[44]:

## 2.2 DICE Project

---

- Facilitating effective coordination and communication in various disciplines involved in engineering;
- Capturing the process by which individual designers make decisions, such as what information should be used, how to use it, and what it creates;
- Forecasting the impact of design decisions on various engineering fields;
- Providing an interactive interface to the designers for the detailed manufacturing process or construction planning; and
- Developing design agents to illustrate the approach.

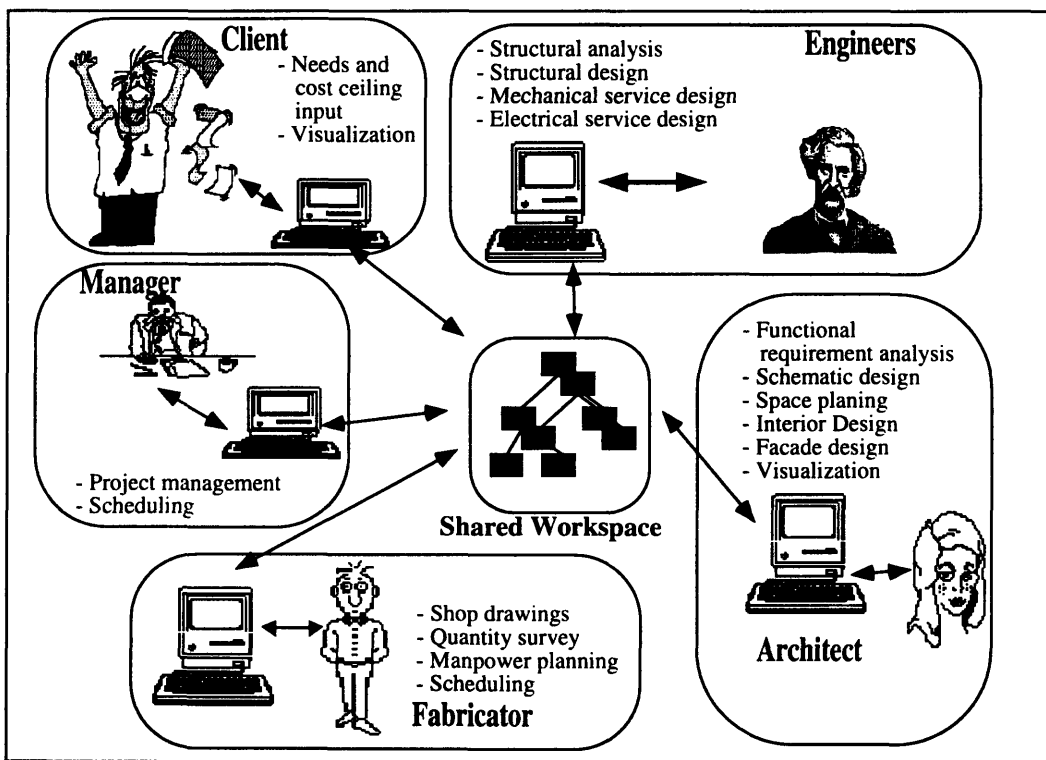


Figure 2-2: The collaborative product development environment.

DICE project was developed as a network of collaborating design agents where the communication and the coordination of the design tasks are achieved through a global database and a control mechanism. Several domain independent shells have been built as a



## 2.3 COSMOS

---

part of the DICE initiative:: COSMOS - a knowledge base system, GNOMES - a geometric modeler, COPLAN - a constraint manager, and CONGEN - a design shell. These systems are reviewed below.

### 2.3 COSMOS

COSMOS is an object-oriented knowledge base system development tool. It is geared primarily for engineering applications. COSMOS integrates rule-based and object-oriented design to provide a robust framework for processing and developing procedural and heuristic knowledge [42].

COSMOS contains the following modules:

- The Object Manager module manipulates objects in-core and on disk. The module was developed using an object-oriented database EXODUS, thus providing a persistent storage. The module are responsible for the maintenance of all classes and instances created at run time. It also provides the information exchange facilities of rule files, classes, and instances between the user interface and the database.
- The Application Manager keeps a simple record of the information about the applications in the database. It also supports a series of functions to store and retrieve this information.
- The Class Manager handles all the information exchange between the user interface and the database, regarding the C++ classes defined by knowledge base developers.
- The Instance Manager takes the responsibilities of managing the instances created by the end users. The instance manager is used mainly for setting or changing the attribute values of an instance.
- The Rule Managers manages the operations of rule files and rules in the system. They are different from the three managers discussed above. The names of the rule file and the rules are stored in the data base, but the rules themselves are stored as ordinary

## 2.4 GNOMES

---

text files in the current working directory. The managers simply scan the working directory and get all the information through the COSMOS' parsing mechanism.

- The Parser accepts and parses the input typed in the Class Editor and Rule Editor by the knowledge engineers. As its output, the Parser generates two data structures used by the Inference Mechanism. The first data structure is an inference network that is used by the backward chaining (BC) mechanism. The second data structure is an intermediate data structure which is used by the RETE network building algorithm of the forward chaining (FC) mechanism.
- The Inference Mechanism in COSMOS supports forward and backward chaining mechanisms in a unified framework. The forward chaining mechanism uses a modified object-oriented RETE network algorithm which has several advantages over other RETE implementations. The backward chaining mechanism is an object-oriented implementation of the KAS inference network, and incorporates a Bayesian network propagation algorithm [42].

## 2.4 GNOMES

GNOMES is a non-manifold geometric modeler. It is a fully functional solid modeler. It can represent and manipulate design entities at various levels of abstraction during the design evolution. A solid modeler is an important part of a CAD system which displays, manipulates, and applies various useful analyses and operations, including mass property calculations, object translations, and Boolean operations, to the design objects [21].

GNOMES was developed using the object-oriented approach paradigm. This design enables the system to be maintained and extended easily. The object-oriented design of GNOMES needed an implementation of an Object-Oriented Database (OODB) environment which allows concurrent access to provide flexibility for the design group. The OODB Management System accommodates a powerful media for modeling, coordination, storage, and manipulation of engineering information [46].

Moreover, the OODB also provides the superior capability of better design process

## 2.5 COPLAN

---

concurrency than other traditional database architectures because of its semantic content. GNOMES exploits EXODUS OODB system capability in sharing the common data and information with other modules, such as COSMOS, and CONGEN [16].

## 2.5 COPLAN

COPLAN is a constraint satisfaction module which uses the Asynchronous Teams of Agents (ATEams) approach developed at Carnegie Mellon University. Murthy defines the ATeams concept [30]:

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

This approach stresses on the importance of the evaluation and improvement tasks to solve a design problem. The implementation of ATeams in COPLAN also uses the object-oriented paradigm. This provides the flexibility and expandability of the module to model real-world systems. CONGEN utilizes ATeams’ capability to evaluate qualitative constraints, such as the 3D relationships between objects in an application [17].

COPLAN manages the constraints generated during design generation process. These constraints can be resource restrictions, relationships between design objects, or restrictions due to the design context conditions [19].

To support the above scheme, the COPLAN constraint management includes:

- A constraint representation language;
- A consistency verification scheme;
- An instantiation technique for feasible solutions;
- A constraint addition handler; and
- A parametric modification manager.

## 2.6 CONGEN

---

### 2.6 CONGEN

In an engineering environment, geometry plays a key role through the various stages of the product evolution. The part identification process and the fulfillment of the required functionality of each part contribute to the initial, yet critical stage of engineering design. Thus, a major portion of the designer's effort is spent on deriving the geometric structure of the objects.

Though traditional CAD systems tried to work around this obstacle, their own limitations restrict the designer's freedom [17]. Therefore, in order to expand the designer's capability to meet the requirements of geometric arrangements and overall system functionality, CONGEN was conceived as a design support system for the conceptual design process.

CONGEN integrates all the base modules in the DICE system and acts as the uppermost application layer to provide all the individual modular systems of DICE project. CONGEN can also be used to build design agents using all the modules within CONGEN framework.

CONGEN synthesizes the design stages by using product block arrangement to achieve the final design. The synthesis itself depends on the integration of three problem-solving approaches: a functional decomposition, a product-oriented model, and constraint propagation approaches [17]. This integration will support the designer's flexibility and alternatives in generating the preliminary designs.

### 2.7 How to Use this Documentation

As a design support system, CONGEN incorporates the domain knowledge and geometric primitives needed for design. In addition, it includes an OODBMS back end. The CONGEN architecture enables its users to define their problems in the corresponding domains, and the usage of the knowledge base to solve the problems. Hence, there are two different kinds of users in CONGEN:

1. *Knowledge Engineers (KEs)*. The personnel who tackles the task of providing the basic product and process knowledge and generating an application in an appropriate

## 2.7 How to Use this Documentation

---

domain.

2. *End-Users (EUs)*. The group of people who use the application to solve specific domain problems.

Although both roles can be played by a single person or group, we need to clarify the difference in the two roles and what is expected from each group.

In using CONGEN effectively, KEs must know how to structure the knowledge for the end-user. The tasks include:

1. *Developing the application.*
2. *Defining and developing the classes.*
3. *Developing the rulesets.*
4. *Setting up the process links which includes understanding the concepts of Goals, Plans, and Artifacts, and how they interact with each other.*
5. *Setting up the relationships and geometry capability of GNOMES.*

The role of the KE is very critical because the flexibility and the capability of the application depends on the knowledge defined by the KE. If the KE hardcodes the knowledge, then the end-user cannot generate expected alternatives from the application.

On the other hand, the end-users must be able to:

1. *Define the problem clearly to the Knowledge Engineer to avoid future ambiguity of the problem statement.*
2. *Provide the requirements and specifications of the problem.*
3. *Provide the constraints and relationships of the application parts.*
4. *Understand the basic concepts of:*
  - Classes and Rulesets
  - Goals, Plans and Artifacts

## 2.7 How to Use this Documentation

---

- Synthesizer and Geometric Modeler

5. *Become well-versed in navigating CONGEN's user interface.*

Within these two distinct scopes, there are two parts of CONGEN structure to be dealt with each kind of user. The Knowledge Engineers must deal with:

1. Knowledge Problem Definition,
2. Knowledge base Development, and
3. Multiple Views of Knowledge Acquisition Process.

whereas the end-users must understand the importance of:

1. End-user Problem Definition,
2. Application Flow, and.
3. Notes about CONGEN implementation.

This documentation is divided into two parts: 1) Basic concepts and implementation of CONGEN; and 2) The tutorial samples. Both groups (i.e., KEs and EUs) are expected to know the basic concepts of CONGEN, and perform the tutorial samples.

The tutorials are tailored to make the user fully utilize the capabilities of CONGEN in a gradual manner. Specifically, each tutorial are organized as follows:

- **Tutorial I** provides an overview of CONGEN's user interface, including how to navigate through the menus, how to start CONGEN, and how to enter values.
- **Tutorial II** presents a simple geometrical problem to the user and how to solve it using CONGEN.
- **Tutorial III** presents an intermediate problem to the user, which includes multiple goals and plans. This tutorial also implements basic CONGEN application structure.
- **Tutorial IV** elaborates on an advanced, real-world CONGEN design support application - CABIN DESIGN.

## **2.8 Summary**

---

After going through the tutorials, the users can be expected to develop their own application based on the examples and concepts presented.

## **2.8 Summary**

This chapter presented the overview of all the modules composing CONGEN. The various capabilities of the modules provide a general survey of CONGEN's capabilities as a stand-alone integrated system. The following chapter will elaborate on the important concepts employed in CONGEN.

---

## Chapter 3

# CONGEN Application Concepts

This chapter presents an overview of the capability of CONGEN in more depth. Additionally, it also explains about different concepts used throughout CONGEN. The concepts span from knowledge base to product and process hierarchies. Each concept is defined and explained thoroughly in the chapter. Section 3.2, section 3.3, and section 3.4 provide the basic structures of CONGEN's application development framework. Section 3.6 provides a template of a CONGEN application for users who want to develop their own applications.

### 3.1 CONGEN – CONcept GENERator

As explained in the previous chapter, CONGEN consists of a layered knowledge-base system, a context mechanism manager, and a friendly user interface. Specifically, CONGEN combines together the capabilities of:

- **Knowledge Representation.** The CONGEN design environment integrates together the knowledge-intensive nature of engineering process and the judgments made according to the knowledge. This knowledge contains the design products, the design processes, and the interrelationships between products and processes [17]. CONGEN provides the design knowledge representation scheme, maintains the design alternatives, and supports design as a synthesis process.



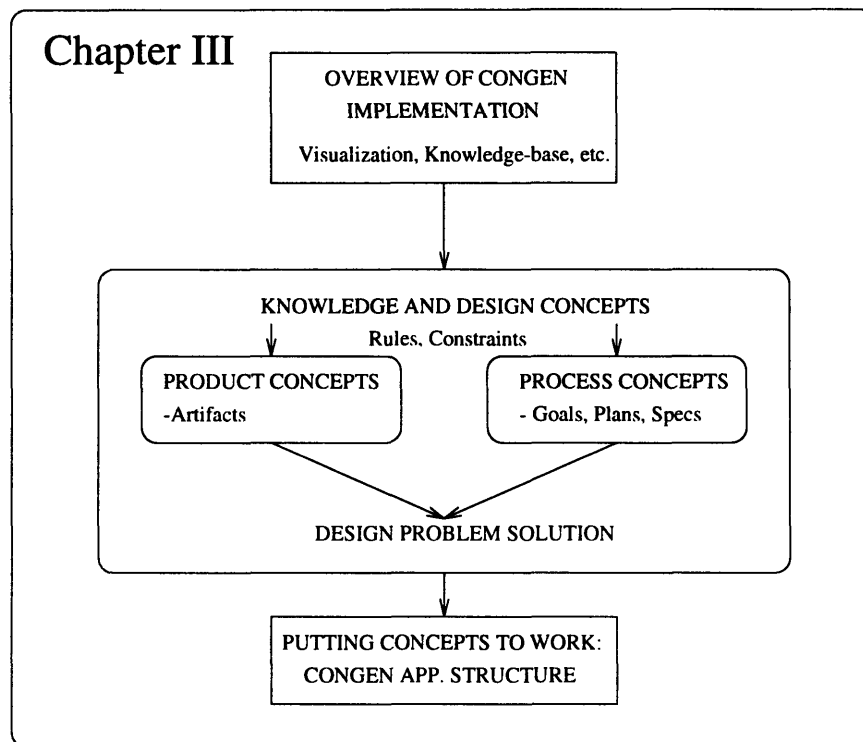


Figure 3-1: The organization of this chapter.

- **Visualization Support.** CONGEN provides the capability of supporting form evolution and the visual interface to the designer. This is done via GNOMES, the non-manifold geometric modeler.
- **Problem Solving Support.** CONGEN incorporates various problem solving approaches, such as top-down refinement, bottom-up reasoning, constraint propagation, etc. COSMOS, the object-oriented expert system shell, and COPLAN, a constraint satisfaction framework contribute to this capability.
- **Database Support.** CONGEN was built on a robust object database system provided by the University of Wisconsin, Madison (EXODUS). EXODUS provides facilities for persistency and maintenance of design data, alternatives and histories, etc.

The architecture of CONGEN with all the modules is shown in figure 3-2

### 3.2 Knowledge and Design Concepts

---

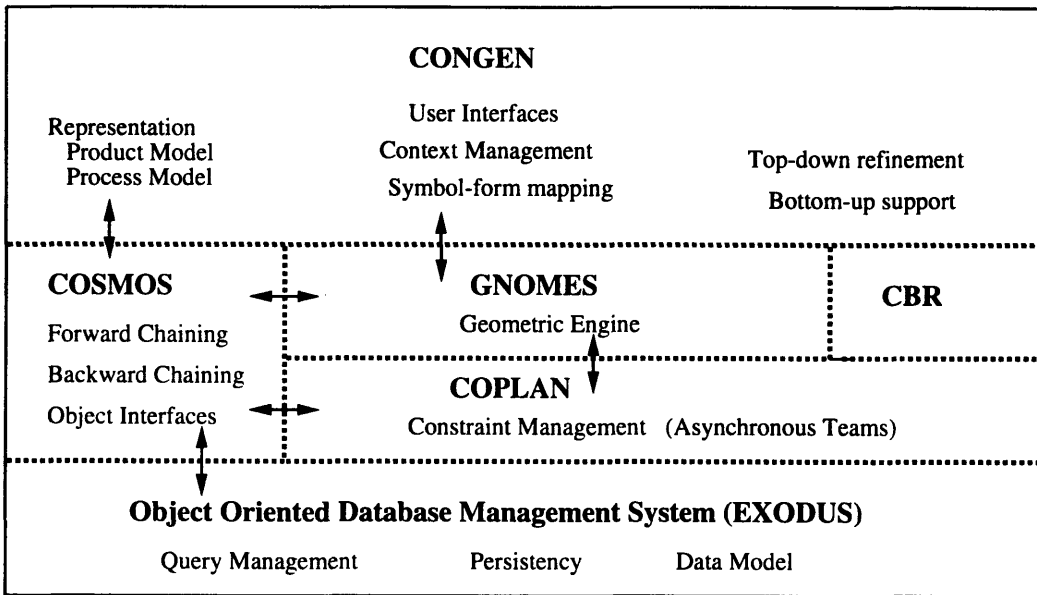


Figure 3-2: The architecture of CONGEN and the supporting modules within its framework

### 3.2 Knowledge and Design Concepts

A definition of the design can be found in [52]:

“Design involves specifying a description of an **Artifact(s)** satisfying constraints arising from a number of sources by utilizing diverse sources of knowledge.”

Understanding the concept of design is very essential in performing engineering tasks. While engineering design essentially maps a function to a physical structure, there are many other aspects involved in the design process.

An expert usually has enough knowledge to direct the pruning of many search space branches into a single one which is most feasible and requirement-satisfying. This task becomes more complex as A relationship between the design space, the knowledge base, and the design process which departs from this analogy, is as follows:

“A design process operation such as refinement, patching, or optimization may generate a new point in design space from one or more old ones;...may

### 3.2 Knowledge and Design Concepts

involve creating new instances of design object classes from the design knowledge base [52]”.

CONGEN is a knowledge-based design tool which provides support for three design steps of:

- Identifying the design task - the user side of design.
- Configuring and instantiating the design process model - CONGEN.
- Implementing and evaluating the design process model - CONGEN Knowledge-Based Expert System.

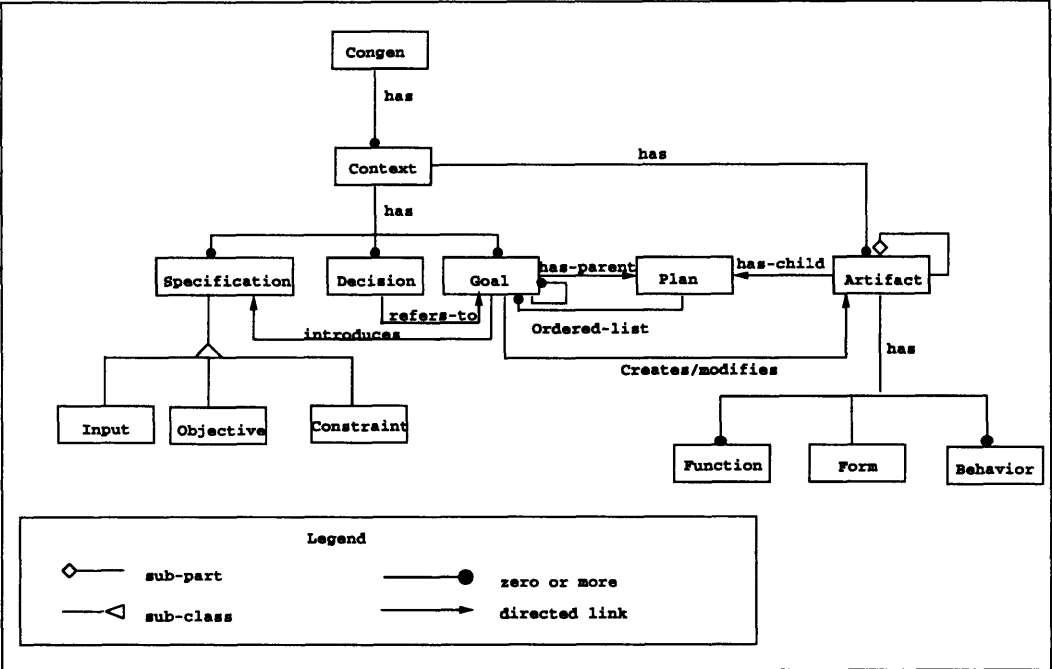


Figure 3-3: The integrated CONGEN’s product and process concepts.

In representing the knowledge, CONGEN utilizes COSMOS knowledge-based system. COSMOS provides the flexibility to represent the knowledge via if-then rules. Before continuing with CONGEN’s design concepts, it is important to define some of the basic concepts of knowledge base implemented in COSMOS, they are:

## 3.2 Knowledge and Design Concepts

---

- **RULESET.** The set of knowledge dependent rules acts as the decision maker of a specific problem. The **Rulesets** depend on the knowledge engineer's capability to refine and analyze different choices at distinct levels in the design process. The **Rulesets** are defined as part of the COSMOS expert system shell in CONGEN. An example of a rule is as follows:

```
(RULE: ProblemDeadBattery1 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
((init_problem == "starting_system") AND
(headlights == "dim" ) )
)
THEN (
(MODIFY (OBJ:$x
(problem " has a dead battery")
)10000 0.001)
)
COMMENT:"If the car's problem is still unknown, but we know that there is
a problem with the starting_system and dim headlights, then we
can conclude that the car has a dead battery.")
```

- **CONSTRAINT.** The Constraints in CONGEN help in narrowing down the design search space. All artifacts are connected to constraints, such as size, time, cost, etc. The constraints restrict the values of design parameters. The constraints in the design can be categorized into [13]:

1. **Synthesis.** Synthesis constraints are associated with individual decision making in the design stages. Synthesis constraints apply to decisions with a finite set of discrete choices. They are mainly within the domain of symbolic values.
2. **Parametric.** Parametric constraints are associated with the design parameter which represents the numerical quality of the artifact.
3. **Interaction.** The design of complex systems is usually decomposed into smaller design goals. Although they are separated, the subsystems may not be independent. As the subsystems interact with each other, the interaction constraints

### 3.3 CONGEN Product Concepts

---

govern the process of interaction by imposing constraints on information flow, compatibility of materials and behavior, and geometric alignment constraints.

4. *Causal.* The Causal constraints refer to the equilibrium equations as well as the compatibility, constitutive, and other relationships based on the physics of the problem.

CONGEN utilizes the Asynchronous Teams of Agents constraint management module from COPLAN for solving constraint satisfaction problems.

### 3.3 CONGEN Product Concepts

Within the CONGEN environment, it is important to define a flexible means of encapsulating the designer's product information and requirements. The design products are actually the mapping of functional description to concrete objects. The functional descriptions provide a clear requirement of the artifact's behavior and constraints. To produce the best design alternative, the functional descriptions have to be decomposed hierarchically. CONGEN's main product concept is :

- *ARTIFACT.* All user defined classes in CONGEN are derived automatically from the **Artifact** class. An **Artifact** is a part of the overall design system which contains information about:
  1. the functionality,
  2. the form (geometry information), and
  3. the behavior of the part.

An **Artifact** can be linked explicitly to a child **Plan** expanding into more subtasks in the design process. A **Plan** is a sequence of design decisions which will be explained in depth in the next section. In the event that an **Artifact** has parts that are not defined explicitly by the user in the class definition, the method **make\_part** invoked via a rule should be used to link an **Artifact** to its parts.

### 3.4 CONGEN Process Concepts

---

The **Artifact** itself contains the information about the function, form, and behavior of the intended design. However, an **Artifact** does not always represent the overall design intention. A primitive **Artifact** may have its own functional description regardless of the **Context** in which it is created. For an assembly comprising of more than one **Artifact**, a design **Context** is needed. The design **Context** will involve the more complete knowledge and decisions required to integrate different **Artifacts**. The integration is needed in order to satisfy the functional descriptions of the overall design system. **Contexts** integrating the concepts of product and process will be discussed in depth in the following section.

### 3.4 CONGEN Process Concepts

CONGEN effects the handling of the design iteration process via the concepts of **Plans**, **Goals**, and **Specifications**. The process concepts provide the knowledge required of the synthesized **Artifacts** to be integrated as a functional entity. As a designer generates the alternatives, it is very crucial that information created within the steps can be analyzed and recorded. CONGEN's process concepts are:

- **CONTEXT**. This class represents the information and data generated during the design process. For example, a new **Context** is created whenever a new alternative is generated at a decision point. A **Context** contains the product information, such as the **Artifacts**, the design relationships, the decisions, and the design **Specifications**. The **Contexts** provide the flexibility for the designer to change from one alternative to another and analyze them for further requirement fulfillment.
- **GOAL**. This class illustrates the design objective, which can also be a decision point for a design. A **Goal** can:
  1. expand for further design **Plan**;
  2. create a new **Artifact**;
  3. modify an existing **Artifact** / set the attributes of an **Artifact**; and

### 3.4 CONGEN Process Concepts

---

4. invoke a function externally, or defined within a class.

In essence, **Goals** represent the functions needed in the functional hierarchy. They provide flexibility to the design to define the functions more clearly. Moreover, they enable the user to focus the attention on the abstraction of the design (**Artifacts**) or the refining of the design subtasks. To achieve the needed functionality, **Goals** may have knowledge associated with them to control the decision making process.

Moreover, the **Goal** can have two kinds of **Rulesets** associated with it - *the selection or choice Rulesets* and *the consequence or effect Rulesets*. The *choice Rulesets* are built to direct the decision-making process of the choice given to the **Goal**, whether it is an attribute, an **Artifact**, or a **Plan**. On the other hand, the *consequence Rulesets* are built to modify or expand the process after the choice has been made. For example, if a designer wants to provide a set of criteria for choosing a material, the designer must first prepare the *choice Rulesets*. After the user makes the choice, the *consequence Rulesets* will be fired.

- **PLAN**. This class organizes the ordered sequence of **Goals** as part of the design task. Basically, a **Plan**'s role is to associate the product information (**Artifacts**) with the process hierarchy. A **Plan** can be associated with an **Artifact** or a **Goal**. In addition, a **Ruleset** can also be attached to a **Plan**. This **Ruleset** can change the **Goal** ordering according to some specific requirement. The order of the **Goals** in a **Plan** can also be changed manually by the user in the course of the design process.
- **SPECIFICATION**. This class contains all the **Specifications** necessary for a particular alternative. **Specifications** are given by the user depending on the **Context** in which the **Specifications** are built. They are used primarily to provide user defined constraints on the **Artifact** relationships, the geometric representation, and the attributes. The **Specification** can be directed towards a specific instance of a class, or all the instances of a class. A new **Specification** instance is also created whenever a user changes an object directly through the instance editor.

### 3.5 Integrating the Concepts

---

- *DECISION*. This class refers to all user decisions which control the flow of the design process. The decisions contain the design alternatives for a **Goal** in a specified **Context**. More importantly, each decision made instantiates a new design **Context** with a different design alternative. As multiple decisions spawn multiple **Contexts**, CONGEN allows the user to pursue multiple design alternatives simultaneously.

### 3.5 Integrating the Concepts

All the concepts laid in the previous sections contribute to the building of a design application shown in the figure 3-3.

A **Context** represents a specific design alternative. The **Context** itself consists of design tasks (**Goals**) referring to the particular design alternative, design flow control (**Plans**), user-defined **Specifications**, decisions made in the process, and the **Artifacts** created within the process. Each corresponding element in the **Context** is needed to satisfy the overall design functionality, such as the function, form and behavior of the design primitives as well as the decisions and the hierarchy of the design process.

The **Contexts** are managed through the Synthesizer. The Synthesizer allows a set of **Specifications** or problem-specific constraints navigate through different satisfying design alternatives or solutions. After defining all the domain knowledge, the Synthesizer also acts as the driving engine to pursue the choices based on the given knowledge.

One simple example is the process of building a box using CONGEN. Logically, the sequence of tasks of building a box is to define the constraints and **Specifications**, build the parts (covers), and lastly, to assemble all the parts which form a box.

This process can be synthesized as follows:

- *INITIATE PROCESS*
  - *Input constraints and initial specifications*
  - *Create the initial assembly*
- *CREATE PARTS*



### 3.5 Integrating the Concepts

---

- *Create top-bottom covers*
- *Create east-west covers*
- *Create north-south covers*
  
- *ASSEMBLE PARTS*
  - *Move top-bottom covers to the geometry placement*
  - *Move east-west covers to the geometry placement*
  - *Move north-south covers to the geometry placement*
  
- *SHOW TO USER*

The preceding steps can be mapped directly onto the CONGEN concepts with some minimal changes:

- *ROOT GOAL - Create Assembly of parts*
  - *Rulesets - Input constraints and specifications*
  - *Rulesets - Create the initial assembly*
  
- *PLAN - CREATE PARTS (has 3 Goals)*
  - *GOAL - Create top-bottom covers*
    - \* *PLAN - Create top-bottom covers (has 2 goals)*
      - *GOAL - Create top cover*
      - *GOAL - Create bottom cover*
  - *GOAL - Create east-west covers*
    - \* *PLAN - Create east-west covers (has 2 goals)*
      - *GOAL - Create east cover*
      - *GOAL - Create west cover*
  - *GOAL - Create north-south covers*

### 3.6 CONGEN Application Structure

---

- \* *PLAN - Create north-south covers (has 2 goals)*
  - *GOAL - Create north cover*
  - *GOAL - Create south cover*
- *Rulesets - PLACE COVERS AND SHOW GEOMETRY*
  - *Rulesets - Move top-bottom covers to the geometry placement*
  - *Rulesets - Move east-west covers to the geometry placement*
  - *Rulesets - Move north-south covers to the geometry placement*

The preceding example is a simple exercise in structuring the knowledge and the constraints to build a simple geometric form. In the following chapters, more examples will be provided and more features of CONGEN will be presented.

### 3.6 CONGEN Application Structure

As explained previously, the domain knowledge in CONGEN is represented by **Plans**, **Goals**, **Artifacts**, **Constraints**, and **Functions**. CONGEN was based on a formal object model SHARED defined in [56]. These elements represent the engineering design knowledge which involves mapping specified functions onto the physical structure of products or **Artifacts**. Based on the complex needs of design, a model of design product and process is thus needed to satisfy the conditions of the designer.

Modeling the design process of an application is the same as applying logical steps in performing the tasks such as those presented in the previous section. However, the CONGEN internal structure must also be understood in order to combine the subtasks with the CONGEN knowledge system.

First of all, the application must define the product and process knowledge. The product consists of the classes and the rulefiles. On the other hand, the process knowledge consists of the **Specifications**, the **Goals**, the **Plans**, and the **Artifacts**.

To form the product knowledge, the application developer must first decompose the final design system into smaller subsystems if applicable. For example, a box must be

### 3.6 CONGEN Application Structure

---

decomposed into covers, and covers can also be decomposed into inner and outer layer, and so forth. The task of breaking down the final design product proves to be very essential because the simpler the base classes, the more information can be embedded into them.

After decomposing the subclasses forming the design system, the rules must be written. However, providing the knowledge through the **Ruleset** is an extremely cumbersome task. The complexity of the application depends on the depth of the domain knowledge. CONGEN provides freedom from limitations in the task of providing knowledge to the process. The developer must realize that CONGEN is a conceptual design support system. CONGEN was not created to support the detailed design of an **Artifact** with all the supporting modules. CONGEN was geared mainly towards assisting the designers in the field of conceptual design.

The next step is to choose a root **Goal** as the top level **Goal** to execute the Synthesizer. The root **Goal** acts as the entry point into the design process. It also creates the initial **Context** in which all the input from the user is created. A good naming practice for the root **Goal** is to use the application purpose as the name of the root **Goal**. For example, the root **Goal** of a bridge design application should be *build\_bridge*, or the root **Goal** of a cabin design application should be *build\_cabin*.

The root **Goal**'s role is usually the creation of an incomplete **Artifact** of the final product class with preliminary information. Even though this practice is not required, the application flow should start with a preliminary design system and finish with the final design product along with detailed information. Along with the final product class, the developer must provide another class to contain the parts created in the process.

An implementation example of the above procedure is as follows: when a box containing six covers is instantiated, all the **Artifacts** in the Synthesizer window are shown with links when they are attached using *make\_part* method. If there are less than ten parts attached to a class, it is still acceptable. If there are tens, or even hundreds of **Artifacts** attached to a class, the Synthesizer window will be too cluttered and hard to navigate. Therefore, a container class is defined in the Cabin Design application - "Cabin\_part" class acting as the container and entry point for all the parts attached to the "Cabin" class.

### 3.7 Summary

---

In designing the application itself, the developer must remember that **Goals** can be expanded into **Plans**, and **Plans** can contain a sequence of **Goals**. Therefore, to create a hierarchy of **Plans** and **Goals**, the developer must keep in mind that whenever there is more than one **Goal** to be achieved, a **Plan** has to be used. In addition, a **Plan** can be derived from a **Goal** to provide flexibility to the design process. Be advised that the sequence of **Goals** within a task must be performed from left to right whenever they are expanded in the Synthesizer window. An extensive example of the **Goal-Plan-Artifact** application structure can be seen in figure 3-4. We suggest that the user follow the structure shown in the figure to develop an application in CONGEN.

### 3.7 Summary

This chapter has given a brief introduction about the important implementation aspects and various facilities of CONGEN. Knowledge information is represented by COSMOS' knowledge-base system. Meanwhile, the product, and the process concepts provide the flexibility for the users to define, solve, and give alternatives to their problems.

### 3.7 Summary

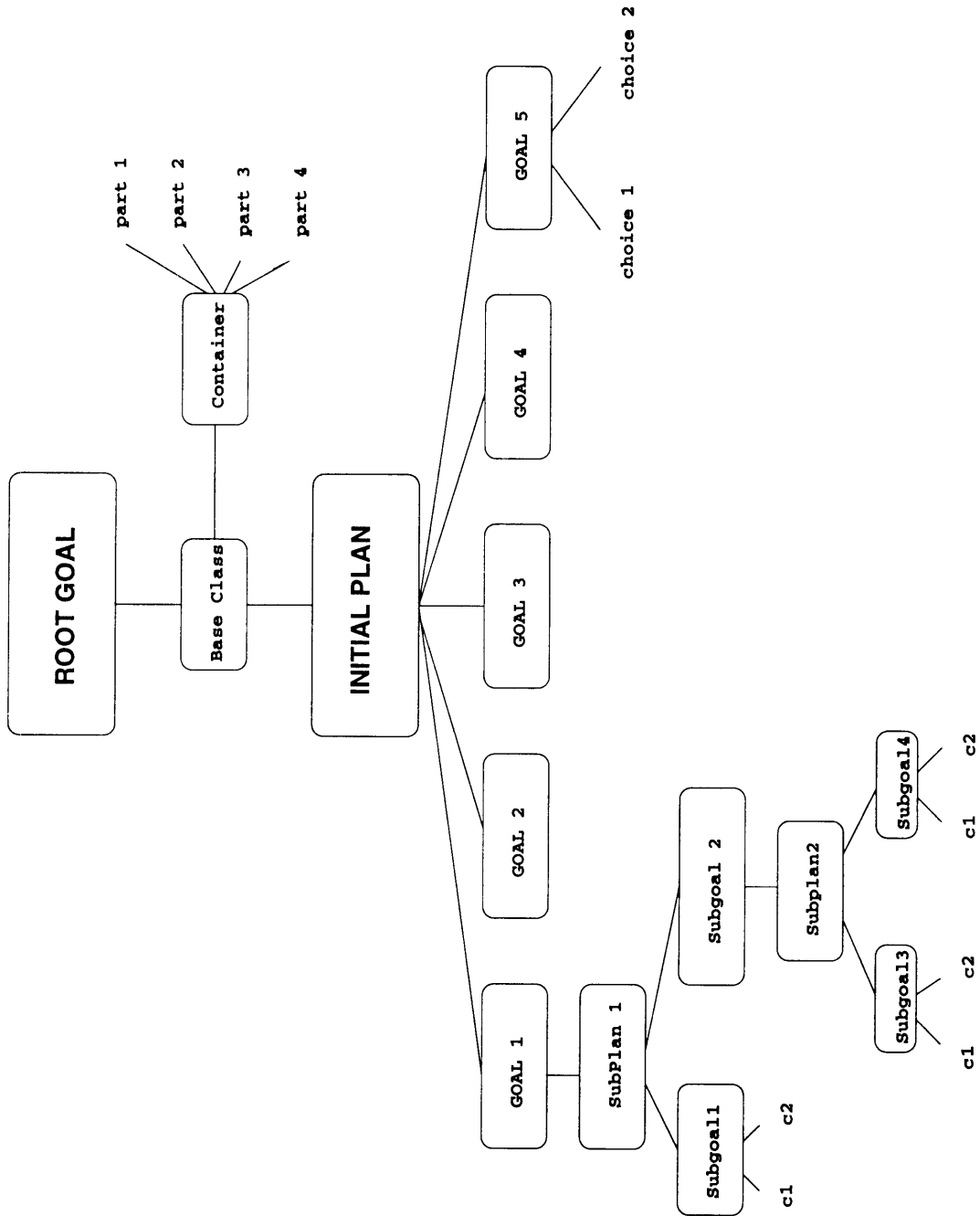


Figure 3-4: CONGEN Application Architecture.

---

## Chapter 4

# Tutorial I: Getting Familiar with CONGEN

This chapter presents CONGEN's basic user interface. The tutorial aims to show the users about various windows and capabilities of CONGEN. This tutorial is also tailored to make the new user familiar with CONGEN's user interface. Moreover, section 4.2.1 gives a detailed description on how the data and the information is managed within the framework of CONGEN and the object-oriented database system of CONGEN.

### 4.1 Starting a CONGEN Session

Before starting each CONGEN session, make sure that the environment is properly defined, such as paths, .Xresources, and the environment variables. You should refer to Appendix B for more detailed information about setting the environment to execute CONGEN. In addition, you have to ensure that an EXODUS database volume is assigned to you by your database administrator.

To start a CONGEN session:

1. Make a directory as the workspace directory for CONGEN (i.e. /congen/work)
2. Go to the directory, and issue the following command: **congen**

## 4.1 Starting a CONGEN Session

---

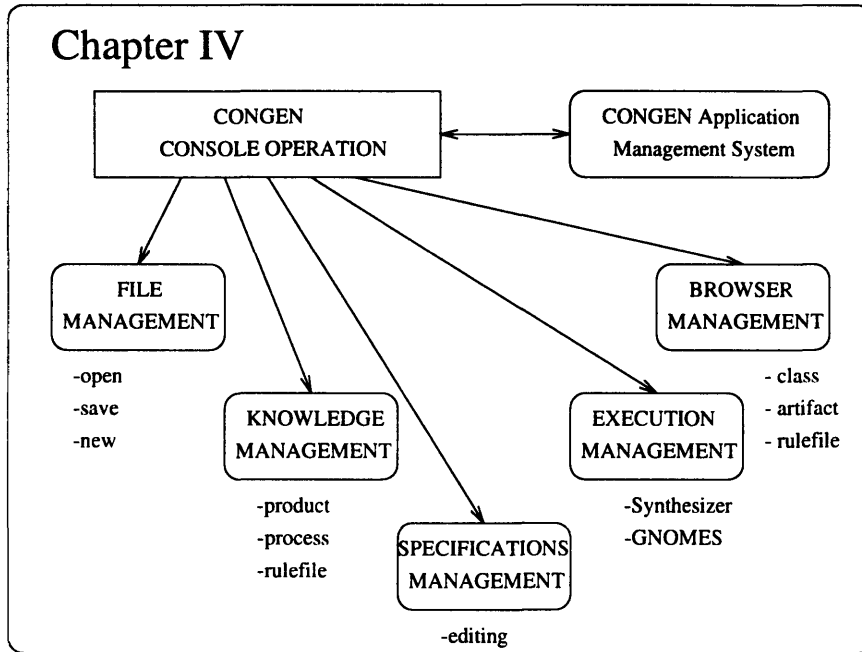


Figure 4-1: The roadmap of this chapter.

3. After a few moments, the MAIN CONSOLE window displays.

The MAIN CONSOLE window contains a window frame, a menu bar, a working directory display, and scroll bars. Several buttons (window menus at the upper left of window, minimize, and maximize) and a title bar are also in the window frame.

The MAIN CONSOLE window provides the ability for the users to manipulate applications, develop and edit knowledge, change specifications, execute the application, and browse the application parts at any point in the design process.

Figure 4-2 shows the CONGEN MAIN CONSOLE window that displays after typing: **congen**.

Now you are ready to start a CONGEN session. It is crucial that you go to the designated workspace directory before starting CONGEN, because CONGEN will automatically look for the files and data it needs in the directory where it was initially executed. You cannot edit the working directory display to change to another directory.

To end a CONGEN session, select **FILE**, and then **QUIT**. You will be asked whether

## 4.2 File Menu

---

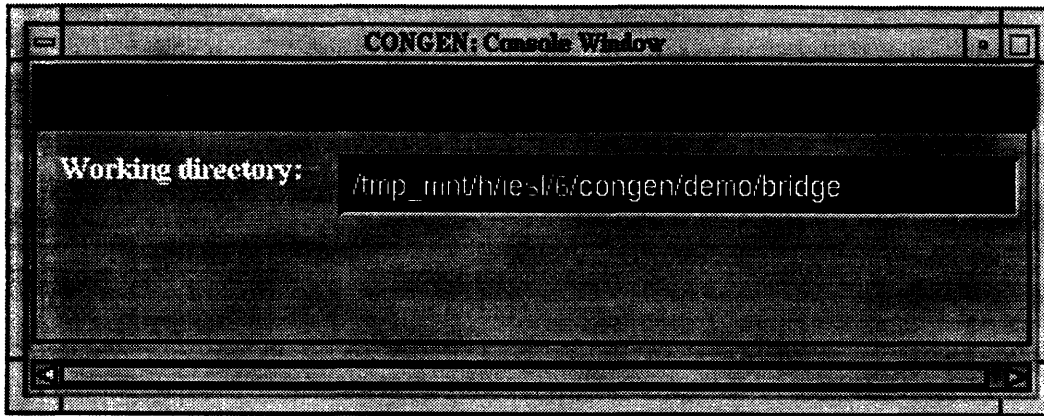


Figure 4-2: CONGEN Main Console.

you want to save the current application or not. Figure 4-3 shows the choices that you have at this point. Press **OK** if you want to disregard all the changes and exit CONGEN, or press **Cancel** to save the application first.

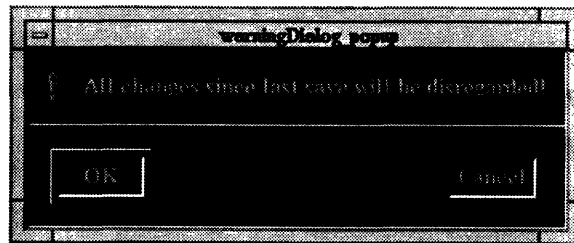


Figure 4-3: QUIT Warning window.

## 4.2 File Menu

The File menu consists of five submenus. They are:

### 1. NEW

Choose this submenu if you want to create a new application in the database. The name of the application must not have any space in between the words, use underline instead, for example: *An\_application* is a valid name, but *An application* is not. Figure 4-4 shows a sample of the window. Press **OK** if you want to create a new application with the designated name, or press **CANCEL** to cancel the process.



## 4.2 File Menu

---

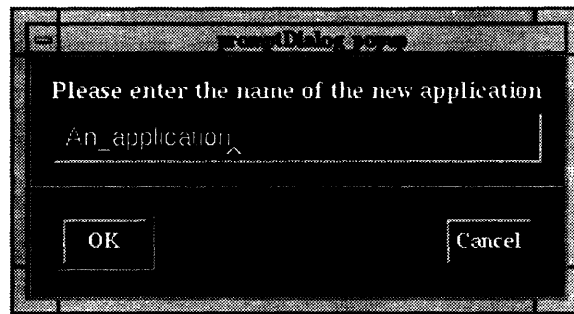


Figure 4-4: NEW Application window.

### 2. RETRIEVE APPLICATION

This submenu activates a window showing a list of available applications in the database. To select an application to be retrieved, press the left button on any application name, and press **OK**. Otherwise, press **CANCEL** to cancel the process. Refer to figure 4-5 for a view of **APPLICATION RETRIEVAL** window.

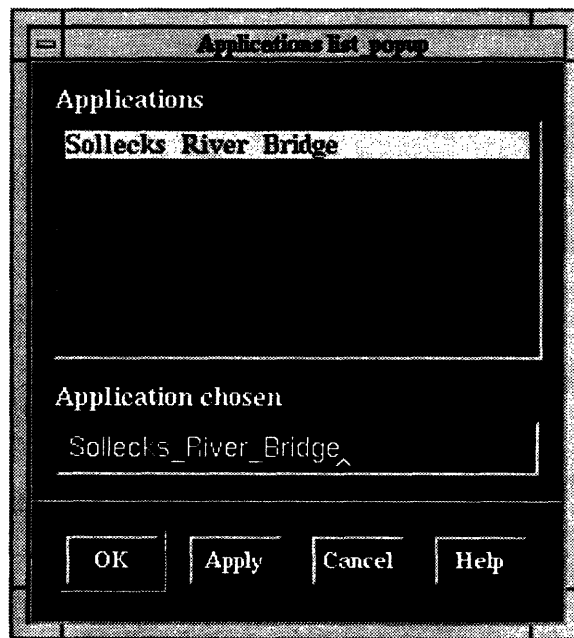


Figure 4-5: RETRIEVE Application window.

### 3. DELETE APPLICATION

## 4.2 File Menu

---

**DELETE** submenu will erase the designated application from the database. This submenu will activate a window showing a list of applications in the database. To select an application to be deleted, press the left button on any application name, and press **OK**. Otherwise, press **CANCEL** to cancel the process. Refer to figure 4-6 for a view of **APPLICATION DELETION** window.

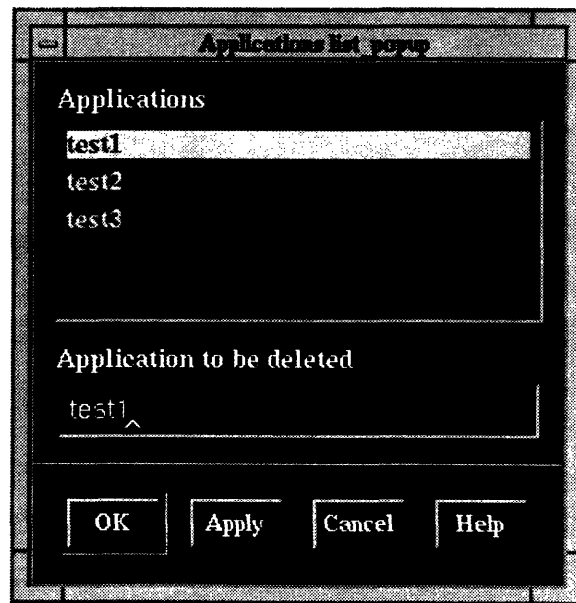


Figure 4-6: **DELETE** Application window.

### 4. **SAVE**

**SAVE** submenu will make all the changes to the active application permanent in the database. This operation is the same as committing transactions in database operations. Refer to figure 4.2.1 to understand how the applications and the database interact with each other.

### 5. **QUIT**

**QUIT** submenu quits the application. Before you quit the application, remember to decide whether you want to save the changes to the application by choosing **SAVE** or to abort all the changes to the application.

## 4.2 File Menu

---

### 4.2.1 CONGEN Application Management System

CONGEN has a specific scheme of application and database organization. All applications in CONGEN are treated as objects in the database. Applications are entities within CONGEN encapsulating a specific set of design processes. The data inside applications consist of classes, goals, plans, specifications, and contexts. In addition, applications provide a database entry point to the information contained inside, such as classes, goals, etc..

CONGEN application classes are publicly available across all applications residing within the same database. Different applications can share classes defined in one database. On the other hand, other information such as goals, etc. is privately held by the owner application. The class sharing mechanism enables a group of people to have common classes representing same kind of objects used by different applications. However, in the current implementation of CONGEN, there is no versioning capability available for the database. The versioning capability is needed to control the changes to the common classes.

The most important thing about the application management system is the notion of **SAVE** in the **FILE** menu in the MAIN CONSOLE Window. Saving an application means that the database is being updated permanently, or in other words, committing the transactions performed up till that time. For example, when the user executes the Synthesizer and the goal/plan creates some instances of an artifact, the instances are created within the database. If the user wants to make the instances permanent, then the user must hit **SAVE** in the **FILE** menu. Otherwise, the created instances are not permanently stored.

On the other hand, if the user needs to undo some of the instance creation, then the user must not **SAVE** the application, not committing the changes to the database. The user must know when and where to commit the changes to the database. Further advice follows below.

Users are advised to **SAVE** the application:

- everytime a new class has been entered, provided the Synthesizer has not been executed;
- everytime a new rulefile has been entered, provided the Synthesizer has not been

## 4.2 File Menu

---

executed;

- everytime a new goal has been entered, provided the Synthesizer has not been executed;
- everytime a new plan has been entered, provided the Synthesizer has not been executed; and
- everytime the user wants to save the created artifact instances after the Synthesizer has finished executing the application.

Users are advised not to save the application whenever:

- the entered class / goal / plan needs to be changed;
- the structure of the application (goals/ plans) needs to be changed after running Synthesizer while creating new artifacts; and
- the Synthesizer results are not satisfying and you have to rerun the Synthesizer again.

There is a major difference between committing transaction via **FILE** → **SAVE** in the MAIN CONSOLE window and in other windows such as DDH EDITOR, GOAL EDITOR, or other EDITOR windows. Whereas committing a transaction makes the update permanent in the database, **SAVE** in other windows means that the information entered in the window fields is transmitted to the corresponding data structure of CONGEN. In other words, there are two phases of saving an object before it is used in the application:

1. *Saving typed entries to the data structure.* This is done via **SAVE** submenu in the corresponding windows. This action will be referred to as **saving the objects** throughout the documentation.
2. *Saving data structure permanently to the database.* This is done via **FILE** → **SAVE** submenu in the MAIN CONSOLE window. This action will be referred as **saving the application** throughout the documentation.

### 4.3 Knowledge Menu

---

**A Reminder:** Whatever the user creates in the database depends on the commit action. If there is an unwanted modification to the database, for example, a mistake in Synthesizer execution, then the user must quit CONGEN to restore to the old state of the database.

### 4.3 Knowledge Menu

The Knowledge menu provides the facilities to encode the knowledge necessary to solve a CONGEN design application problem. The Knowledge menu has three selections

#### 1. Process Definitions

The Process Definitions menu shows the window of the Design Decomposition Hierarchy Editor (DDH). This editor shows all the important processes involved in the application, such as goals, plans, artifacts and contexts. You can click on any of the goals, plans, or created artifacts to access to the corresponding editor. For example if you click on the **Build\_access\_road** Goal entry in the DDH EDITOR window in figure 4-7, you will be presented with the GOAL EDITOR window as shown in figure 4-8.

In addition, to specify the root goal for the application, the **EDIT** submenu has an option for entering the name of the root goal. Moreover, this submenu also provides the ability to enter new plans and new goals to the DDH Editor.

After everything has been entered, you must press **FILE** and **SAVE** to save the entered items into the memory. Be advised that saving the DDH entries are not the same as committing a transaction. To make the update permanent, you must select the entry **FILE** → **SAVE** from the Main Console Menu.

#### 2. Product Definitions

The Product Definitions menu has the following functions: 1) to manipulate classes and rulesets, 2) browse the existing class and ruleset entries, and 3) compile everything into a COSMOS library used by CONGEN. As pictured in figure 4-9, the window

### 4.3 Knowledge Menu

---

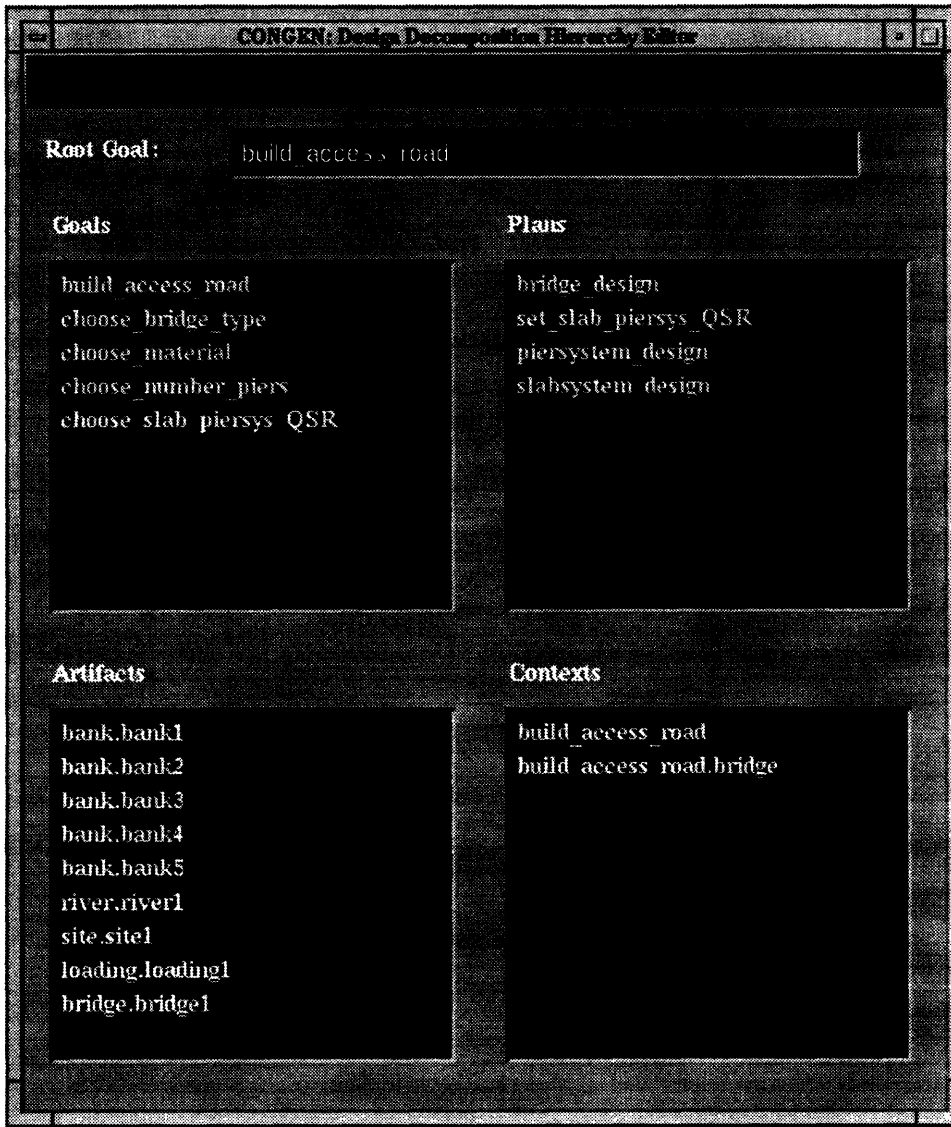


Figure 4-7: Design Decomposition Hierarchy EDITOR window.

enables the user to summon the CLASS EDITOR window by simply clicking on the entries.

For example, when the class **Slabsystem** is clicked, then there are three selections:

- Edit
- Browse

### 4.3 Knowledge Menu

---

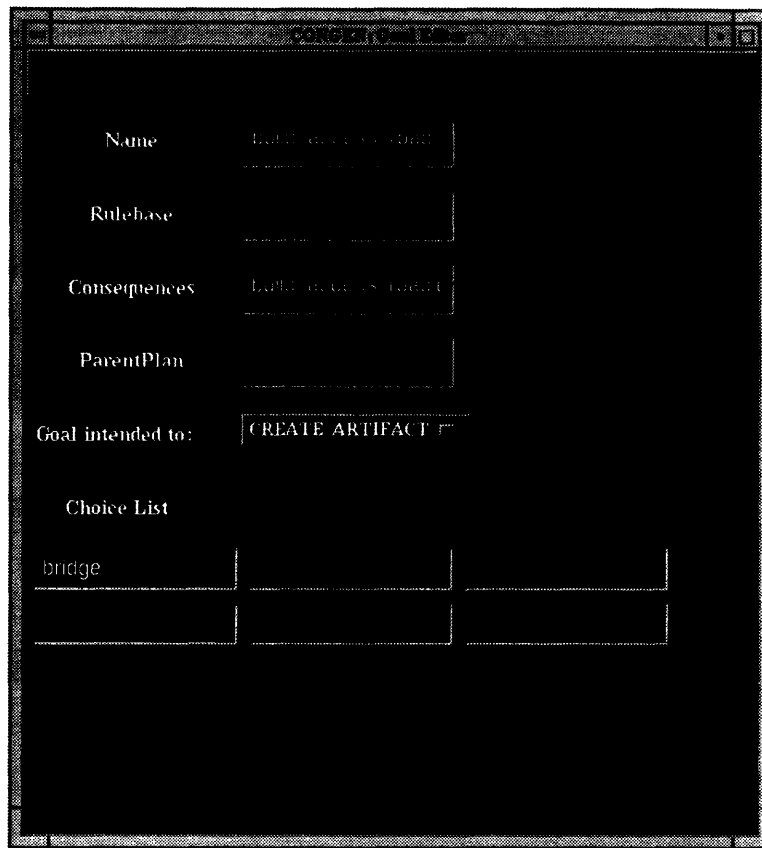


Figure 4-8: GOAL EDITOR window.

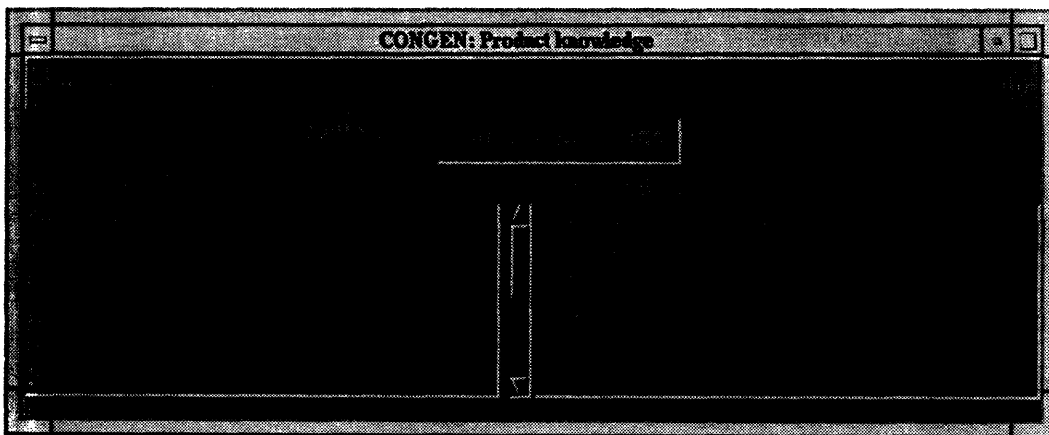


Figure 4-9: Product Knowledge window.

### 4.3 Knowledge Menu

---

- Delete

To choose one of the above selections, press the *right* button of the mouse on the item that you want to perform. After the *right* button is pressed on Edit, then the CLASS EDITOR (Figure 4-10) window is shown.

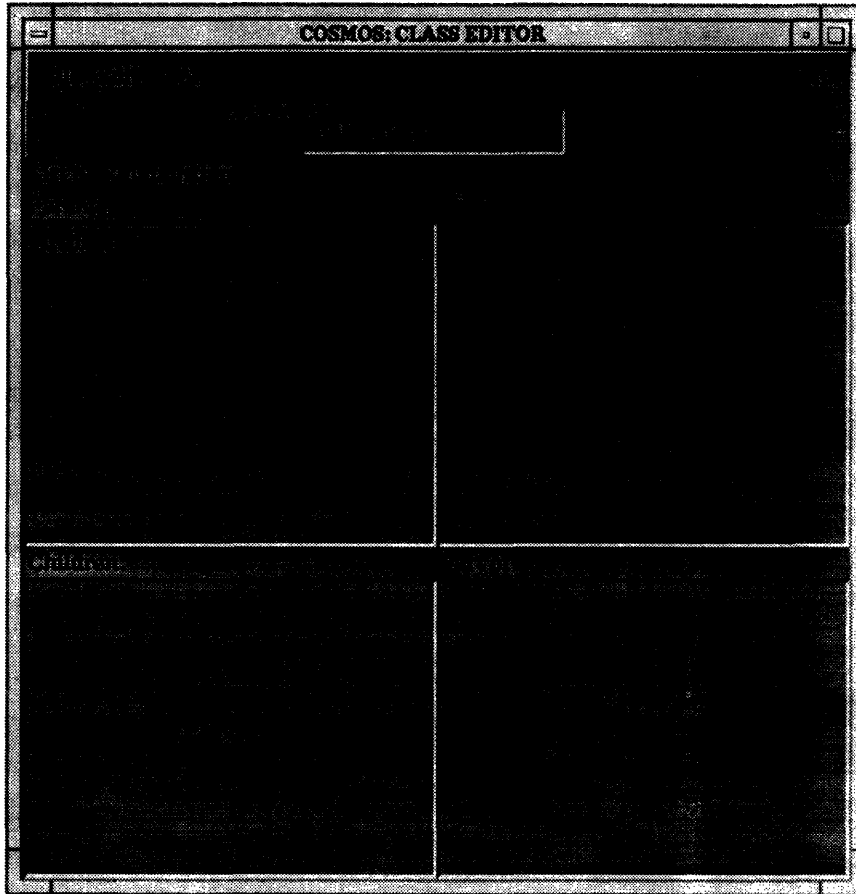


Figure 4-10: CLASS EDITOR window.

In addition to this, the **FILE** submenu gives the ability to create a new class, a new rulefile, or compiling the classes into COSMOS library. The **TOOL** submenu enables the user to browse the classes and rulesets.

### 3. Rule Definitions

The Rule Definitions menu displays the list of rules available for the application (Figure 4-11). To modify a rule, select the name of the rule with the mouse, and



#### 4.4 Specifications Menu

---

click on the button. You will be brought into the RULE FILE EDITOR window (Figure 4-12).

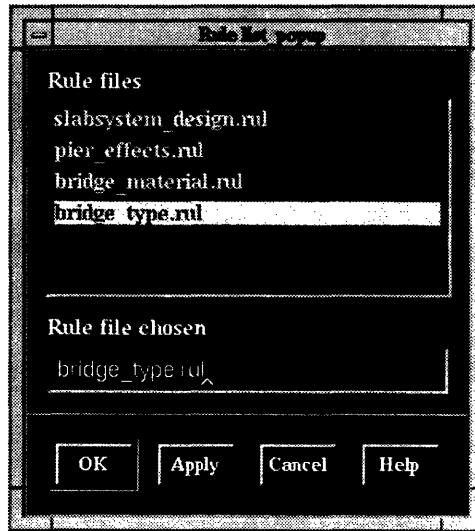


Figure 4-11: Rule List Selection window.

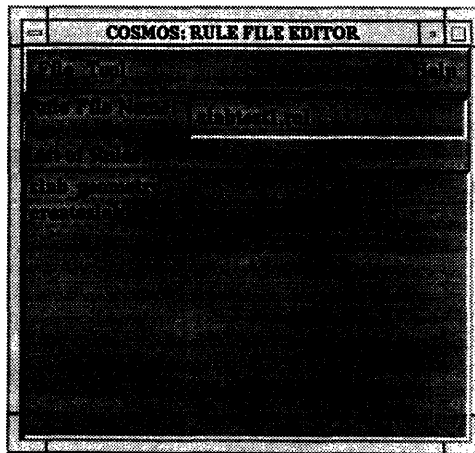


Figure 4-12: Rule File EDITOR window.

#### 4.4 Specifications Menu

The Specifications menu is created to help the users set the specifications for the application. The Specifications can be applied for a single instance of an artifact, or all instances of a specific class. When the **SPECIFICATIONS** submenu is clicked on the

## 4.5 Execute Menu

---

MAIN CONSOLE window, there are two selections available:

### 1. Input Context

When Input Context is selected by the user, the SPECIFICATIONS EDITOR window is shown. Users can create new specifications, add and delete specifications, or save the specifications to the memory. As a reminder, saving the SPECIFICATIONS EDITOR window does not mean saving it permanently into the database. To save it permanently, you must choose **FILE** → **SAVE** from the MAIN CONSOLE window.

Manual entries in the fields of the window are done using the mouse to navigate through the fields, or pressing the *TAB* key to move around the fields. After you have arrived at the desired field, then type the entries as wished. An example of a SPECIFICATIONS EDITOR window is presented in figure 4-13.

### 2. Geometry

Pressing the Geometry selection in this menu brings up the GNOMES Geometric Modeler application, as shown in figure 4-14. The purpose of this menu item is to provide the user flexibility in creating and manipulating new geometric forms.

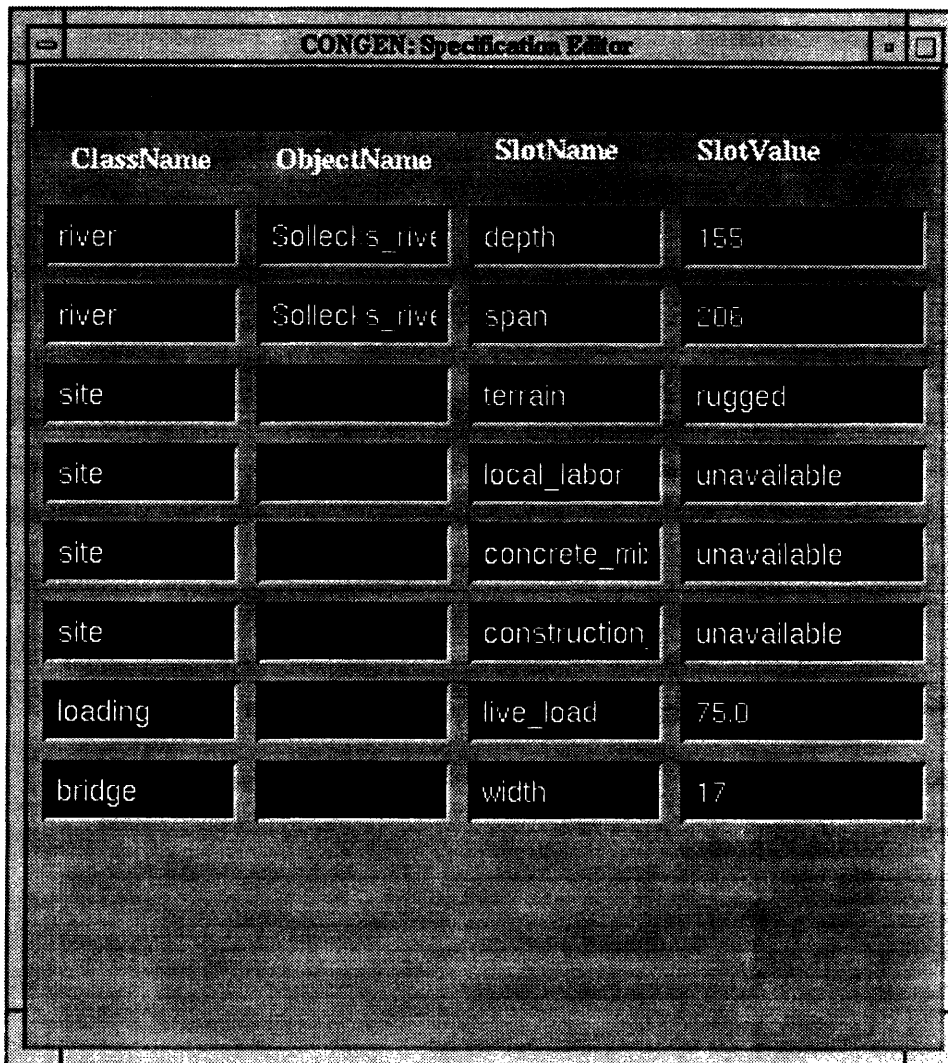
## 4.5 Execute Menu

The Execute menu consists of two submenus, the Synthesizer, and the Geometric Modeler. The Geometric Modeler submenu performs the same task as the Geometry in the Specifications menu. It is provided under the Execute menu to associate GNOMES with the Synthesizer whenever an application is executed in the Synthesizer. The SYNTHESIZER window is equipped with the scroll bars to help the user navigate through the branches of application decision tree.

For example, in figure 4-15, the goals and the plans have produced two slabs at the bottom, which are used as the top and bottom covers of a box. To look at the real geometry form of both covers, the user selects **Geometric Modeler** from the **Execute** menu, and executes GNOMES. Then, the user presses **Display Geometry** in the **Geometry** submenu in the SYNTHESIZER window. As soon as the selection is pressed, the

## 4.5 Execute Menu

---



The image shows a window titled "CONGEN: Specification Editor". It contains a table with four columns: "ClassName", "ObjectName", "SlotName", and "SlotValue". The table lists specifications for various classes and objects.

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

Figure 4-13: Specifications EDITOR window.

parts that have been created (top and bottom covers) are shown in the GNOMES window (Figure 4-16).

In addition to that, the SYNTHESIZER window provides flexibility for the user to:

- edit the specifications;
- add new artifacts and relationships;
- show the geometry of the artifacts; and

## 4.6 Browsers Menu

---

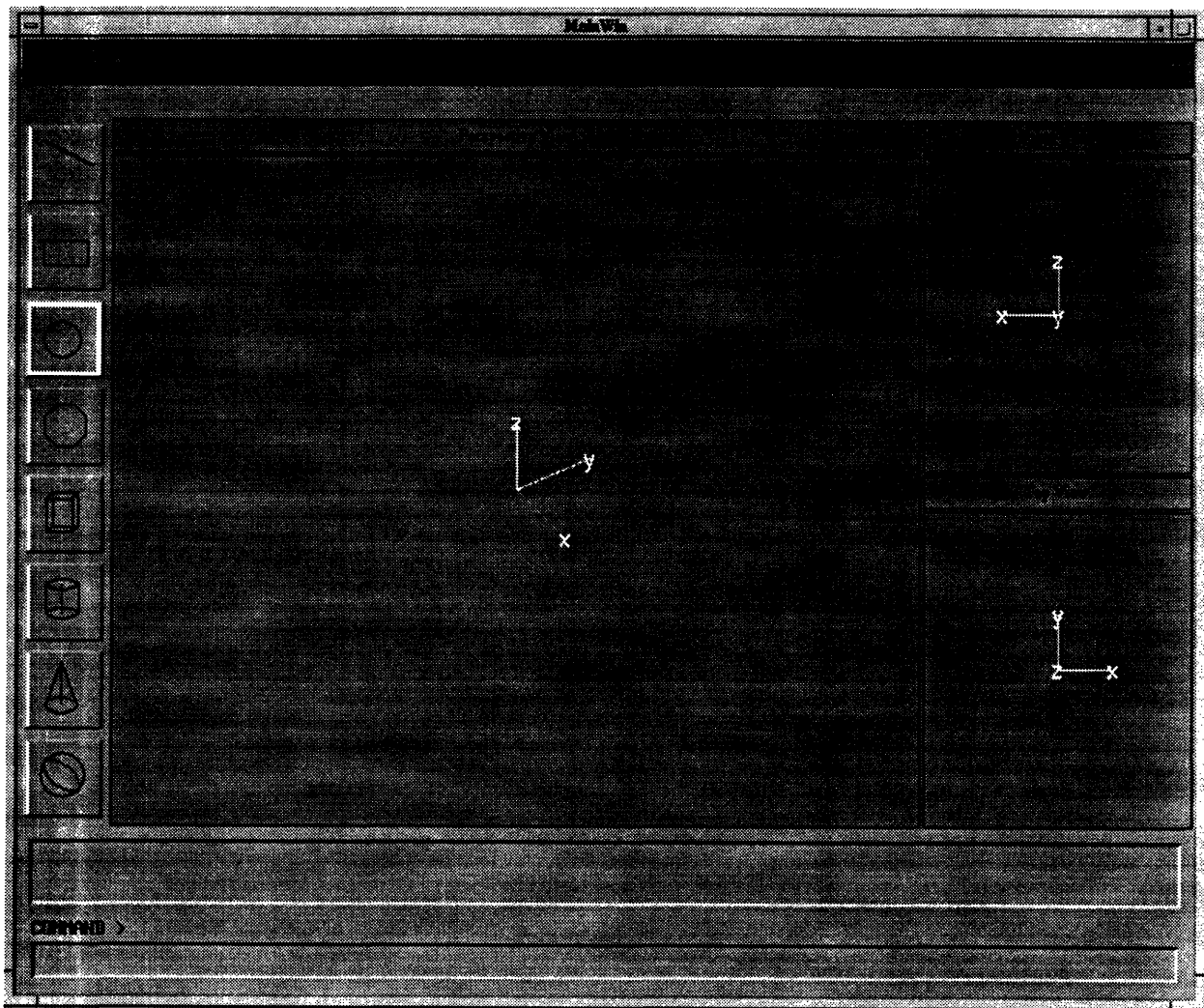


Figure 4-14: GNOME Geometric Modeler window.

- run the ATeams Constraint Manager.

## 4.6 Browsers Menu

This menu enables the user to browse different aspects of the application, such as artifacts, goals, plans, and rulefiles. In addition to the ability to browse, the user can also edit the corresponding objects from the browse windows. The Browsers Menu consists of three selections:

## 4.6 Browsers Menu

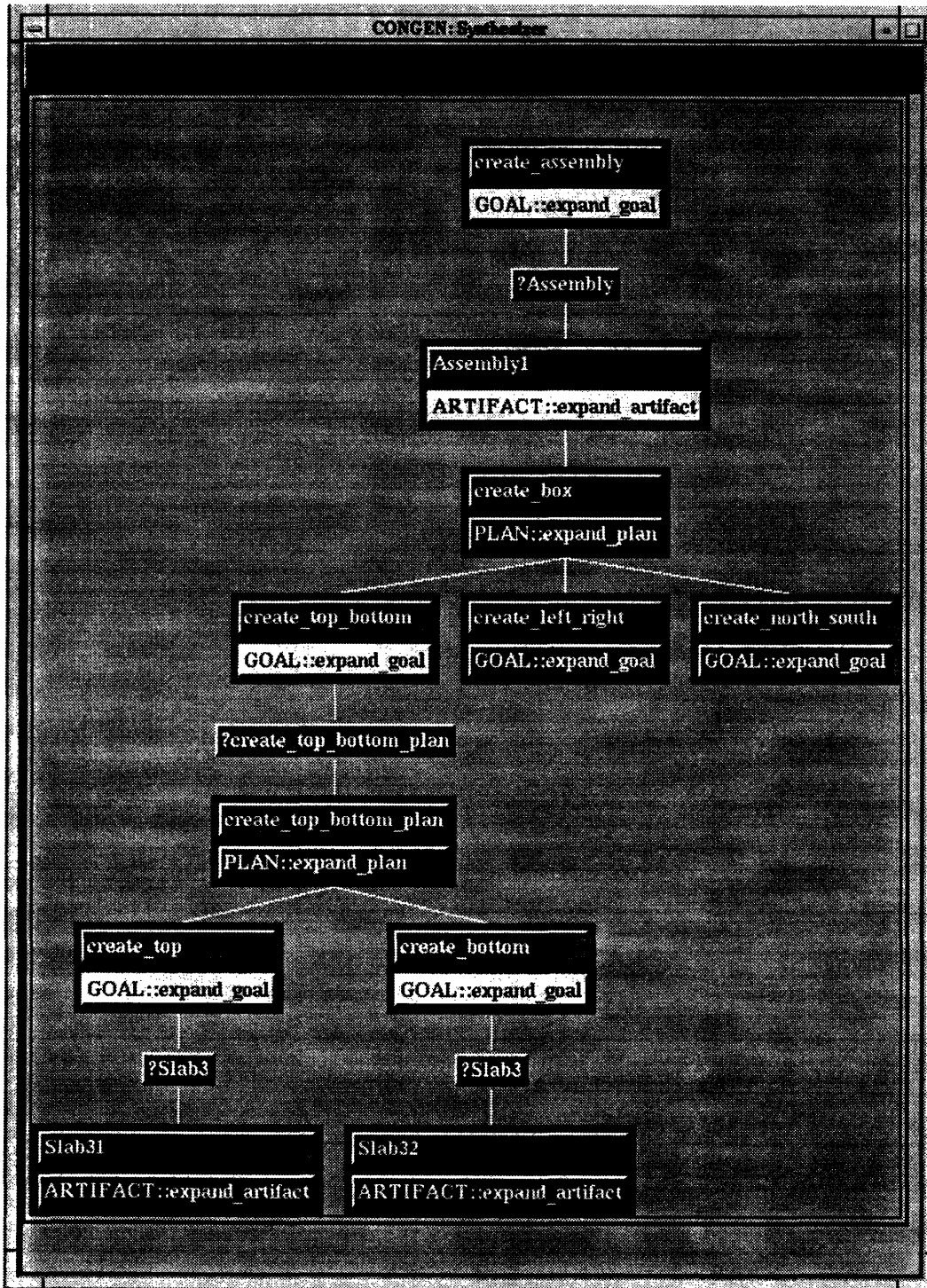


Figure 4-15: Synthesizer window.

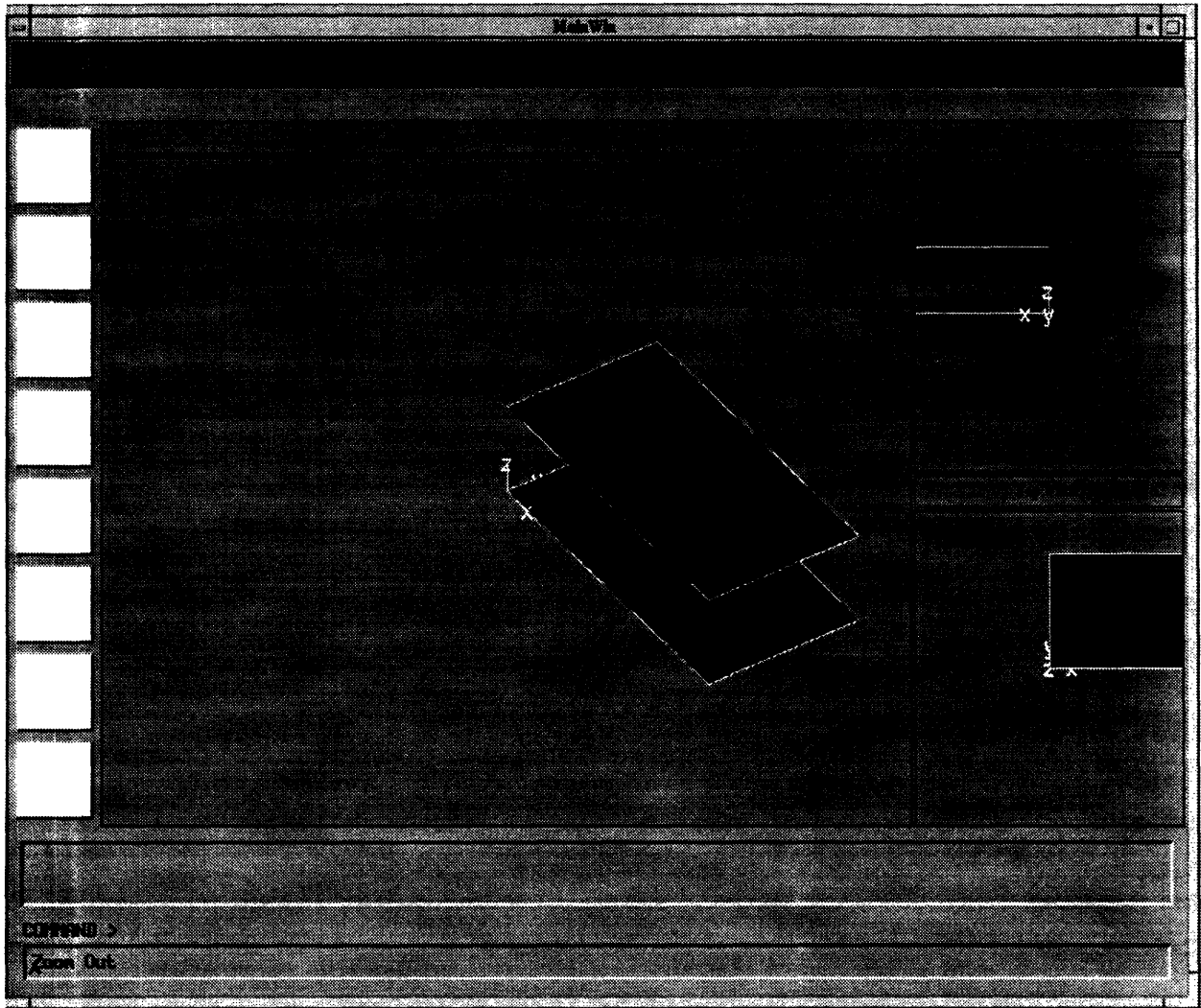


Figure 4-16: GNOME Geometric Modeler window with top and bottom covers of an application.

1. **Artifact Browser.**

The Artifact Browser acts as a class browser, as pictured in the figure 4-17.

2. **Design Decomposition Hierarchy (DDH) window.**

Selecting this submenu summons the DDH EDITOR window. This window is the same as the PROCESS DEFINITIONS EDITOR window from the **Knowledge** menu in the MAIN CONSOLE Window.

### 4.6 Browsers Menu

---

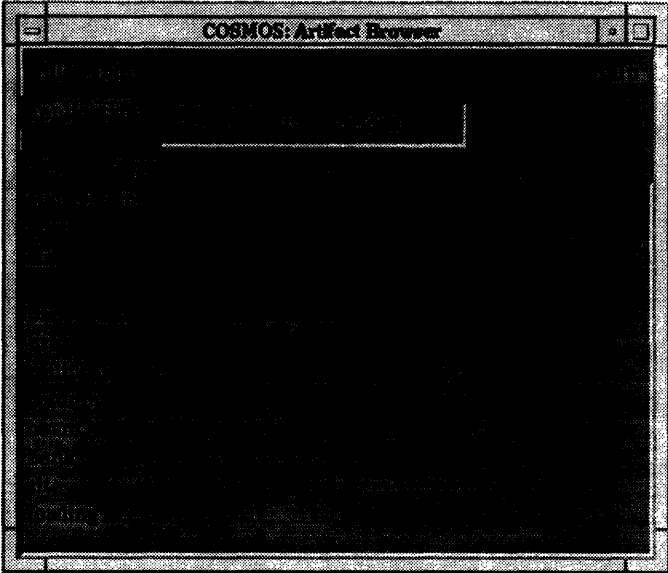


Figure 4-17: Artifact Browser window.

### 3. Rulefile Browser.

The Rulefile Browser allows the user to: look at the rulefiles, create a new rulefile, manipulate rules within a rulefile, and save the changes. A picture of the RULEFILE BROWSER window is shown in figure 4-18.

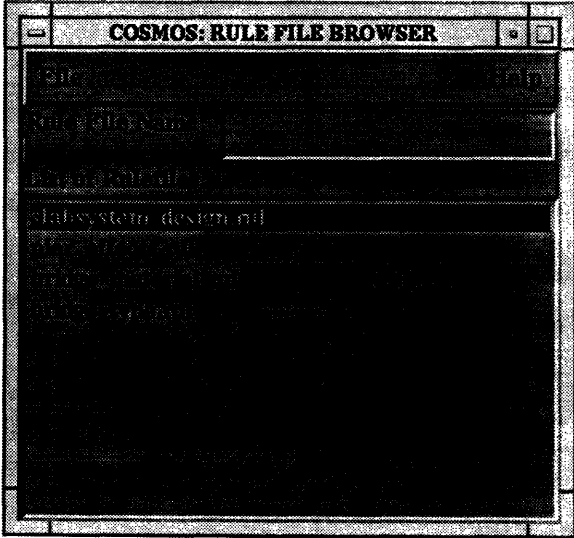


Figure 4-18: Rulefile Browser window.

## 4.7 Summary

---

### 4.7 Summary

This chapter acts as a tutorial to familiarize a new user to the various windows and capabilities of CONGEN. There are some important concepts that have been presented, such as the difference between saving an object and committing transactions to the database. This chapter gives an overview of the user interface of CONGEN, which prepares the user to solve a simple problem explained in the following chapter.



---

## Chapter 5

# Tutorial II: A Simple CONGEN Application

This chapter describes a simple yet representative application in CONGEN. It also shows how the modules and windows described in the previous chapter interact with each other. Firstly, the chapter provides an overview of the problem and the steps necessary to accomplish the final design. Then it guides the user in performing each step in CONGEN while making the user familiar with some of the unique mechanisms in CONGEN. Finally, the chapter closes with various approaches to solve the problem.

### 5.1 The Notation

Before we go on to the tutorial, the notations used throughout the following tutorials are as follows:

- the class names, attributes and instances are referred by quotes, for example “Slab” class with attributes “s.length” and “s.width”;
- the goals, plans, rulesets, and specifications are referred by italics, for example *create\_slab* goal and *design\_slab\_plan* plan;
- the applications are referred by boldface, for example: **Tutorial\_2** application;

## 5.2 The Problem

---

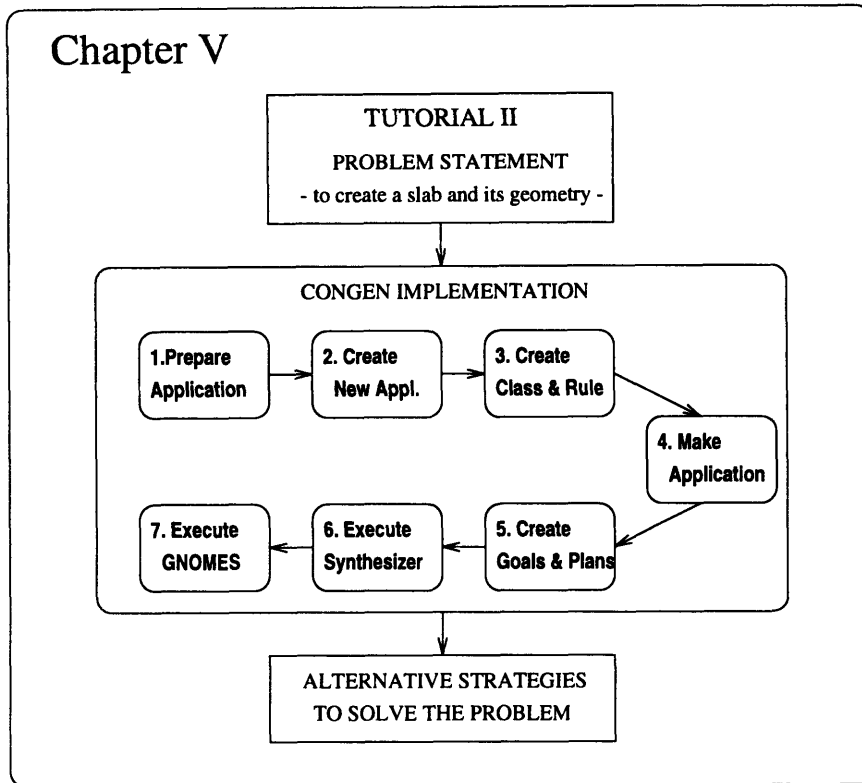


Figure 5-1: The roadmap of this chapter.

- the windows are referred by uppercase, for example: **CLASS EDITOR** window;
- the user entries are referred by a combination of boldface and italics, for example enter *Slab* as the name of the class; and
- the menu selections and the buttons are referred by uppercase and boldface, for example press **FILE** → **SAVE** - first you select the **FILE** submenu and then choose the **SAVE** submenu.

## 5.2 The Problem

The problem in this tutorial is to create a slab instance, create the slab's geometry, and show the geometry in GNOMES using a simple CONGEN design application.

Generally, the steps required to solve this problem in CONGEN are:

.

## 5.3 The Implementation

---

1. Prepare the application and the database.
2. Create a new application.
3. Create classes and rulesets.
4. Make the application.
5. Create goals and plans for the application.
6. Execute the Synthesizer.
7. Execute the Geometric Modeler (GNOMES) and show the geometry.

These seven steps are typical of a CONGEN application. These steps will be used many times throughout the following tutorial chapters. At this point, the details of the application have not been defined yet. The following section will provide further information on how to develop this application.

The simplest way to solve the problem poised above is to create *One Goal, One Class, One Rulefile*. We only need one goal to create the slab. The dimension and geometry of the slab is provided by the rulesets. As we are only creating one slab, we need only one class to contain all the artifact information. This solution is the one implemented by the tutorial.

## 5.3 The Implementation

The steps explained in the previous section apply generally. To be more specific, here are the preliminary steps to solve this problem:

### 5.3.1 Preparation

To build an application, you must do the following:

1. Have a specialized database volume prepared by your database administrator.

## 5.3 The Implementation

---

2. Set the `.cshrc` environment variable in the root directory to point to your database volume :

```
setenv EVOLID (the volume number of the database)
setenv CONDIR (the directory where your congen is installed)
setenv CONGEN_USER_AR (the directory where the class archive is installed)
setenv INCDIR (the directory where the congen include source files is placed)
```

3. Set the `.sm-config` EXODUS configuration file in the root directory to point to your database volume: `client*mount: (volume number) (port number)@(serverhost)`.
4. Create a special directory to run this tutorial such as: `mkdir /mit/gnomes/congen/tutorial2`.
5. Create a special directory to store the classes generated by this application: `mkdir /mit/gnomes/congen/ar`. Note that this directory must be named `ar` - archive. The `CONGEN_USER_AR` environment variable must point to this directory.
6. Change to the special directory that you have specified (`/mit/gnomes/congen/tutorial2`), and run `CONGEN` within that directory. This directory will store the rulefiles needed for this **SLAB** application.

All of these actions can be done with the help of your database administrator and system manager. For more information about the environment variables, please refer to Appendix C. After you have completed these tasks, you are ready to build the application.

### 5.3.2 Creating a new application

Before entering the classes, goals or plans, the first step is to create a new application in the database.

In the MAIN CONSOLE window, select **FILE** → **NEW**. You will be presented with the NEW APPLICATION window.

Enter: *Tutorial.2* and press **OK**

## 5.3 The Implementation

---

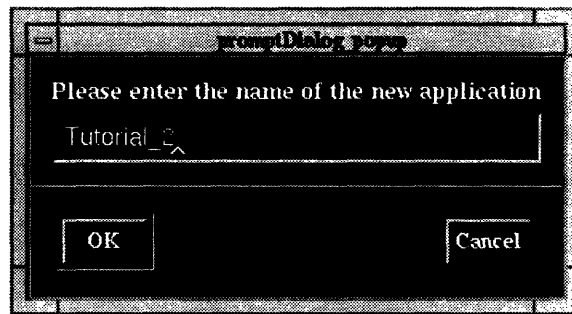


Figure 5-2: Creating Tutorial\_2 application.

Figure 5-2 shows the corresponding window. After you press **OK**, remember to commit the transaction via **FILE** → **SAVE** in the MAIN CONSOLE window. This will ensure the permanent storage of the new application.

- **NOTE:** If you have multiple applications in one database, you have to make the classes, rulefiles into the same directory as the others. This is because the Cosmos Makefile cannot look automatically for the corresponding parts of the applications if they are not within the same directory.

### 5.3.3 Creating classes and rulesets

- Creating classes.

For this application, one class is sufficient - “Slab” class. The “Slab” class consists of two attributes “s\_length” and “s\_width”. These attributes represents the slab length and width.

To create the “Slab” class,

1. Select **KNOWLEDGE** → **PRODUCT DEFINITIONS** to pop up the **PRODUCT KNOWLEDGE** window. Then select **FILE** → **NEW CLASS**. After the **NEW CLASS** window pops up, enter *Slab* for the new class name, and press **OK**.
  - **NOTE:** If you make any mistakes with the name or any window entry, place the cursor with the window in front of the word, and press **DELETE**. This

### 5.3 The Implementation

---

key would not work like the **BACKSPACE** key. You have to move the cursor to the front of unwanted word or letter and press the **DELETE** key.

2. The next thing is to edit the “Slab” class to enter the attributes needed. Highlight “Slab” class in the **PRODUCT KNOWLEDGE** window, and choose **EDIT** with the right button.

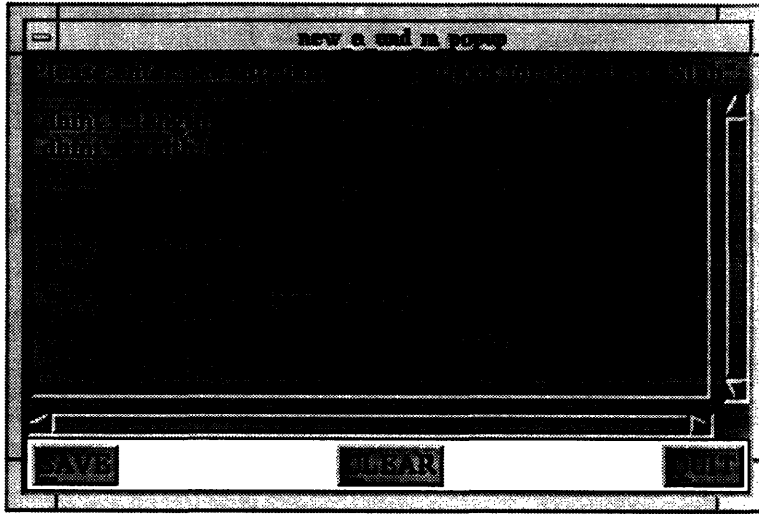


Figure 5-3: Slab Attributes entered.

The **COSMOS' CLASS EDITOR** window will pop up to provide the user with editing capability. Select **EDIT** → **PUBLIC** which will pop up the **COSMOS' CLASS EDITOR (PUBLIC)** window. Again, select **FILE** → **NEW** in this window. A new window is shown that will enable you to enter the attributes.

Enter:

```
dbint s_length;  
dbint s_width;
```

The result is shown in figure 5-3.

3. After everything is entered, press **SAVE**. The window will be cleared. After that, press **QUIT** and **OK** in the **WARNING** window to return to the **CLASS EDITOR (PUBLIC)** window (Figure 5-4). Press **FILE** → **QUIT** to go back

### 5.3 The Implementation

---

to CLASS EDITOR window. Press **FILE** → **QUIT** in the CLASS EDITOR window to go back to PRODUCT KNOWLEDGE window.

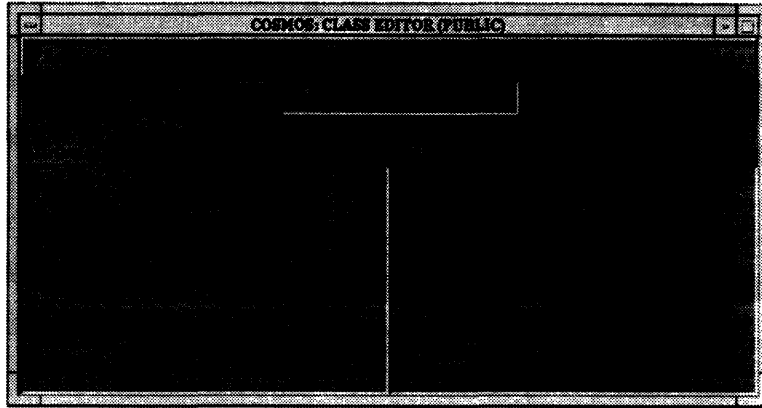


Figure 5-4: CLASS EDITOR (PUBLIC) window after attributes have been entered.

- **NOTE:** The “s\_length” attribute has a domain of dbint. Dbint is equivalent to the integer data type, in addition to that, it is tailored to be persistent in the EXODUS database. The same applies to the “s\_width” attribute. The dbint data type ensures that all the instances are kept permanently in the database.
- **NOTE:** There are some reserved keywords that are used exclusively by CONGEN classes. For example, “length” is the keyword attribute denoting a “Geometry” class’ “length” attribute. Therefore, the user cannot use the attribute name “length” for the Slab length, “s\_length” is more preferable. To avoid confusion, please refer to the list of reserved keywords in Appendix A.

- **Creating Rulefiles**

There are two rules associated with this application, *createslab* rule to create a slab instance with its dimensions, and *create-geometry* rule to set up the geometry values of the slab. To create the rulefiles, the following steps must be taken:

1. In the PRODUCT KNOWLEDGE window, select **FILE** → **NEW RULE**

### 5.3 The Implementation

---

**FILE** submenu. Enter: *Tutorial\_2* and press **OK**. You will be presented by the **RULE FILE EDITOR** window.

2. Select **FILE** → **NEW RULE**. When the **NEW RULE** window pops up, enter: *Create\_slab* and press **OK**. The **RULE EDITOR** window will come up as shown in figure 5-5.

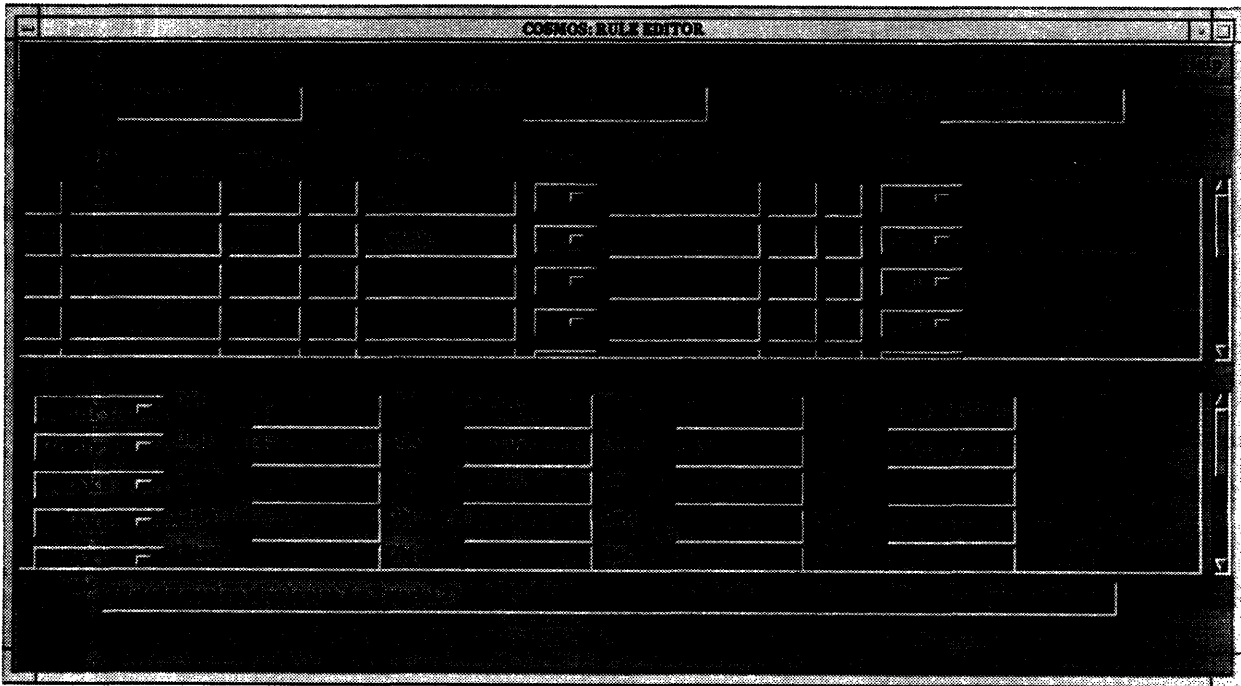


Figure 5-5: *createslab* rule in the **RULE EDITOR** window.

3. There are two ways to enter a ruleset, one is using the **RULE EDITOR** window, and the other one by using a text editor i.e.: **EMACS**. The first ruleset *createslab* is created using the **RULE EDITOR** window. As an exercise, the following is the picture (figure 5-5) of the *createslab* rule, you should enter the values as shown in the picture. Use **TAB** and mouse to guide you around. Remember that the **DEL** key has to be used as explained in the previous section. After entering the rule, you can either view the rule using the **FILE** → **CHECK RULE** submenu or save the rule into the data structure using **FILE** → **SAVE** as in figure 5-6.
4. Another way to enter the rule is through the text editor such as **EMACS**. You



### 5.3 The Implementation

---

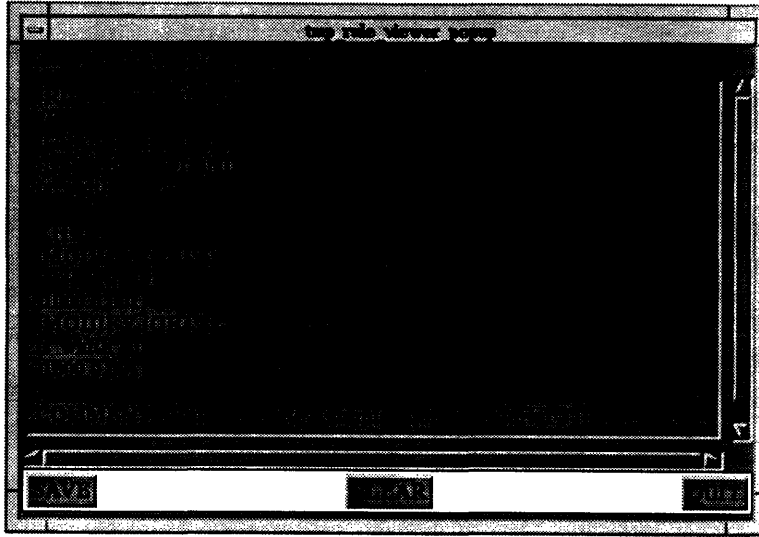


Figure 5-6: Saving *createslab* Rule in the RULE EDITOR window.

should try to type the second rule via a text editor. The detail of the second rule is as follows:

```
(RULE: create_geometry 10
IF
(CLASS: Slab OBJ: $x
(((s_length == $len) AND
(s_width == $wid) ) AND
(length != $len))
)
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(16,"slabgeom2"))
(MODIFY (OBJ:$x
(plane 2)
)1000 0.001)
(MODIFY (OBJ:$x
(length $len)
)1000 0.001)
(MODIFY (OBJ:$x
(width $wid)
)1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(0.0,36.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(90.0,0.0,0.0))
```

### 5.3 The Implementation

---

```
(EXECUTE VAR:$rtn OBJ: $x show_geometry()  
)  
COMMENT:"Rule to set up the slab geometry")
```

- **NOTE:** If the rule is not complicated, the user is invited to use the RULEFILE EDITOR window in CONGEN. But if the rule consists of many branches and actions, it is suggested that the user exploits the capability of a text editor. Entering a complex rule in a text editor allows easier editing if there are any errors.

5. The meaning of the two rules are as follows:

- (a) The first rule states that when there is a slab created, and the parameters are not set yet, then set the length to 40 and the width to 20.
- (b) The second rule states that after a slab is created, pass the contents of the parameters to the variables, and use the parameters to create a geometry instance of that slab to be shown in GNOMES. After a geometry has been created, fill the parameters of the geometry, set the translation and rotation parameters, and show the geometry in GNOMES screen if available.

- **NOTE:** The syntax of the rule sets are listed in the appendix B. Notice:

```
((s_length == $len) AND  
(s_width == $wid) ) AND  
(length != $len))
```

The usage of the third condition is to stop the rule from firing continuously, because the first two conditions will always be fulfilled whenever the first rule is fired. Therefore, it is crucial to stop the rule from firing by comparing the "Geometry" class attribute, before and after the create\_geometry rule has been fired. It ensures that after the "Geometry" class attribute "length" has been changed, the rule will not be fired again.

In addition, the conditions in a rule must be paired, for example ((( cond 1 ) AND ( cond 2 )) AND ( cond 3 )), the expert system cannot handle more than two conditions to be evaluated in one set of parenthesis. Refer

## 5.3 The Implementation

---

to Appendix B for more information about COSMOS rules.

### 5.3.4 Making the application

After all the classes and rulefiles have been saved and entered into CONGEN and the application has been saved (transactions have been committed), then you are ready to compile all the classes (making the application). Press **FILE** → **MAKE APP** to compile the “Slab” class and make it ready for CONGEN Synthesizer execution. You can monitor the compilation of the classes via the XTERM window. After the compilation is finished, you should save the application again to ensure that everything has been saved permanently in the database.

### 5.3.5 Creating goals and plans for the application

When the parts that make the application have been created and compiled, it is time to enter the processes that governs the application. Firstly, select **KNOWLEDGE** → **PROCESS DEFINITIONS** submenu from the MAIN CONSOLE window. After it is selected, then the Design Decomposition Hierarchy (DDH) EDITOR window will show up. This is the window where the manipulation of goals, plans and artifacts are done. **Tutorial\_2** application only has one goal - to create a slab and set the geometry of the slab.

After the DDH EDITOR window comes up, select the **EDIT** → **GOAL** submenu that brings up the GOAL EDITOR window. Enter the same entries as figure 5-7.

To specify the Root Goal as *create\_slab*, select **EDIT** → **SPECIFY ROOT GOAL** and enter **create\_slab** as the root goal and press **OK**.

There are six important parts in the Goal Editor:

1. *Name.* This field denotes the name of the goal to be entered. The name cannot have any space in between the words. We choose the name *create\_slab* as this Goal's name. You cannot have duplicate goal names within the same application.
2. *Rulebase.* This field denotes the name of the rulefile used to make a choice for the intention of the goal. For example, if the goal has to create an artifact from

### 5.3 The Implementation

---



Figure 5-7: create\_slab Goal shown.

three different classes, then the Rulebase will contain the rulesets needed to make the choices based on the information given by previous decisions. In other words, Rulebase contains rulesets that are fired according to the backward chaining concept. In this case, we do not have any ruleset to choose because we only have one choice, the “Slab” class.

3. *Consequences.* This field contains the name of the rulefile used to do anything after the intention of the goal has been executed. In this case, the *Tutorial\_2.rul* ruleset is chosen because the ruleset performs all the tasks necessary to complete the application. It modifies the Slab attributes after the Slab instance is created, and creates the corresponding geometry for the Slab instance. Whereas the Rulebase uses the backward chaining, the Consequences rulesets use the forward chaining concept.
4. *ParentPlan.* This field is optional. A root goal doesn't have to have any value in

### 5.3 The Implementation

---

this field. On the other hand, other goals must have a value for this field because it will ensure the synchronicity of the Synthesizer execution process and the logic of the application. In this case, because the goal is also the root goal, then *create\_slab* goal doesn't have any ParentPlan.

5. *Goal intended to.* This field has four different choices, as explained in chapter 3:

- **CREATE\_ARTIFACT**
- **MODIFY\_ARTIFACT**
- **EXPAND\_PROCESS**
- **EXTERNAL\_FUNCTION**

The *create\_slab* goal's intention is to create a Slab artifact.

6. *Choice List.* This field gives alternative choices to the intention of the Goal; for example, if the goal is intended to choose one out of three types of slabs - waffle slab, two-way slab, and simple slab, then the class names must be listed in the fields of the *Choice List*. In this case, there is only one choice for the Goal, therefore there is only one entry in the list.

After finishing entering the values, you should save the values by selecting **FILE** → **SAVE** in the GOAL EDITOR window, and **FILE** → **SAVE DDH** in the DDH EDITOR window. Save the whole changes in the application by choosing **FILE** → **SAVE** in the MAIN CONSOLE window.

The **Tutorial-2** application only has one goal, and no plans. This is done because there is no need to expand the process. As we know, the plan acts as an organizer of goals. Therefore if there is only one goal, then there is no need for any plan. The connection between goals and plans will be discussed in deeper detail in the following chapters.

#### 5.3.6 Executing the Synthesizer

When the product and process knowledge has been entered and saved, the problem solving mechanism should be checked by running the Synthesizer. To run the Synthesizer,

### 5.3 The Implementation

---

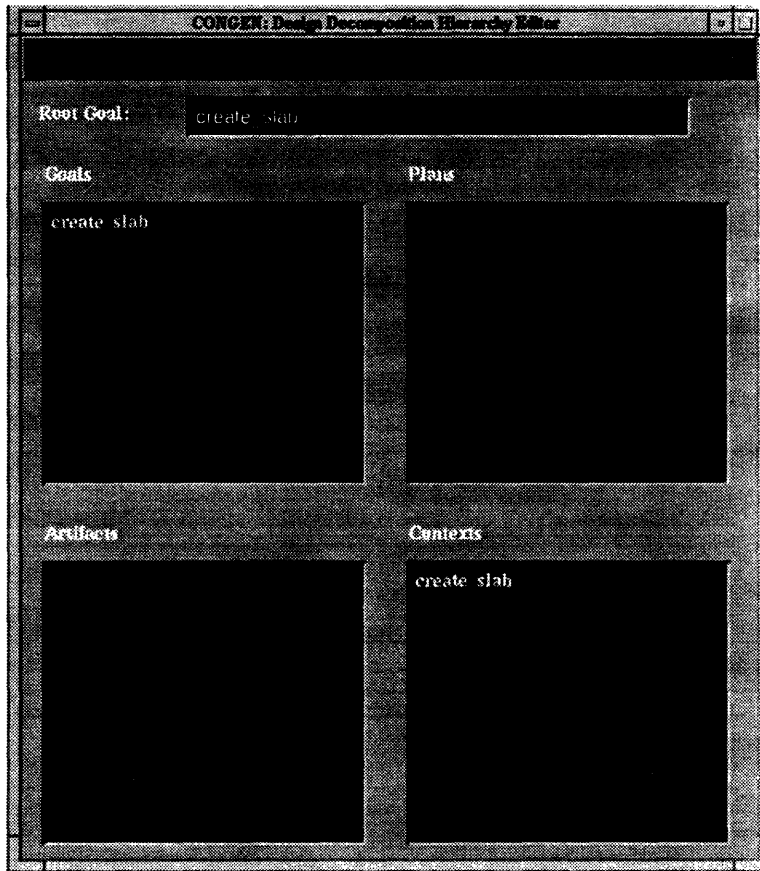


Figure 5-8: DDH Editor after everything has been entered and saved.



Figure 5-9: SYNTHESIZER window after finished creating the Slab.

### 5.3 The Implementation

---

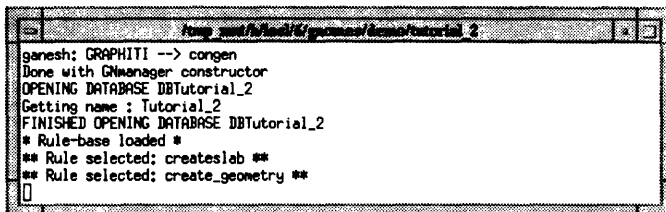


Figure 5-10: XTERM window after finished creating the Slab.

select **EXECUTE** → **SYNTHESIZER** in the MAIN CONSOLE window. After it is selected, the SYNTHESIZER window will pop up.

To run the application, press the button **GOAL ::EXPAND\_GOAL**. Pressing that button will create another button labeled **?SLAB**. This button means that the user is expected to continue the process of creating a Slab, as contained in the *create\_slab* goal.

Pressing this button will create an instance of a “Slab” with the specifications enlisted in the *Tutorial\_2.rul* rulefile. The outcome of this process is shown in figure 5-9. To check whether the rules have been fired correctly, refer to the XTERM window, as shown in figure 5-10.

- **NOTE:** Expanding the artifact once more after it has been created will produce the error message:

```
Error opening Slab.rul
```

The reason behind this message is that there is nothing more to be done after creating the instance of the artifact. However, a class of artifacts can have a ruleset attached to it to enable the user shape the basic behavior of the artifact itself. The ruleset attached to the class may check the constraints of the class or even continue the design process of a slab by attaching beams to it. The name of the attached ruleset must be the same as the class added by extension .rul such as *Slab.rul*. In this application, the ruleset *Slab.rul* is fired automatically whenever the “Slab” instance is expanded. Since *Slab.rul* does not exist, the error message will be produced.

- **NOTE:** When running the Synthesizer, remember that whenever the Synthesizer creates an instance of an artifact, the instance is still in the transitional memory before

## 5.3 The Implementation

---

being committed permanently to the database. If you want to change a goal or a plan after running the Synthesizer, you should remember to **QUIT** the application. If you do not **QUIT** then Whatever changes made after the Synthesizer finishes running will not be reflected in the next run of Synthesizer. In addition to that, you have to make sure that after running the Synthesizer, **DO NOT** save the application because all the instances will be made persistent. Unless you want to keep the data and instances permanent in the database, you should **QUIT** the application.

### 5.3.7 Executing the Geometric Modeler (GNOMES) and show the geometry

The last step of the application is to execute the Geometric Modeler to see the geometry of the slab. So far, we have defined the geometry of the slab in the **create\_geometry** rule as:

```
(EXECUTE VAR:$rtn OBJ: $x create_geometry(16,"slabgeom2"))
(MODIFY (OBJ:$x
(plane 2)
)1000 0.001)
(MODIFY (OBJ:$x
(length $len)
)1000 0.001)
(MODIFY (OBJ:$x
(width $wid)
)1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(0.0,36.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(90.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x show_geometry())
)
COMMENT:"Rule to set up the slab geometry")
```

This information means that:

1. Create an instance of “Geometry” of the artifact with the name “slabgeom2”. Any name for the instance is possible.
2. Set the attribute values of the geometry in the XY plane (2). In addition, also set the “length” and “width” attribute of the “Geometry” the same as the “Slab”’s.



### 5.3 The Implementation

---

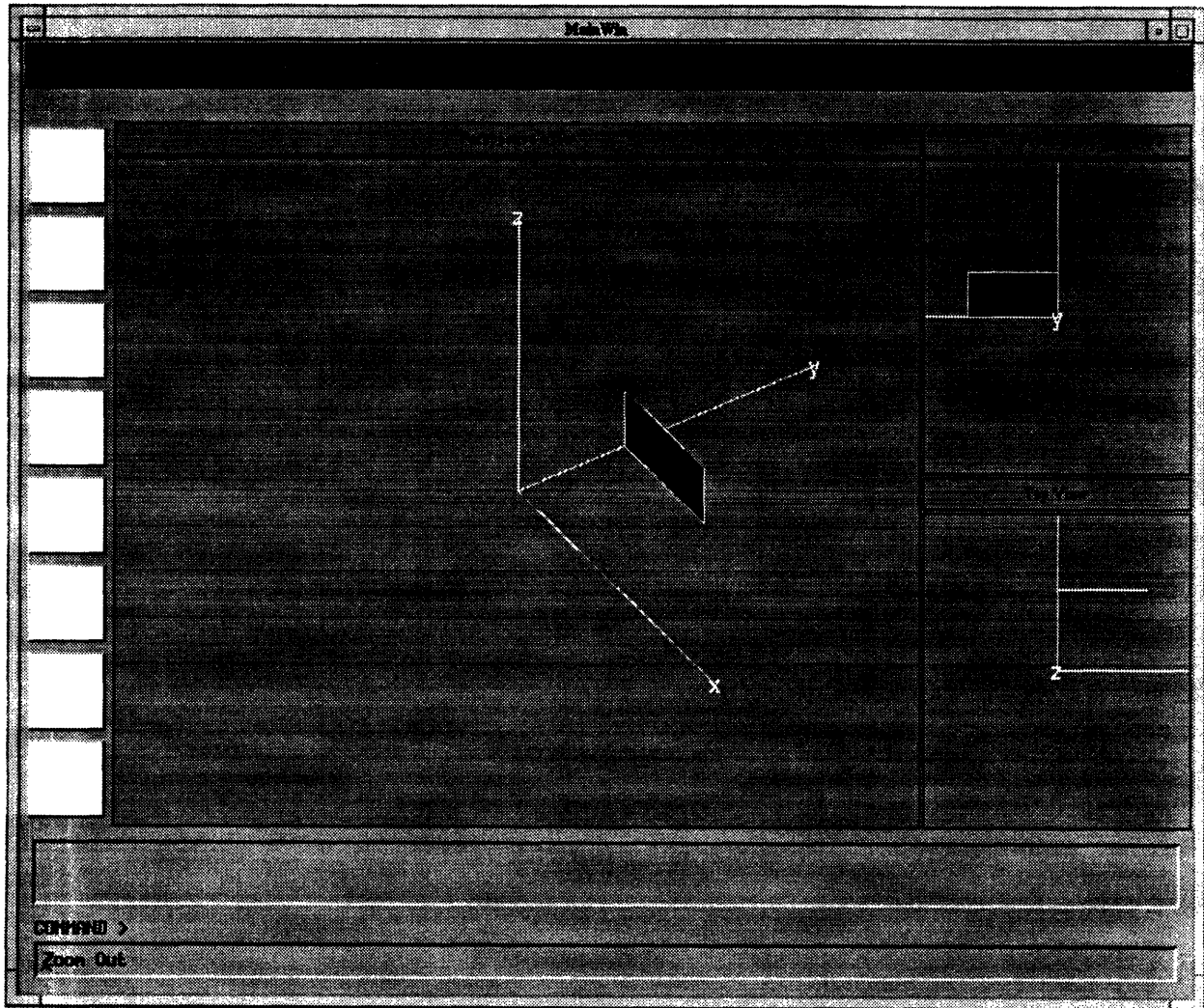


Figure 5-11: GNOMES showing the Slab geometry.

3. Translate the created geometry 0.0 in the X axis, 36.0 in the Y axis, and 0.0 in the Z axis.
4. Rotate the created geometry 90.0 degrees w.r.t X axis, 0.0 degrees w.r.t Y axis, and 0.0 degrees w.r.t Z axis.

For more information about the Geometry operations, refer to Chapter 7 Section 1.4. To execute GNOMES Geometric Modeler, select **EXECUTE** → **Geometric Modeler** in the MAIN CONSOLE window. After the GNOMES window is shown, go back to the

## 5.4 Other Solutions to the Problem

---

SYNTHESIZER window and select **GEOMETRY** → **SHOW GEOMETRY**. This step assumes that you have run the Synthesizer correctly with the *createslab* and *create\_geometry* rules without error. After selecting the **SHOW GEOMETRY**, go back to the GNOMES window, and the “Slab” geometry will be shown, e.g. as in figure 5-11.

- **NOTE:** You can always start GNOMES before the Synthesizer starts or executes everything. If the rule states:

```
(EXECUTE OBJ: $x show_geometry())
```

then the geometry is automatically shown whenever this rule is fired. Again, this depends on the user’s preference of Synthesizer and GNOMES execution.

## 5.4 Other Solutions to the Problem

In solving a design problem, there may well be many strategies applicable. This section will elaborate on how we can develop the **Slab** application differently to produce the same design. Below is a list of various ways to solve the problem :

1. *One Root Goal, One Plan, One Class.*

The above structure is considerably more complex than our primary example in this chapter. The steps for this alternative is as follows:

- **Root Goal:** *create\_slab*. This root goal will create a slab to be used in the application.
- **Plan:** *create\_slab\_plan*. This plan modifies the dimension and creates the geometry. There is no goal associated with this plan, only a rulefile to change the “Slab” dimension and geometry.

This example can be seen in figure 5-12 and the listings are provided in the Appendix D.

## 5.4 Other Solutions to the Problem

---

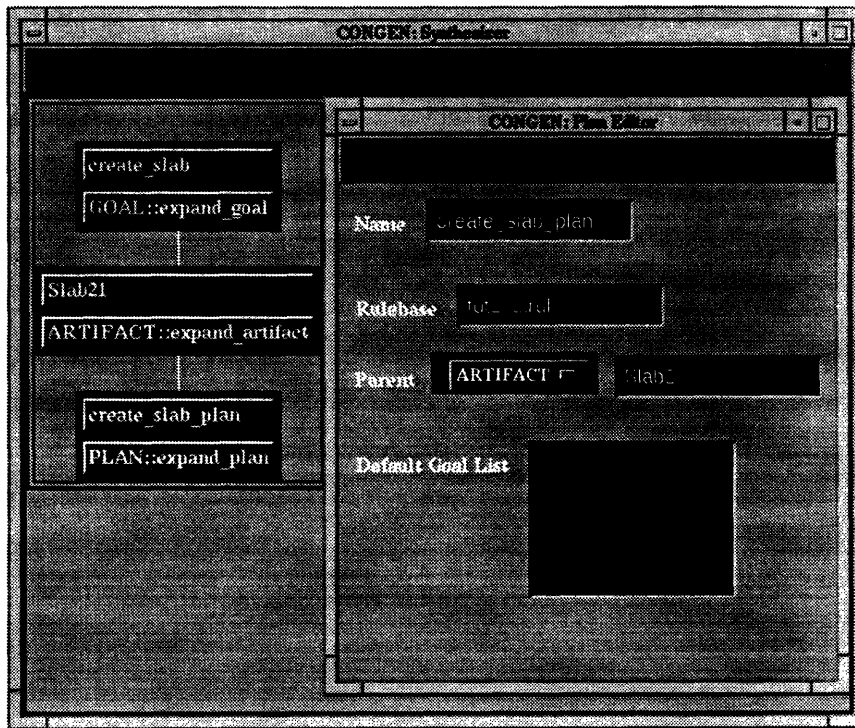


Figure 5-12: Other alternative solution to Tutorial.2.

### 2. One Root Goal, One Plan, Two Subgoals, Two Rulefiles.

Other complicated alternative is to expand everything. The steps towards solving this one is:

- Root Goal: *create\_slab*. This root goal will create a slab to be used in the application.
- Plan: *create\_slab\_plan*. This plan modifies the dimension and creates the geometry. The goals in this plan are:
- Goal: *modify\_slab*. This goal modifies the dimension of the slab and has a subplan Plan: *modify\_slab\_plan* to fire the rule to change the dimension.
- Goal: *geometry\_slab*. This goal modifies the geometry of the slab and has a subplan Plan: *geometry\_slab\_plan* to fire the rule to change the geometry of the artifact.

## 5.5 Summary

---

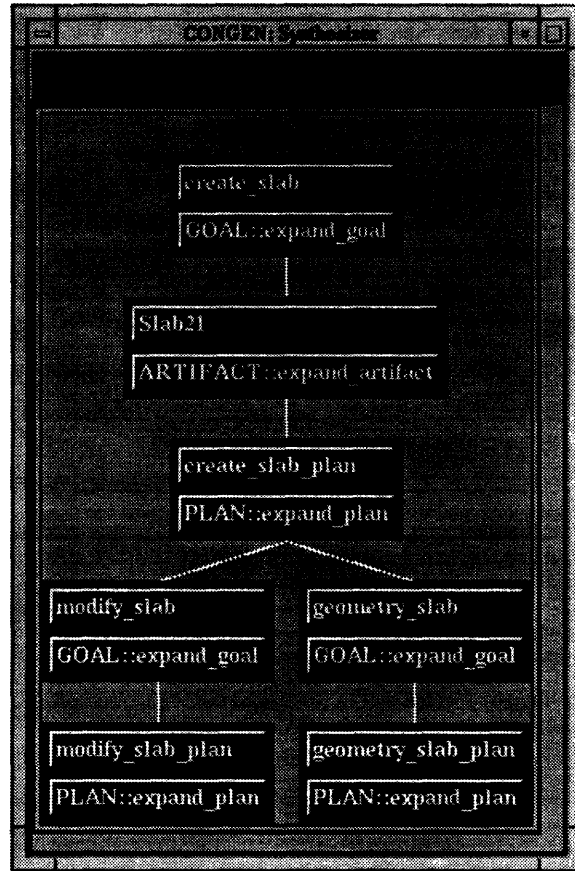


Figure 5-13: Other alternative solution to Tutorial\_2.

The output for this example is shown in figure 5-13 and the listings are provided in Appendix D. This solution employs a special technique to run a ruleset by only expanding a process. This technique was explained in Chapter 3, Section 5. The application of this is shown in figure 5-14.

## 5.5 Summary

So far, we have succeeded in finishing Tutorial 2 which implemented basic concepts of CONGEN, such as:

1. *Creating Tutorial\_2 application*
2. *Creating Slab class*

## 5.5 Summary

---

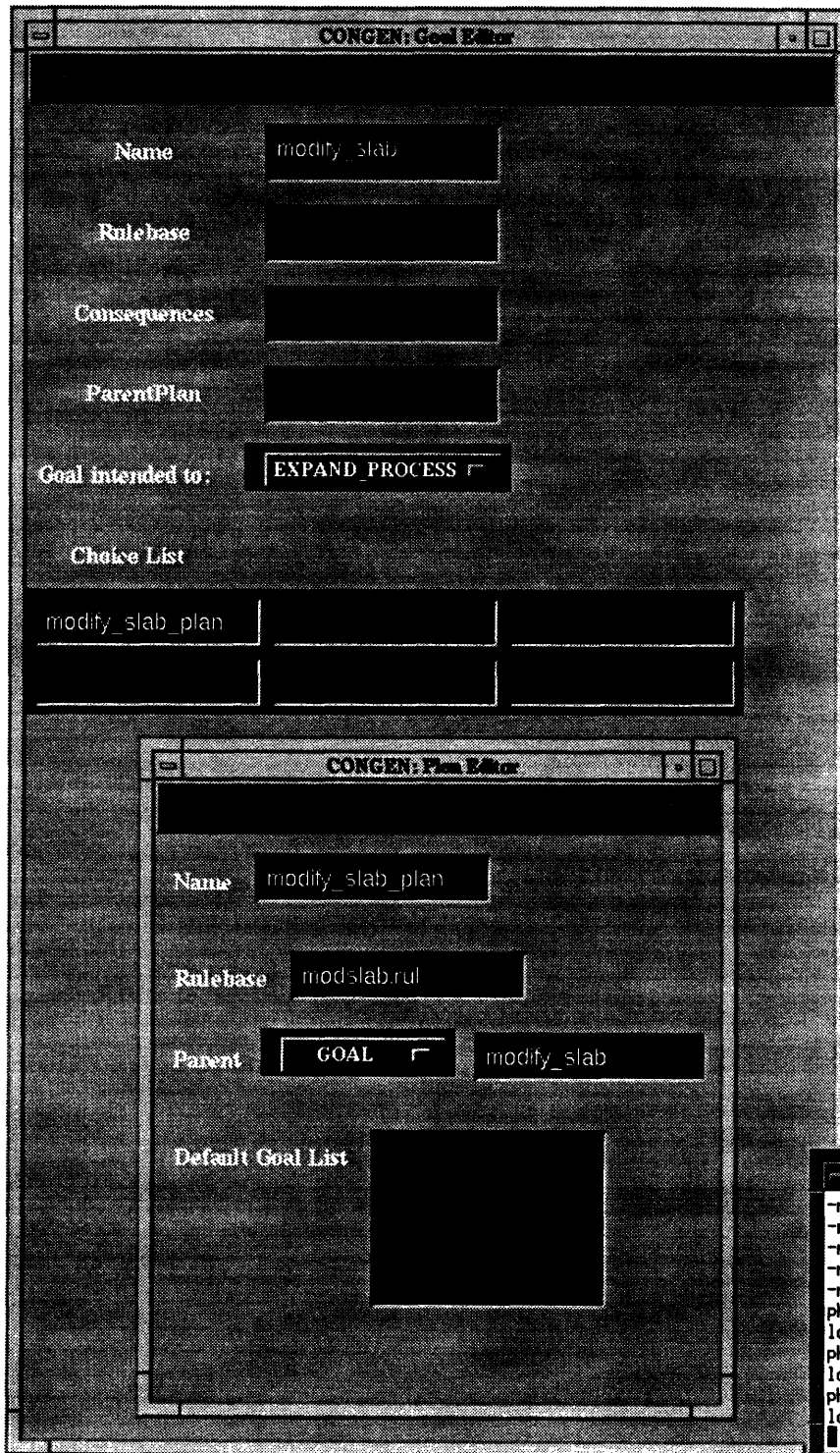


Figure 5-14: The technique of combining a dummy plan and a goal to fire a ruleset.

## 5.5 Summary

---

3. *Creating Tutorial.2.rul ruleset*
4. *Entering create-slab goal*
5. *Running Synthesizer*
6. *Running GNOMES and viewing the Slab's geometry*

The following chapter will provide an example involving multiple goals and plans.

---

## Chapter 6

# Tutorial III: Building a Box

This chapter provides a deeper knowledge of structuring an application in CONGEN. Whereas the previous chapter only deals with one Goal and one Class, this tutorial will introduce multiple goals and plans to prepare the user for a simple real-world application. The steps for developing a CONGEN application is also presented. Finally, this chapter closes with various CONGEN application approaches to solve the problem.

### 6.1 The Problem

The problem to be solved in this tutorial is to create a box, with six covers. After the box and the covers have been created, the geometry of the box must be shown in GNOMES.

Here are the important steps of structuring the application:

1. Prepare the application and the database.
2. Create a new application.
3. Create classes and rulesets.
4. Make the application.
5. Create goals and plans for the application.
6. Execute the Synthesizer.

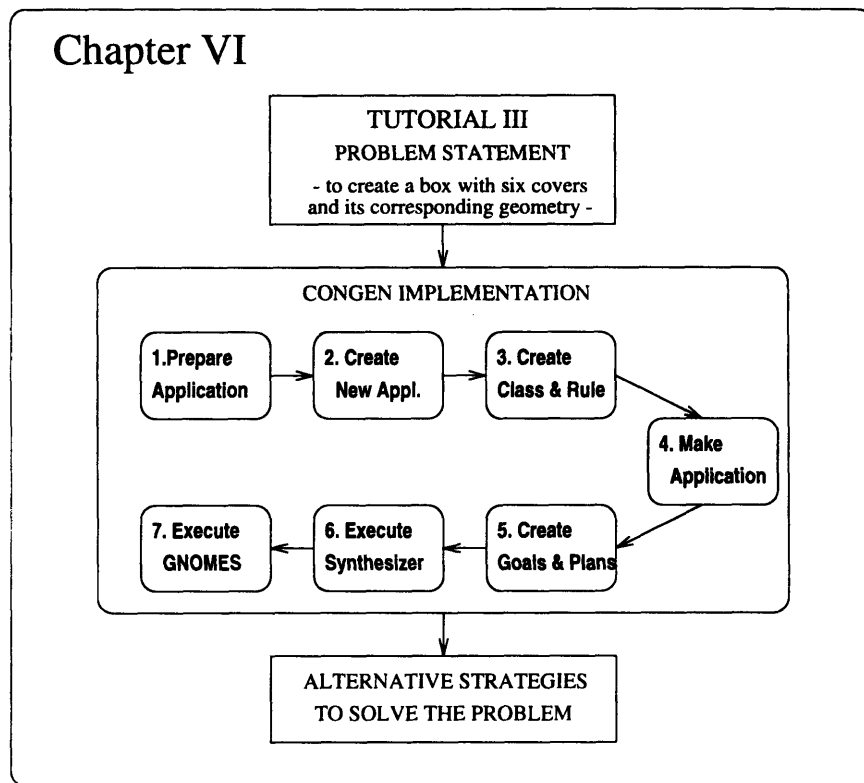


Figure 6-1: The roadmap of this chapter.

7. Execute the Geometric Modeler (GNOMES) and show the geometry.

These steps will be elaborated in more depth in the following sections.

Before the implementation, it is important to decompose the problem in tasks and objects. The decomposition will help in structuring the application with the combination of Goals, Plans, and Artifacts.

The intuitive steps to solve this problem are:

1. Get the dimensions (specifications and constraints).
2. Create the bounding box that will contain the covers.
3. Create the covers (top, bottom, east, west, north and south).
4. Move the covers to the corresponding location (geometrical information).
5. Show the box.



## 6.1 The Problem

---

To elaborate further, firstly we have to identify and define the unique objects. The crucial objects are: *the assembly - (the box)* and *the slabs - (the covers)*. The assembly object provides the final design product which has six slabs. The six slabs contain two kinds of information: the dimensions and the geometry of each slab to function as covers for the box. The dimensions of the cover slabs should correspond to the dimension part of the box covered.

Referring to the above steps, the logical GOAL-PLAN-ARTIFACT steps are:

1. *ROOT GOAL: Create the box*
2. *PLAN: Create box* which has three goals:
  - *GOAL: Create top-bottom covers* which in turn has a plan:
    - *PLAN: Create top-bottom cover plan* which consists of two goals:
      - \* *GOAL: Create top cover*
      - \* *GOAL: Create bottom cover*
  - *GOAL: Create east-west covers* which in turn has a plan:
    - *PLAN: Create east-west cover plan* which consists of two goals:
      - \* *GOAL: Create east cover*
      - \* *GOAL: Create west cover*
  - *GOAL: Create north-south covers* which in turn has a plan:
    - *PLAN: Create north-south cover plan* which consists of two goals:
      - \* *GOAL: Create north cover*
      - \* *GOAL: Create south cover*

The PLAN - GOAL - ARTIFACT hierarchy for this application is shown in figure 6-2.

This approach may not be the best solution to the problem. There are lots of ways in structuring the goal-plan-artifact combination for design. You, as the designer, are expected to exercise your creativity in building the application and structuring the concepts of CONGEN.

## 6.1 The Problem

---

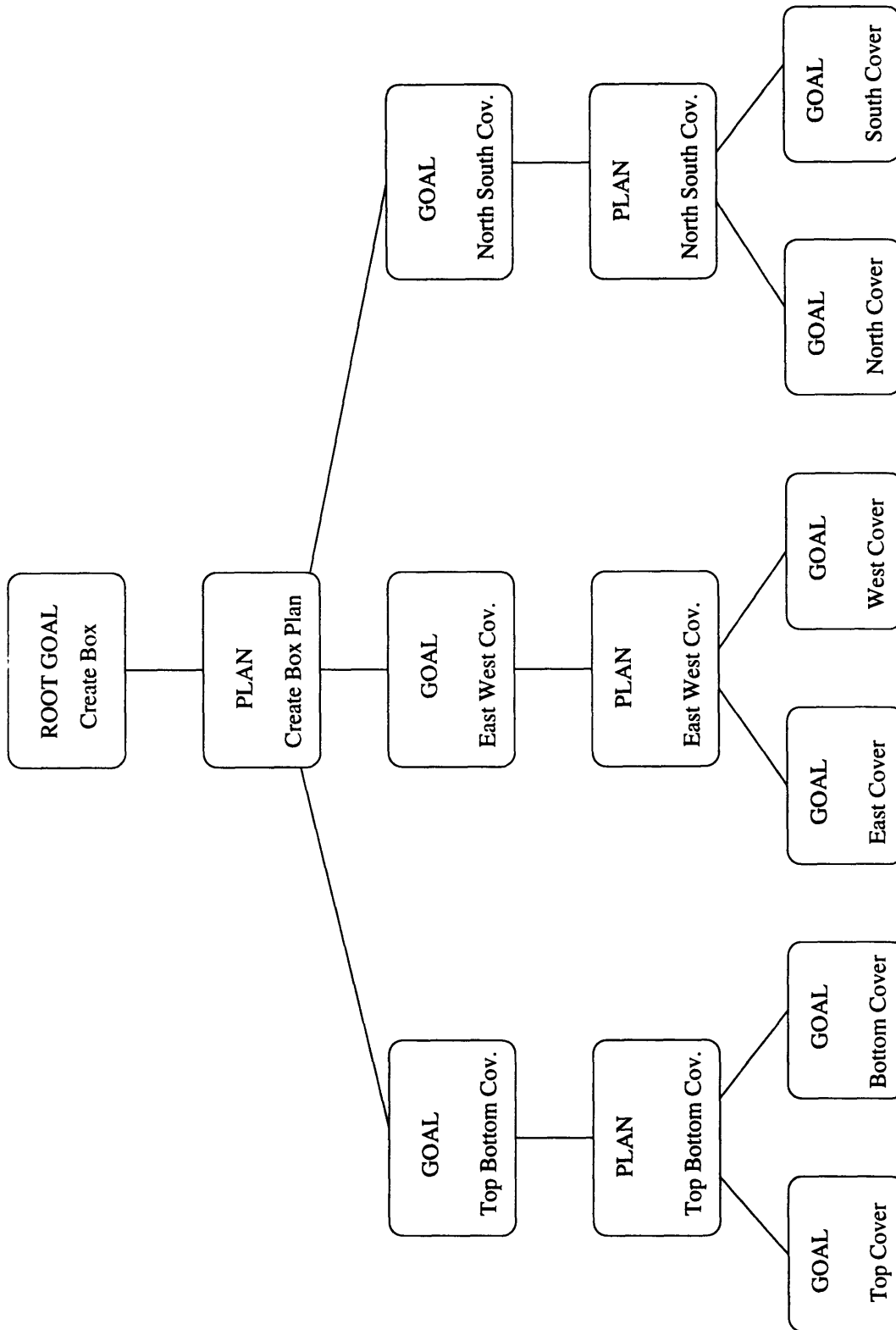


Figure 6-2: Tutorial 3 Application structure.

## 6.2 The Implementation

---

The next step is to develop the rulesets for the application. To create six covers, we need six rulesets for the creation of the slab and its geometry. Creating the rulesets conclude the preliminary step of developing the application. The next section will elaborate more on the implementation of this application.

## 6.2 The Implementation

To comply with the seven steps of creating a CONGEN application, we should start with:

### 6.2.1 Preparation

Before creating the tutorial application, you must do the following:

1. Have a specialized database volume prepared by your database administrator.
2. Set the `.cshrc` environment variable in the root directory to point to your database volume :

```
setenv EVOLID (the volume number of the database)
setenv CONDIR (the directory where your congen is installed)
setenv CONGEN_USER_AR (the directory where the class archive is installed)
setenv INCDIR (the directory where the congen include source files is placed)
```

3. Set the `.sm-config` EXODUS configuration file in the root directory to point to your database volume: `client*mount: (volume number) (port number)@(serverhost)`.
4. Create a special directory to run this tutorial such as: `mkdir /mit/gnomes/congen/tutorial3`.
5. Create a special directory to store the classes generated by this application: `mkdir /mit/gnomes/congen/ar`. Note that this directory must be named `ar` - archive. The `CONGEN_USER_AR` environment variable must point to this directory.
6. Change to the special directory that you have specified (`/mit/gnomes/congen/tutorial3`), and run `CONGEN` within that directory. This directory will store the rulefiles needed for this **BOX** application.

## 6.2 The Implementation

---

All of these actions can be done with the help of your database administrator and system manager. For more information about the environment variables, please refer to Appendix C. After you have completed these tasks, you are ready to build the application.

### 6.2.2 Creating a new application

Create the application by:

- Select **FILE** → **NEW** in the Main Console.
- Enter *Tutorial\_3* and press **OK**.
- Save the application.

### 6.2.3 Creating classes and rulesets

- *Creating Classes*

There are two classes to be created: the “Slab3” and the “Assembly” class:

#### 1. *Creating Slab3.*

From the MAIN CONSOLE window, select **KNOWLEDGE** → **PRODUCT DEFINITIONS**. Then in the PRODUCT KNOWLEDGE window, select **FILE** → **NEW CLASS** to create a new class. You should enter *Slab3* and press **OK**. In the CLASS EDITOR window, select **EDIT** → **PUBLIC**, then **FILE** → **NEW** in the next window. Enter:

```
dbint s_length;  
dbint s_width;
```

Press **SAVE**, **QUIT**, and **OK** in the WARNING window.

Exit the editor by selecting **FILE** → **QUIT** in the CLASS EDITOR window, and **FILE** → **QUIT** to go back to PRODUCT KNOWLEDGE window.

The result can be seen in figure 6-3. This class will be used for the covers.

## 6.2 The Implementation

---

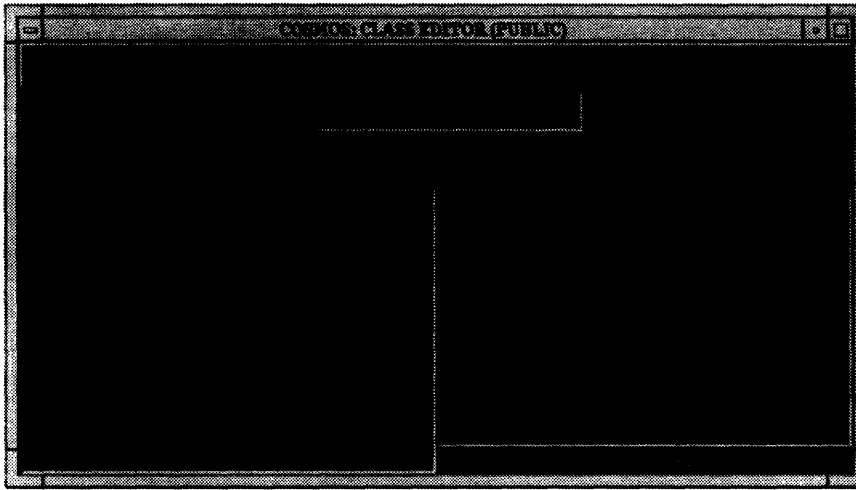


Figure 6-3: Slab3 class attributes.

- **NOTE:** We can use a class as the attribute of another class to implement the *part-of* relationship, for example:

```
CLASS COVER; // a paper slab.  
  
CLASS BOX{ // a paper box consisting of paper covers.  
COVER top_cover;  
COVER bottom_cover;  
}
```

The *part-of* relationship is not shown in the list of attributes in the CLASS EDITOR window. This is because all *part-of* relationships are explicitly and separately maintained by the system. The similar implementation of the relationship using CONGEN's ruleset is:

```
in the rulefile:  
CLASS BOX OBJ: $x  
...  
MAKE CLASS:COVER OBJ:COVERTOP  
MAKE CLASS:COVER OBJ:COVERBOT  
EXECUTE VAR: $rtn OBJ: $x make_part('top_cover','COVER','COVERTOP');  
EXECUTE VAR: $rtn OBJ: $x make_part('bottom_cover','COVER','COVERBOT')
```

## 6.2 The Implementation

---

What the *make-part* method does is to create a *part-of* link between the BOX and the cover. The relationship between the box and the cover is *top-cover*. The last two arguments in the *make-part* method refer to the classname of the child, and the instance name of the child.

2. *Creating Assembly.* Follow the same instructions as above, but with *Assembly* as the name of the new class. Then for the attributes, enter:

```
dbint a_length;  
dbint a_width;  
dbint a_depth;
```

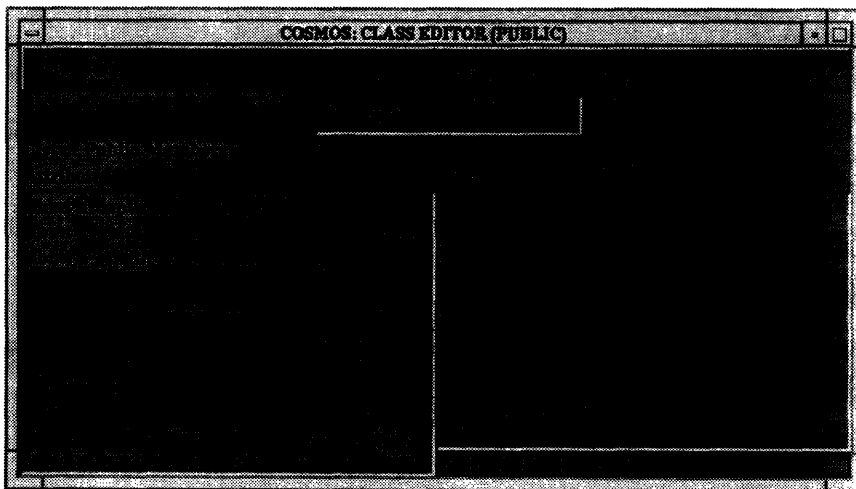


Figure 6-4: Assembly class attributes.

- **NOTE:** Because “Assembly” class is connected explicitly to the Plan - *create\_box\_plan* the class must set its plan default to the corresponding plan. The steps are explained below. The reason why it has to maintain an explicit connection is because the artifact needs the subplan to continue with the design procedure.

In other words, the “Assembly” class still needs to attach itself to the six covers, but the covers haven’t been created yet. In order to provide a way to expand the process and continue with the design procedure, the “Assembly”

## 6.2 The Implementation

---

must have a subplan linked explicitly. To do this, select the **EDIT → ARTIFACT DEFAULTS** in the **CLASS EDITOR** window, and set the values as shown in the figure 6-5.

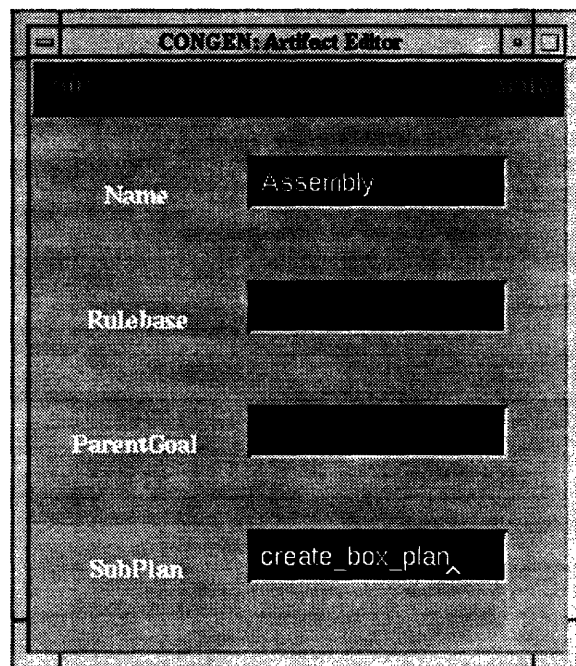


Figure 6-5: Assembly class Artifact Defaults.

Save the class and now you can compile the classes to make it ready for the application. The result can be seen in figure 6-4. This class will be used as the bounding box.

- *Creating Rulesets*

There are six rulesets that have to be defined:

1. **Top cover ruleset.**

By using the RuleFile Editor or EMACS, enter:

```
(RULE: createtop 1000
IF
((CLASS: Slab3 OBJ: $x
```

## 6.2 The Implementation

---

```
((s_length == 0) AND
(s_width == 0))
)
AND
(CLASS: Assembly OBJ: $y
(((a_length == $len) AND
(a_width == $wid)) AND
(a_depth == $dep))
))
THEN (
(MODIFY (OBJ:$x
(s_length $len)
(s_width $wid)
(s_depth 2)
) 1000 0.001)
)
COMMENT:"Rule to create the top cover for tutorial 3")

(RULE: top_geometry 10
IF
(CLASS: Slab3 OBJ: $x
(((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
)
THEN (
(EXECUTE VAR: $rtn OBJ: $x create_geometry(16,'slabgeom2'))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(0.0,0.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(0.0,0.0,30.0))
(EXECUTE VAR: $rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the top ")
```

These top cover rules state that :

- (a) If the top slab hasn't been created, then pass the length and width of the box to create the top slab dimension.
- (b) Modify the top slab dimension to be the same as the box's length and width.



## 6.2 The Implementation

---

- (c) If the geometry of the top slab hasn't been changed, then create the geometry of the top slab.
- (d) Set the geometry dimension the same as the slab itself, do not rotate the slab, but translate the top cover up as high as the depth of the box.

### 2. Other covers' rulesets.

The other covers follows the logic of the previous example. For example, to create the bottom cover, the rule must:

- (a) Create the bottom slab cover, then pass the length and width of the box to create the top slab dimension.
- (b) Modify the bottom slab dimension to be the same as the box's length and width.
- (c) If the geometry of the top slab hasn't been changed, then create the geometry of the bottom slab.
- (d) Set the geometry dimension the same as the slab itself, do not rotate or translate the slab.

The other thing to remember is to understand the basic geometry of GNOMES. For example, when a slab is created, the user must know where it is created and the corresponding amount of translation and rotation needed to place the slab's geometry.

The remaining rulesets are listed in the Appendix D, along with the other tutorials' listings.

### 6.2.4 Making the Application

After defining all the classes and rulesets, it is time to compile the classes to make it ready for CONGEN. When the compilation is finished, please remember to save the application permanently to the database.

## 6.2 The Implementation

---

### 6.2.5 Creating goals and plans for the application

Now it is time to enter the process knowledge into the database. After you have summoned the DDH EDITOR window, you should enter:

1. Root Goal: *create\_box*.

This goal creates the artifact of an “Assembly” as shown in figure 6-6. Even though there is no rulefile to define the parameters of the “Assembly”, the Specifications Editor will provide the parameters needed for the dimension of the box. Section 6.2.6 will explain further about the Specifications entry.

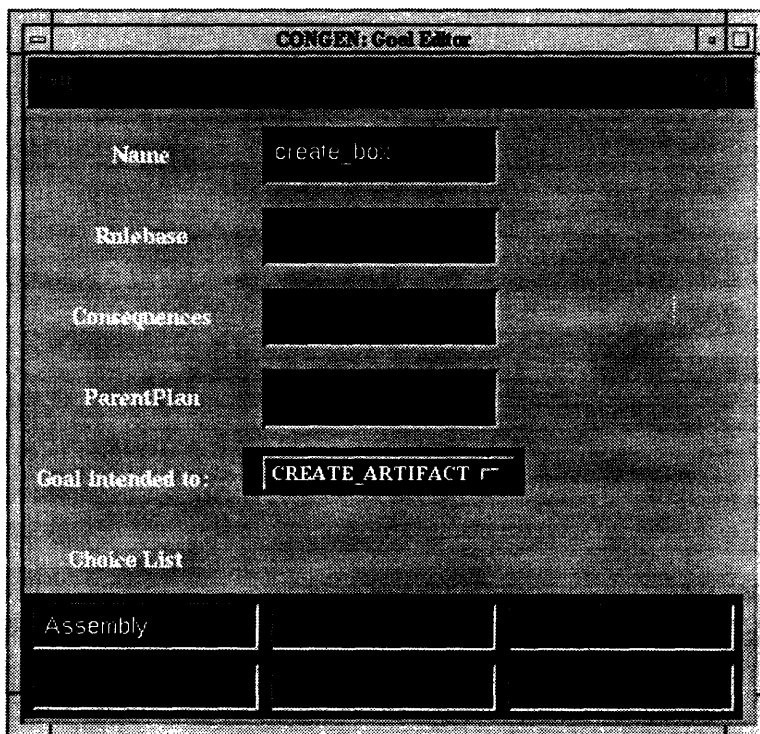


Figure 6-6: The create\_box root goal for the application.

2. Plan: *create\_box\_plan*.

This plan acts as the manager of the goals to be executed by this application. This plan is linked explicitly to the “Assembly” artifact after it is created by the root goal (refer to the previous section).

## 6.2 The Implementation

---

*Create-box-plan* will consist of three subgoals: create the top-bottom, east-west, and north-south covers. See figure 6-7 for the definition of this plan.

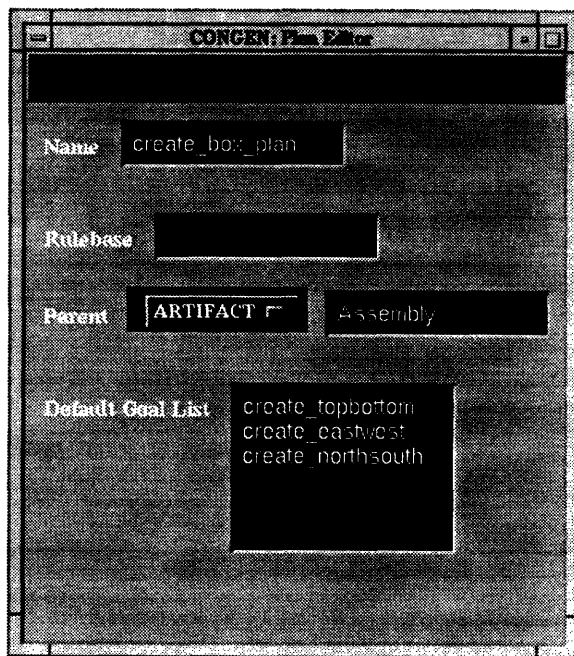


Figure 6-7: The create\_box\_plan for the application.

3. Goal: *create\_northsouth*, *create\_topbottom*, *create\_eastwest*.

Below the create\_box\_plan, we divide the tasks to three subgoals to create the pair of covers. The example of *create\_northsouth* goal explains about how this goal is used to expand the process for further refinery of subplans, and ultimately, the creation of each cover complete with the geometry information.

See figure 6-8 for the example of the *create\_northsouth*. The user is expected to create two more goals similar to this example to tackle the task of creating top-bottom, and east-west covers.

4. Plan: *create\_northsouth\_plan*, *create\_topbottom\_plan*, *create\_eastwest\_plan*.

Because we still have to create two cover slabs for each goal, it is crucial that we use another plan to contain the tasks of creating each cover slab. The reason behind this

## 6.2 The Implementation

---

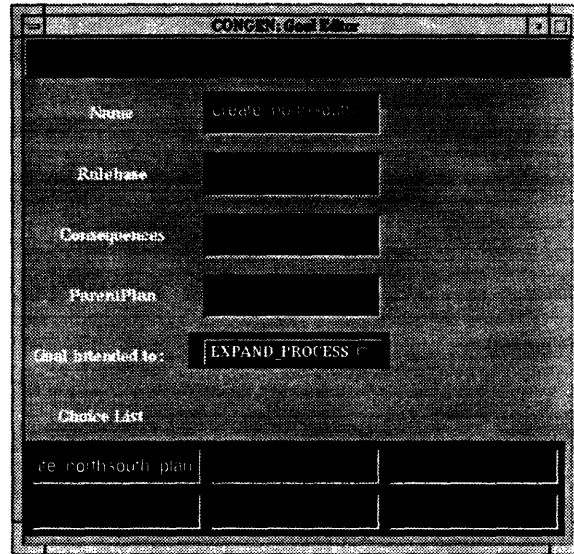


Figure 6-8: The create\_northsouth goal for the application.

is that a goal can only create one artifact. To create multiple instances of an artifact, you must use rulesets.

To fulfill the need for creating two cover slabs, the goal must expand the process by incorporating another plan. The subplan contains the goals / tasks of creating each cover slab.

The example of *create\_northsouth\_plan* provides a better view of how the tasks are organized. It can be seen in figure 6-9. You should create two more plans, similar to this example, for the *create\_topbottom\_plan*, and *create\_eastwest\_plan*.

5. Goal: *create\_north*, *create\_south*, *create\_east*, *create\_west*, *create\_top*, *create\_bottom*.

These are the end goals that create the corresponding covers and provide the geometry of the cover. The consequence field contains the name of the ruleset that performs the above tasks.

The example of *create\_north* goal is shown in the figure 6-10. This example contains one rulefile which is used to control the creation of the **Slab3** artifact and its geometrical information. Again, you are expected to type in the five remaining goals to create the other five cover slabs.

## 6.2 The Implementation

---

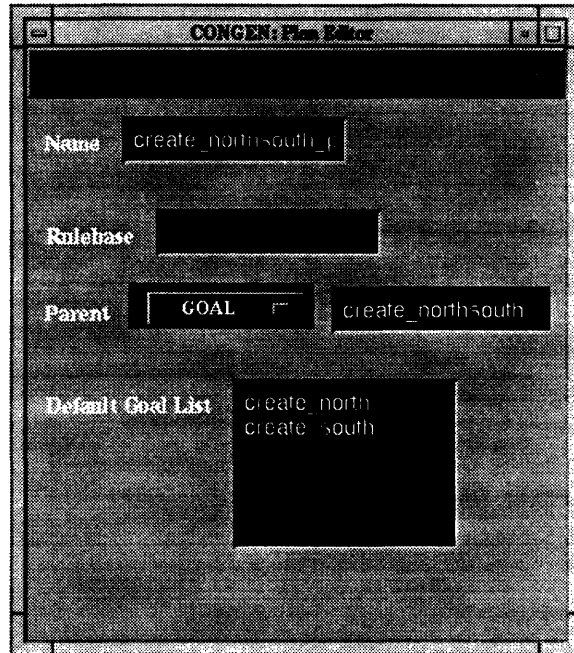


Figure 6-9: The create\_northsouth\_plan for the application.

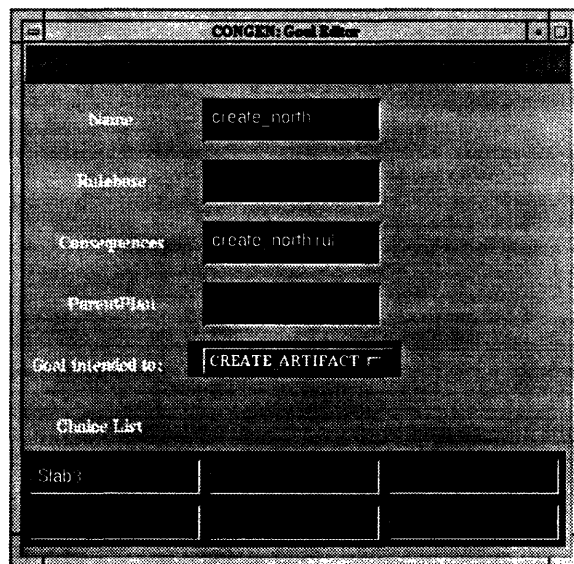


Figure 6-10: The create\_north goal for the application.

## 6.2 The Implementation

---

### 6. DDH window.

After everything has been entered correctly, you should save the application before continuing to execute the Synthesizer. Figure 6-11 shows all the information entered into the DDH Editor.

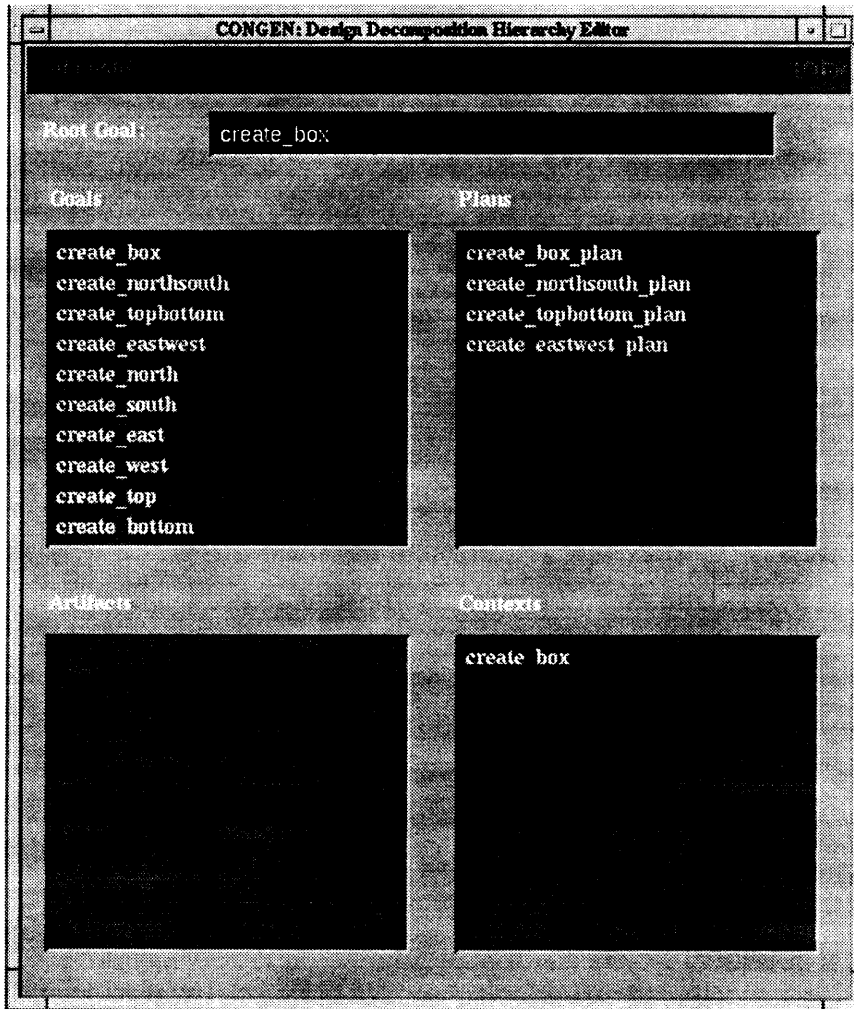


Figure 6-11: The DDH EDITOR window after everything has been entered.

### 6.2.6 Executing the Synthesizer

After all the classes, rulesets, and the process knowledge have been saved, the flow of the application should be checked using the Synthesizer. Before running the application

## 6.2 The Implementation

---

using the Synthesizer, you must enter the Specifications value for the Assembly box.

To enter the Specifications, press **SPECIFICATIONS** → **INPUT CONTEXT** in the **MAIN CONSOLE** window. When the **SPECIFICATION EDITOR** window pops up, you can start entering the Specifications item by selecting **EDIT** → **ADD AN ITEM**.

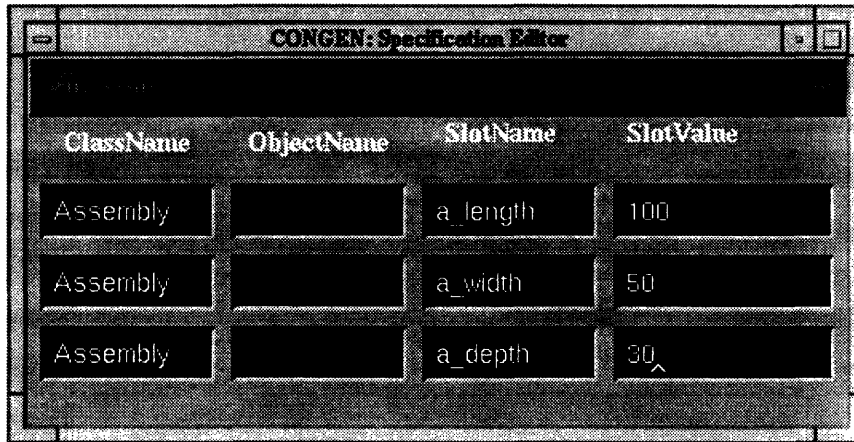


Figure 6-12: The **SPECIFICATION EDITOR** window after everything has been entered.

Everytime you select that submenu, four slots of specifications will be shown. Under the **ClassName** field you should enter *Assembly*. You should not enter anything under the **ObjectName** field because we want the Specification to be shared by every instance of the class “Assembly”. In the **SlotName** field enter *a\_length* pointing to the attribute of “Assembly”. Lastly, you should enter *100* in the **SlotValue** field as the value of the attribute. Continue entering the values as shown in figure 6-12.

- **NOTE:** When you do not specify any **ObjectName** field entry in the Specification item, it means that this specification is valid for every instance of the class *Assembly*. To limit the Specification scope to a particular instance, you must enter the name of the instance in the **ObjectName** field.

After you are finished, select **FILE** → **SAVE** in the **SPECIFICATION EDITOR** window to transmit the information temporarily to the active application, and finally save the application.

To run the application, you should execute the Synthesizer from the **MAIN CONSOLE**

## 6.2 The Implementation

---

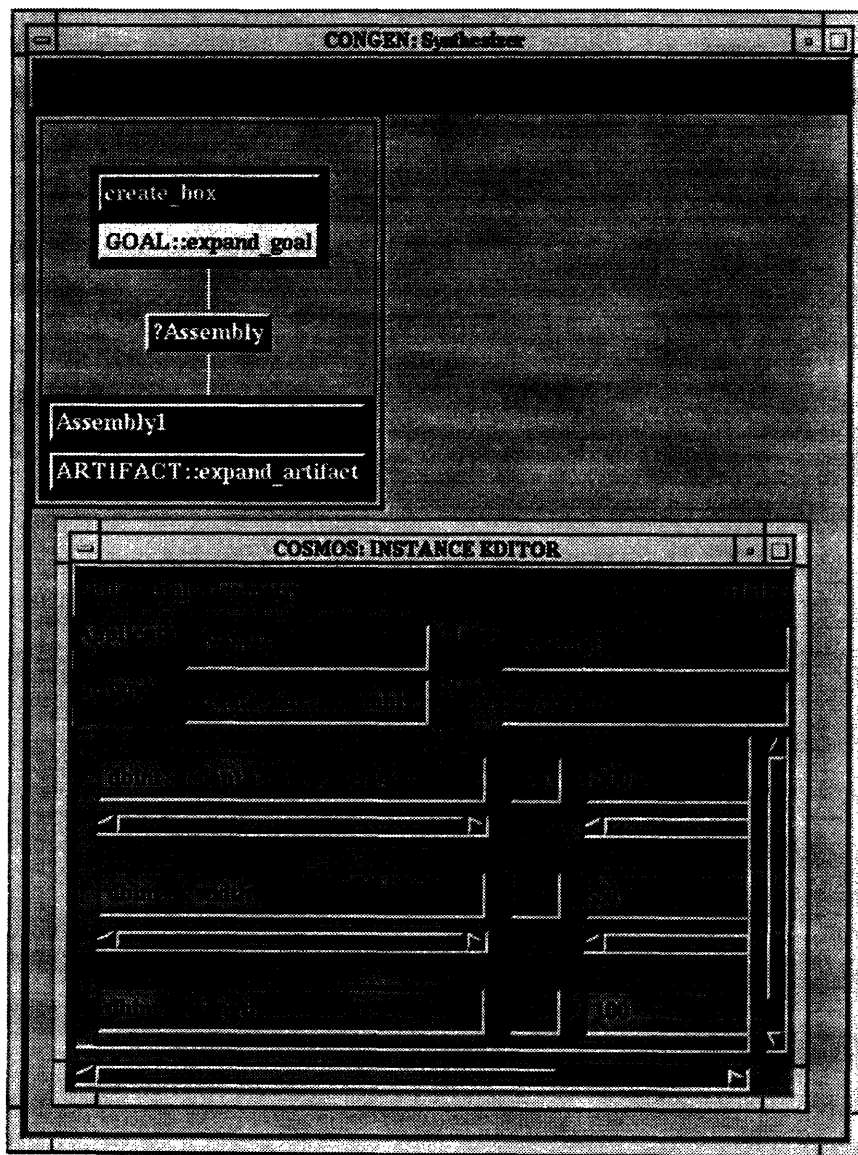


Figure 6-13: The SYNTHESIZER window with the created instance of Assembly.

window.

In the Synthesizer, you should press the button **GOAL::expand\_goal** in the *create\_box* item followed by a click on the **?Assembly** button. The action creates an instance of the "Assembly" with the required Specifications entered above. To confirm whether the Specifications values have been entered correctly and passed to the instance of "Assembly", you should refer to figure 6-13.



## 6.2 The Implementation

---

After all the steps have been performed correctly, you may continue executing the Synthesizer by clicking at the **ARTIFACT::expand\_artifact** button below the **Assembly1** item. Please remember that you can always access the editor of the corresponding object by clicking on the upper buttons, e.g.: if you click on the *Assembly1* button, the ARTIFACT EDITOR window will pop up, as seen in figure 6-13.

To continue, you should click the **expand\_artifact** button will pop up four more buttons, the **create\_box\_plan** button, the **create\_topbottom**, the **create\_eastwest**, and the **create\_northsouth** goal buttons in sequence.

After expanding the goal of *create\_topbottom*, and clicking on the *?create\_topbottom\_plan* button, more buttons will be shown on the screen. Exploring this path further will create the top cover slab for the box, as shown in figure 6-14.

Continuing this example and traversing every path available for the application will create five more cover slabs. The overall view of the Synthesizer after every path has been traversed can be seen in figure 6-16. You can check whether the corresponding rulefiles have been fired from the XTERM window. It should give an output like the one shown in figure 6-15.

The DDH EDITOR window can also provide you with the information about all created artifacts, slab instances, and new contexts, such as those shown in figure 6-17. The DDH Editor also gives you the capability to examine every context listed in the window.

To explore different contexts, you can click on any item on the list of contexts. For example, if you click on the *create\_south.Slab3* context, you are accessing the state of the application when you create the south cover slab. A DDH BROWSER window will pop up to show the decision tree of the design path at that instant.

What the DDH Browser shows you is the design decision point when the *create\_south.Slab3* is created. The *create\_south.Slab3* context refers to the decision of creating a Slab to cover the South area of the “Assembly”. Before the South Slab is created, there have been two more slabs created with the instance name of *Slab31* and *Slab32*. Those slabs refer to the top and bottom slabs. Tracing the path, we found that the deepest path ends with *Slab33*. This means that this context pursues that decision as the last decision point.

## 6.2 The Implementation

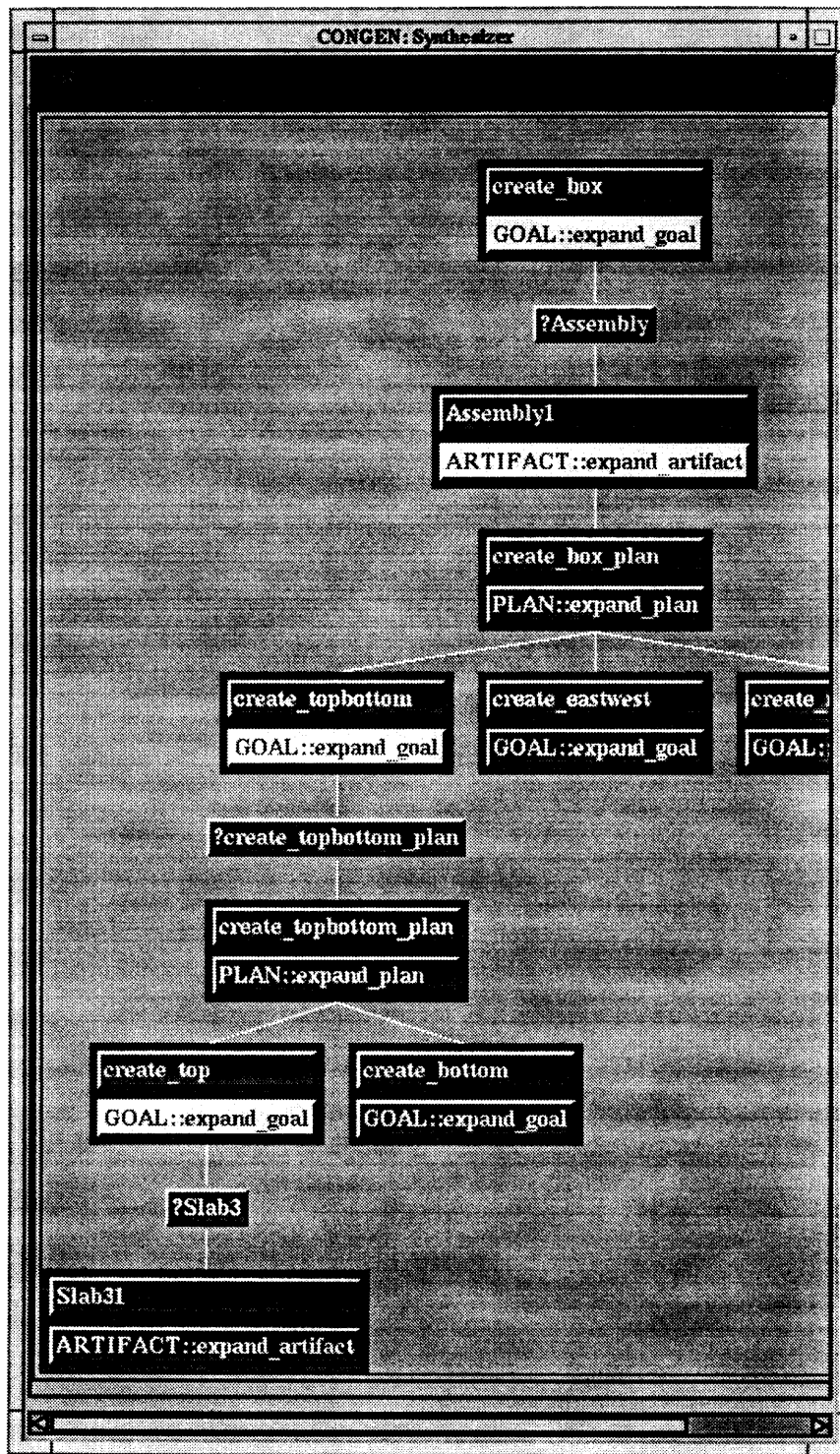


Figure 6-14: The SYNTHESIZER window after the top cover slab has been created.

## 6.2 The Implementation

```
genesh: GRAPHITI --> congen
Done with GManager constructor
OPENING DATABASE DBTutorial_3
Getting name : Tutorial_3
FINISHED OPENING DATABASE DBTutorial_3
* Rule-base loaded *
** Rule selected: createtop **
** Rule selected: top_geometry **
* Rule-base loaded *
** Rule selected: createbottom **
** Rule selected: bottom_geometry **
* Rule-base loaded *
** Rule selected: createsouth **
** Rule selected: south_geometry **
* Rule-base loaded *
** Rule selected: createnorth **
** Rule selected: north_geometry **
* Rule-base loaded *
** Rule selected: createleft **
** Rule selected: left_geometry **
* Rule-base loaded *
** Rule selected: createright **
** Rule selected: right_geometry **
```

Figure 6-15: The XTERM window showing all the rulefiles fired.

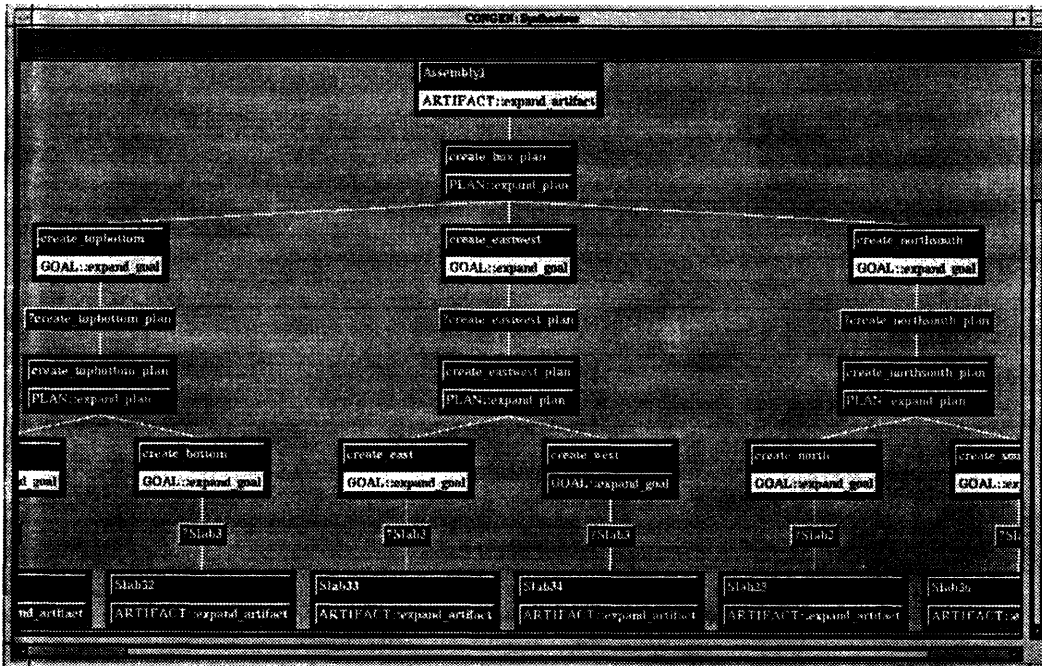


Figure 6-16: The SYNTHESIZER window after every path has been traversed.

## 6.2 The Implementation

---



Figure 6-17: The DDH EDITOR window after every path has been traversed.

### 6.2.7 Executing the Geometric Modeler (GNOMES) and show the geometry

To execute GNOMES Geometric Modeler, select **EXECUTE** → **GEOMETRIC MODELER** in the MAIN CONSOLE window. After the GNOMES window is shown, go back to the Synthesizer and select **GEOMETRY** → **SHOW GEOMETRY**. After selecting the **SHOW GEOMETRY**, you should go back to the GNOMES window. The GNOMES window will display the box as shown in figure 6-19.

### 6.3 Other Solutions to the Problem

---

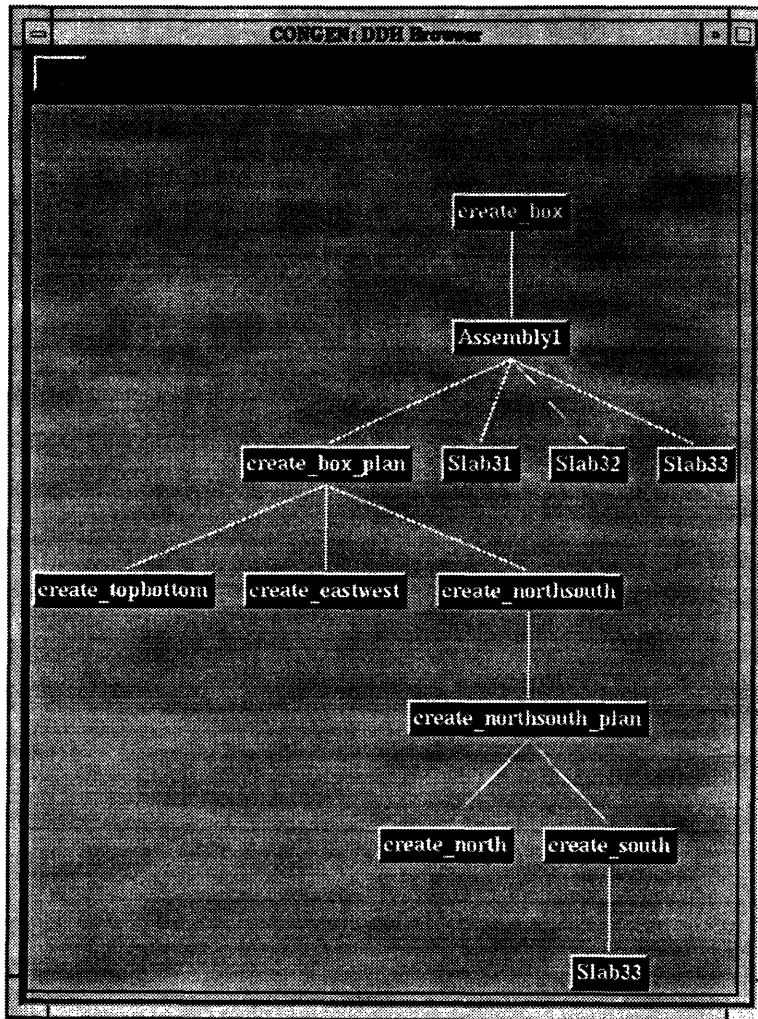


Figure 6-18: The DDH BROWSER window after *create\_south.Slab3* is pressed.

### 6.3 Other Solutions to the Problem

The more complex the problem, the more application alternatives you can create to solve it. Below is the list of the various solutions. The listing of the alternative rulesets can be seen in Appendix D.

1. *One Root Goal, Two Classes, One Rulefile*

Instead of creating many plans and goals, we can implement the solution using only one long rulefile to create the “Assembly” and all the cover slabs. This is a very

### 6.3 Other Solutions to the Problem

---

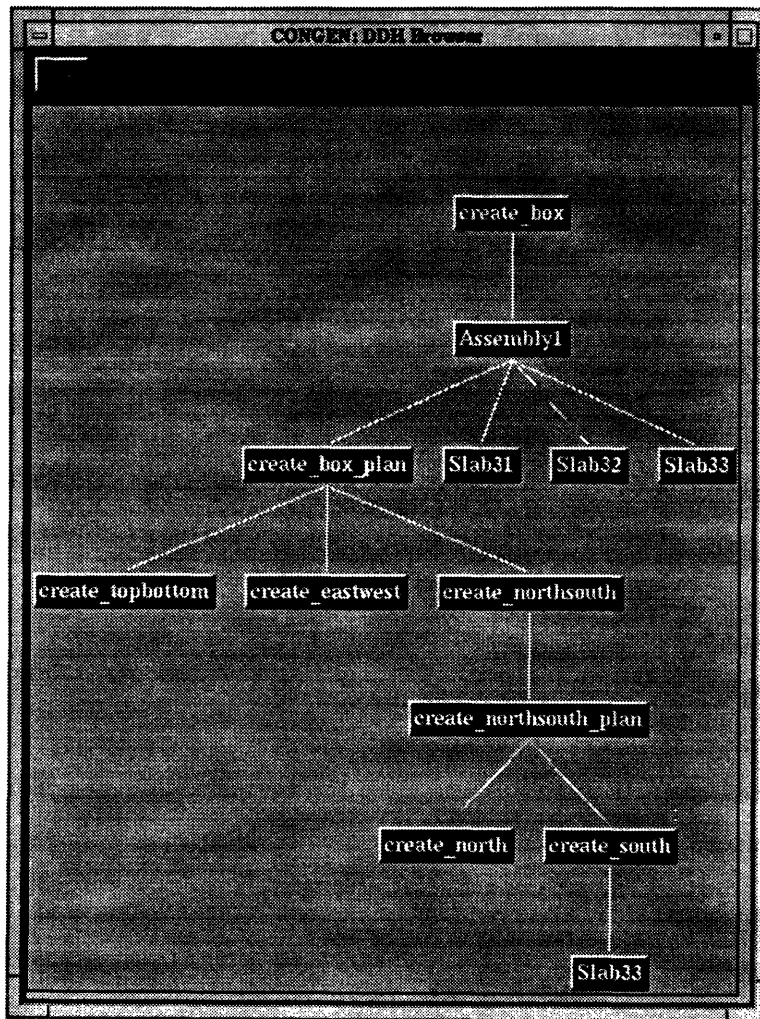


Figure 6-19: The Gnomes window with the box.

simple solution. The specifications are the same for the “Assembly” class.

An example of the rulefile follows:

```
RULE: createslabs 1000
IF
(CLASS: Assembly3 OBJ: $y
(((a_length == $len) AND
(a_width == $wid)) AND
(a_depth == $dep))
)
```

### 6.3 Other Solutions to the Problem

---

```
THEN (  
  (MAKE (CLASS: Slab OBJ: top  
    (s_length $len)  
    (s_width $wid)  
    (s_depth 2)  
  ))  
  (MAKE (CLASS: Slab OBJ: bottom  
    (s_length $len)  
    (s_width $wid)  
    (s_depth 2)  
  ))  
  (MAKE (CLASS: Slab OBJ: north  
    (s_length $dep)  
    (s_width $wid)  
    (s_depth 2)  
  ))  
  (MAKE (CLASS: Slab OBJ: south  
    (s_length $dep)  
    (s_width $wid)  
    (s_depth 2)  
  ))  
  (MAKE (CLASS: Slab OBJ: east  
    (s_length $len)  
    (s_width $dep)  
    (s_depth 2)  
  ))  
  (MAKE (CLASS: Slab OBJ: west  
    (s_length $len)  
    (s_width $dep)  
    (s_depth 2)  
  ))  
  )  
  COMMENT:"Rule to create the covers for alternative tutorial")  
  
  (RULE: top_geometry 10  
  IF  
    ((CLASS: Slab OBJ: $x  
    (instancename == "top")) AND  
    (CLASS: Slab OBJ: $x  
    ((s_length == $len) AND  
      (s_width == $wid)) AND  
      (length != $len))  
    ))  
  THEN (  

```

## 6.4 Summary

---

```
(EXECUTE VAR: $rtn OBJ: $x create_geometry(16,'slabgeomtop'))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(0.0,0.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(0.0,0.0,30.0))
(EXECUTE VAR: $rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the top "
```

etc... for five more geometry.

### 2. *One Root Goal, One Plan, Six Subgoals, Six Rulefiles*

The complex structure explained in the previous section can be simplified by using:

- Root Goal: *Create\_box*
- Plan: *Create\_box\_plan* comprised of six subgoals:
  - Goal: *Create\_top* which has the consequence rulefile *create\_top.rul*.
  - Goal: *Create\_bottom* which has the consequence rulefile *create\_bottom.rul*.
  - Goal: *Create\_east* which has the consequence rulefile *create\_east.rul*.
  - Goal: *Create\_west* which has the consequence rulefile *create\_west.rul*.
  - Goal: *Create\_north* which has the consequence rulefile *create\_north.rul*.
  - Goal: *Create\_south* which has the consequence rulefile *create\_south.rul*.

## 6.4 Summary

Tutorial 3 introduced many new concepts including:

1. *Explicit connection between an Artifact and a Plan*
2. *Structuring information with Goals and Plans*
3. *Various ways to tackle the problem*



## 6.4 Summary

---

4. *Specifications Editor usage*

5. *DDH Browser*

The next chapter will deal with the real world application of CABIN DESIGN.

---

## Chapter 7

# Tutorial IV: CABIN DESIGN

## Application

This chapter will present a real world application designed to help Structural Engineers. The application will incorporate all of the concepts explained in the preceding chapters, including the application structure, the goal-plan-artifact relationship, and the geometry of the objects. Section 7.1.1 provides the flow of the application to solve the problem. Section 7.1.2 presents the knowledge acquisition process in completing this application. Section 7.1.3 overviews the basic application vocabulary - classes. Section 7.1.4 explains about various aspects of the Geometry class in CONGEN. Finally, this chapter gives the steps toward completing the application.

### 7.1 The Problem

The problem posed in this chapter is the process of designing a simple single-room cabin. The design of the cabin itself actually spans from the Architectural, Structural, Mechanical, and Contractor. In this case, the area chosen to be the focus of the application is the Structural Engineering. Structural Engineers (for this project) will determine the structural system and structural members to be used, e.g. foundation, beams, columns, and trusses. These members are designed to sustain horizontal loads such as wind loads,

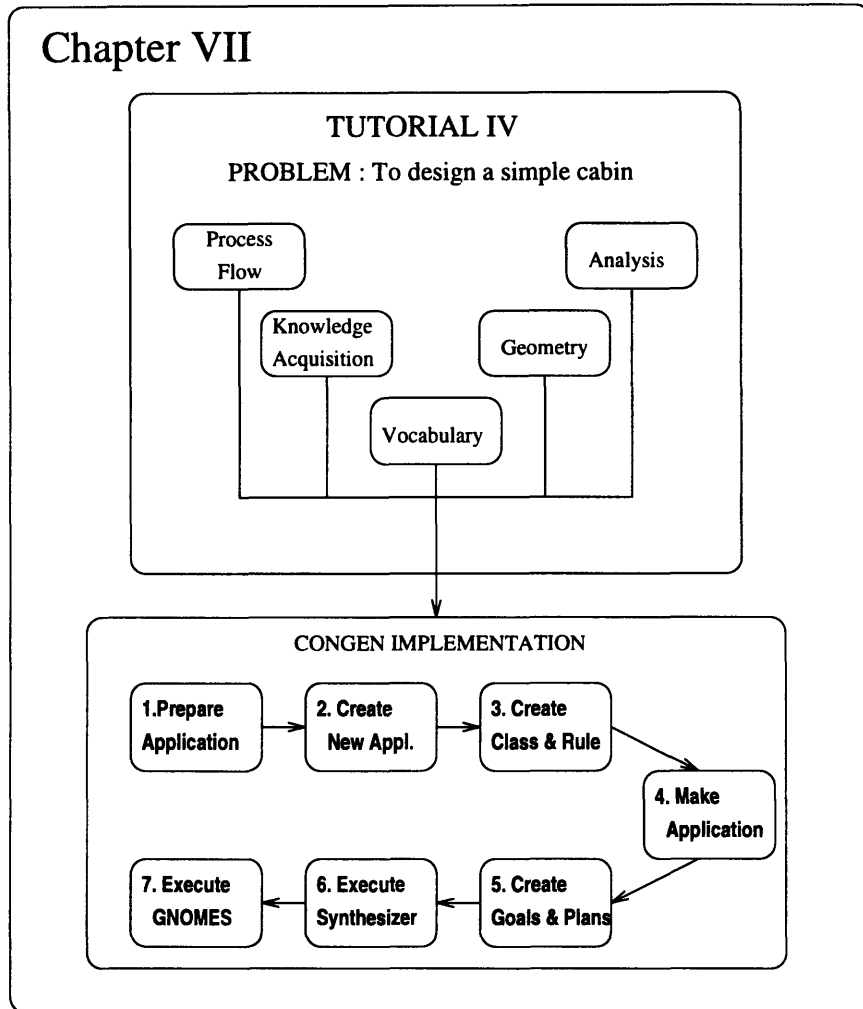


Figure 7-1: The roadmap of this chapter.

and vertical loads, such as dead load and live load.

The cabin itself has a specified dimension, which is done by the architect. The building plan can be seen in figure 7-2.

### 7.1.1 Process Flow

The structural design process aims to produce a safe, and serviceable structural system satisfying several constraints, such as cost, geometry, loads, etc. The structural design process can be divided into three subtasks [9]:

7.1 The Problem

---

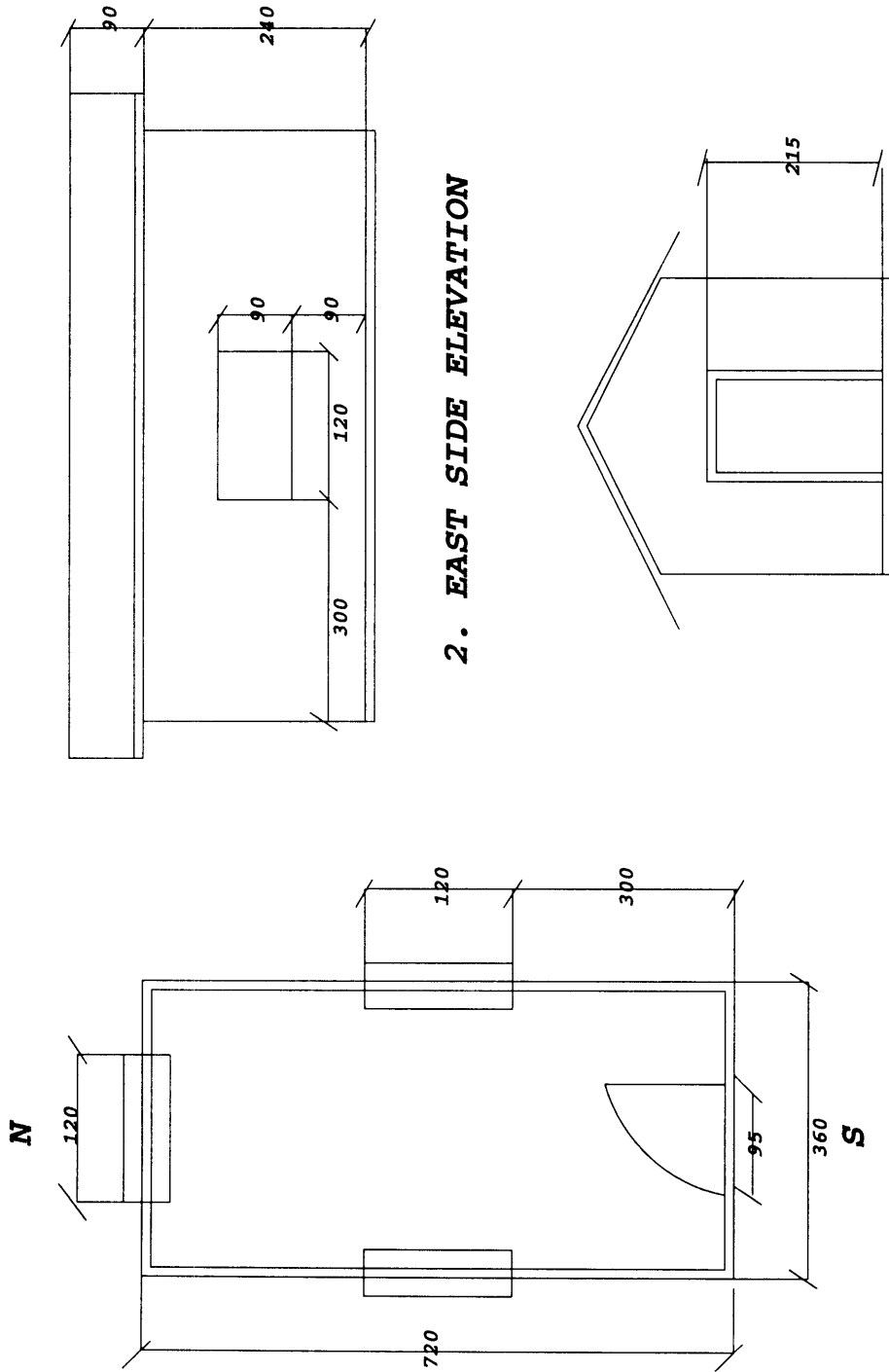


Figure 7-2: Cabin Architectural plan.

## 7.1 The Problem

---

1. *Preliminary Design.* The conceptual design of a structure is closely related to the synthesis of the design alternatives fulfilling the constraint requirements. The preliminary design stage primary task is to decompose the whole structure into smaller substructures. In addition, the preliminary constraints must be satisfied. Ultimately, the process will choose one of the preliminary design alternatives.
2. *Analysis.* The analysis stage simulates the response of the selected alternative according to different environmental effects. The tasks of transforming the alternative into a mathematical model, utilizing structural analysis procedures, and evaluating the results are the focus of this stage.
3. *Detailed Design.* This stage further refines the selection process of different members in the system to satisfy all the constraints required. There are several subproblems regarding this stage, such as main structural members' selection, supporting members' selection, and designing connections between the members in the system.

The three separate subtasks are generally not sequential. In practice, they may be interlinked and rehashed according to the needs of the designer. In the CONGEN application, the structural design processes is represented by the subtasks performed in the Detailed Design stage. However, the other stages - the Preliminary Design and the Analysis stages - are directly integrated into the application and interlinked with the Detailed Design stage. Therefore, the three stages are transparent to the user of this application.

The Cabin Design process consists of the following stages :

1. **Setting the Beam and Column Grid.** Setting beams and columns grid stage is decomposed into further subtasks, such as setting the number of columns, the number of girders, and the number of supporting beams for the structure. In this stage, the layout of the structure is determined by the main structural members. All other members are to follow the basic configuration of the beam-column grid.
2. **Setting the Truss System.** After putting the beams and columns in place, the trusses are created according to the main members. For example, in four-column

## 7.1 The Problem

---

cabin, we can have two trusses, while in six-column cabin, we can add one more truss in the middle, making it a three-truss roof. Moreover, this stage also defines the number of purlins involved in the roof to sustain the loads further.

3. **Setting the Walls.** Setting the walls consists of selecting the type of wall for each direction. For example, setting the north wall will select the type of wall, such as shear, brick, or wall with an opening.
4. **Setting the Foundation.** This stage provides the user the capability of selecting the foundation type. However, the foundation type depends on the number of columns and other considerations. The available foundation types are spread foundation, mat foundation, and strip footings.

For the task structure of this problem, refer to figure 7-3.

### 7.1.2 Knowledge Acquisition

Outlining the structural design processes and integrating them into a system in CONGEN requires a mix of structural design knowledge, real-life experiences, and creativity. The representation of conceptual design knowledge consists of the topology, geometry, structural functionality, structural behavior of the members, and analysis of the overall structure [11]. The design knowledge in this application combines the expertise of a real-life engineer and the textbook information.

The knowledge base implemented in CONGEN covers problem-solving rules, and facts or intuition from the expertise of a real-life engineer. It is stored as *If-Then rules* in the COSMOS module. COSMOS solves the problem in two steps: First, it incorporates the knowledge base with the working memory - the task specific data for the problem. Secondly, COSMOS combines the knowledge base and the working memory in the Inference Engine (IE). IE applies some control mechanism on the knowledge base and the working memory in order to achieve a feasible conclusion.

The engineer's expertise in this system uses the skills to achieve a feasible solution utilizing the heuristic reasoning strategy. This type of reasoning which is also called "rules

## 7.1 The Problem

---

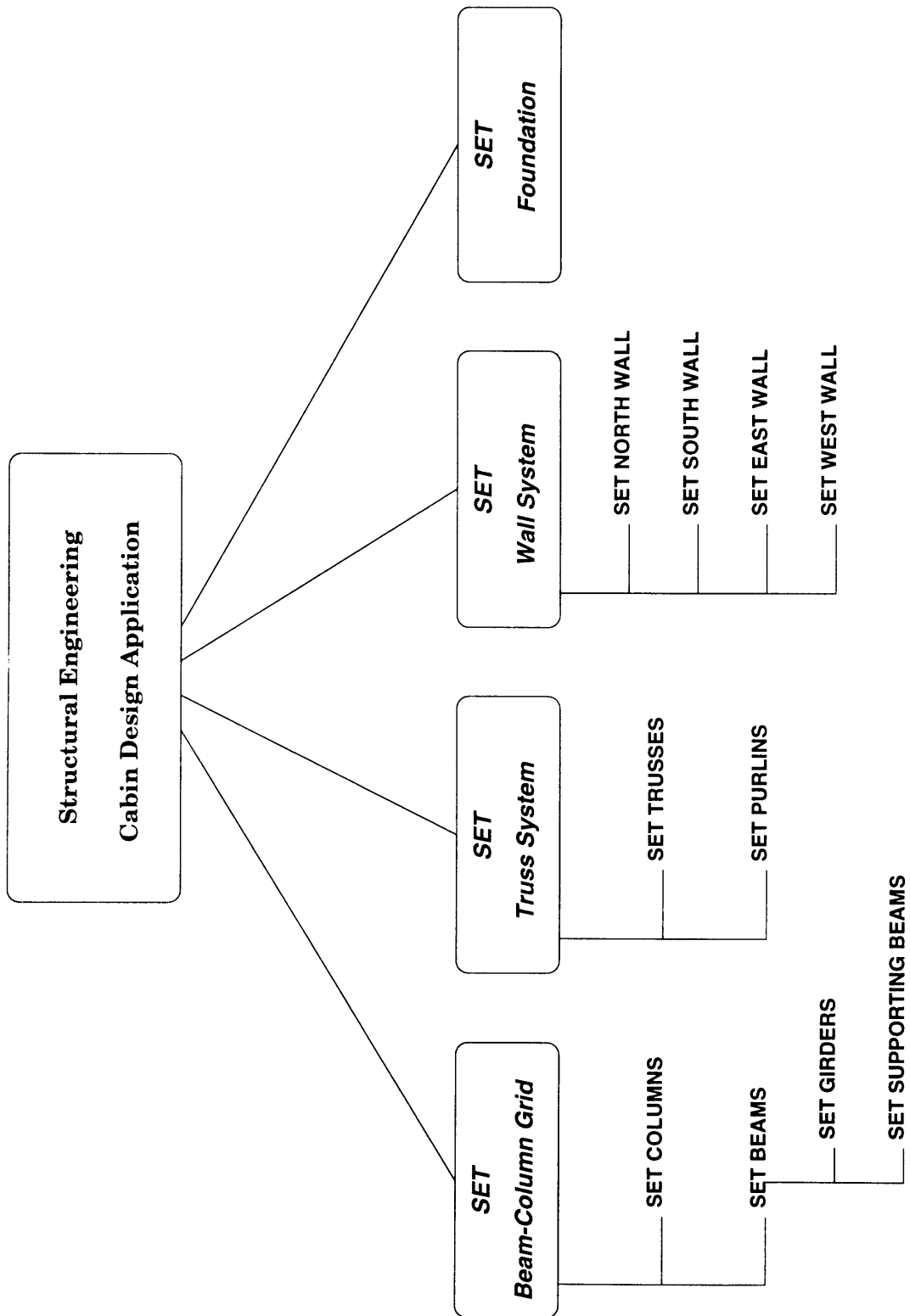


Figure 7-3: Cabin Design Application structure.

## 7.1 The Problem

---

of thumb” or “expert heuristics” provides ways and solutions to the expert to arrive at a good decision. COSMOS utilizes *forward chaining* and *backward chaining* methods to check the validity of a decision made at a certain point in the design process.

The knowledge acquisition process in developing a CONGEN application begins when the knowledge engineer acquires some basic knowledge from the application domain. The characteristics of the knowledge acquired depend on the domain of the problem, and the level of expertise of the knowledge source. The vast variety of the textbook information sometimes becomes a barrier in the process. The overflow of the knowledge acquired from the textbooks happens because the structural engineering area has spawned many qualitative and quantitative procedures in ensuring the safety and serviceability of a structure [9]. This overflow makes the knowledge acquisition too broad.

One of the problems encountered in this phase was that the knowledge engineer has the same background as the expert - structural engineering. The same background creates two scenarios:

1. The knowledge engineer can absorb and process the acquired expertise thoroughly so that users will understand the application more easily.
2. Both of them become so comfortable with the subject that they do not consider the users' perception of the problem and proceed with more technical and domain-dependent approach for the application.

In the Cabin Design application, the knowledge always changes, because of the evolving nature of the domain knowledge itself. New ways and innovations are created to solve more complex problems. Therefore, it is almost impossible to create a fully complete expert system in the first cut. As the knowledge base is tested and expanded, there will be more modifications to be incorporated in the later versions. The most important thing is that the first version should lay a strong groundwork for the subsequent versions, because changing the application structure is harder to accomplish than adding more knowledge to the knowledge base.

The extent of the usage of Cabin Design application depends on the qualitative and



## 7.1 The Problem

---

quantitative complexity of the structural engineering design processes. Although the expert system approach helps in the selection process, the expert system itself deals only with *shallow* knowledge [9] such as empirical knowledge. This is the reason why the application needs to have a structural analysis program to support deeper knowledge. Some of the uses of this application can be the preprocessor - preparing the preliminary data - or postprocessor - rule-of-thumb testing of the output - of a structural analysis program. The complexity of preparing data for an analysis program will be reduced to conceptual design artifacts satisfying preliminary constraints rather than the extensive process of defining the topology and geometry of the structure to be fed to the analysis program [11].

### 7.1.3 Vocabulary

The vocabulary used in this application follows the definitions of all the basic members of a structural system. Basically, all basic member classes are subclassed from the class “Structural Members”. The “Structural Member” class acts as the superclass of all the physical parts of a structure as shown in figure 7-4. The class definition of “Structural Member” contains detailed information about:

- *The relative coordinate & rotation of a member.* This information provides a direct relationship between the location of the structural member and the system that contains this member. The relative coordinate and rotation of a member are defined according to the Structural System’s coordinate, not the global coordinate i.e. (0,0,0).
- *The belongs-to relationship.* This information will help to trace which Structural System contains this member.
- *The loadings.* This information consists of the main design loads of a member i.e.: dead load, live load, wind load, etc.

The “Structural Member” class is subclassed into three dimensional member classes representing linear, area, and joint member classes. Ultimately the dimensional classes are subclassed into basic structural classes containing more information about the physical part they represent, for example:

## 7.1 The Problem

---

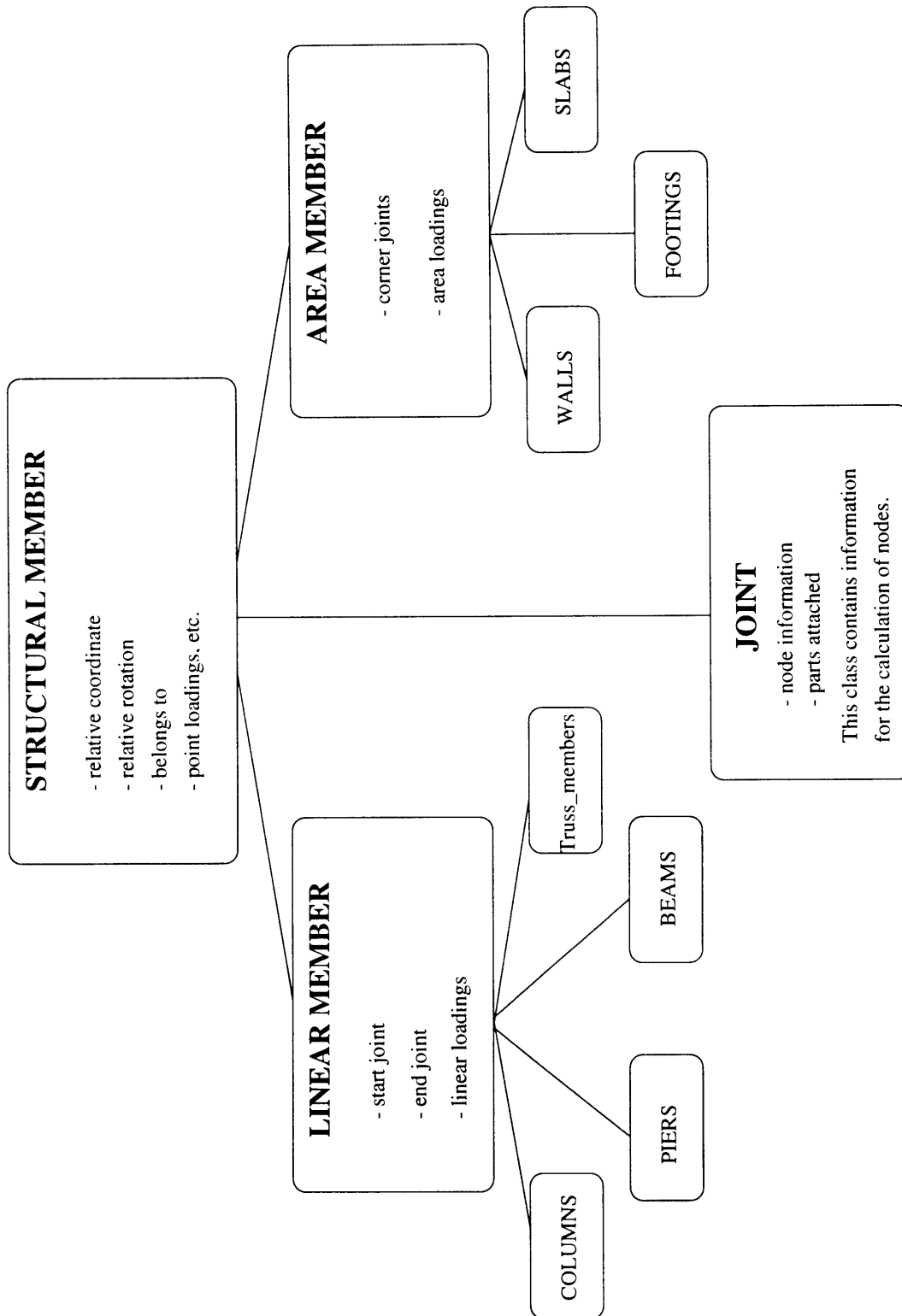


Figure 7-4: The New Structural Member hierarchy

## 7.1 The Problem

---

- **Beam**, representing the concept of a beam.
- **Column**, representing the concept of a column.
- **Slab**, representing the concept of a concrete slab. The “Slab” class is subclassed into two other kinds of Slab - “Waffle\_slab” and “Ribbed\_slab”.
- **Pier**, representing the concept of a pier.
- **Wall**, representing the concept of a wall.
- **Wall\_opening**, representing the concept of an opening within a wall such as doors, windows, etc.
- **Truss\_system**, representing the concept of a truss system of the roof.
- **Truss\_member**, representing the concept of a truss member within a truss system.
- **Mat\_found** representing the concept of mat foundation.
- **Spread\_found** representing the concept of column footing, or isolated foundation.
- **Strip\_footing** representing the concept of wall foundation.

The definition of “Beam” class complete with its attributes to simulate the real life behavior of the member can be seen in Appendix E. In addition, more detailed information about the Structural Member class and its subclasses can also be found in Appendix E.

In addition to the above classes, there are three more classes representing the basic behavior of the cabin itself and its environment. They are: “Cabin”, “Site”, and “Cabin\_part”. The “Cabin” class contains the dimensional constraints of the design, along with additional information about the number of columns, number of beams used, etc. The “Site” class provides information about the condition of the location i.e. geological and topological condition, and the soil bearing capacity. “Cabin\_part” class acts as the container of all the parts created in the process.

“Cabin” and “Cabin\_part” instances have a *part-of* relationship. All other structural member instances have a *part-of* relationship with “Cabin\_part”. Therefore, the designer

## 7.1 The Problem

---

can access the data from the “Cabin” class by traversing the *part-of* hierarchy of the design from the entry point “Cabin\_part”.

### 7.1.4 Geometry

The geometry involved in this application consists of the representation of beams, columns, trusses, walls, and foundations. The basis of the geometry is the Geometric Abstractions module within the GNOMES module. The Geometric Abstractions (GAB) module is a collection of spatial classes that form a layer of abstraction over the actual geometry representation. GAB supports conceptual design, through the use of the different levels of abstraction.

The GAB taxonomy represents objects commonly used in the areas of Civil Engineering, such as beams, plates, shells, etc. GAB uses two different levels of abstraction: internal and external. The external abstraction is between the classes - supported by the concept of inheritance of object oriented programming. On the other hand, the internal abstractions are provided by the information and behavior contained inside each class. Each class has the knowledge to display a representation of the object, based on the information within its attributes [26]. The GAB classes can be accessed directly using the GNOMES Geometric Modeler interface in CONGEN.

There are three important statements regarding GAB geometry:

1. A GAB geometry is created by using the statement:

```
(EXECUTE VAR: $rtn OBJ: $x create_geometry(7,"negeom"))
```

in the COSMOS rule. What this statement means is to inform GNOMES to create a geometry instance of a LINEREC SOLID (Straight Line with 3D extensions) to simulate, for example, the form for beams and columns.

2. By using further GAB geometry statements such as:

```
(EXECUTE VAR: $rtn OBJ: $x set_rotation(0.0,270.0,90.0)) ,
```

the geometry is rotated by 0 degrees around X axis, 270 degrees around Y axis and 90 degrees around Z axis. The rotation example follows the rule of the right thumb,

## 7.1 The Problem

---

with the thumb pointing to the positive side of the rotational axis.

3. We can also utilize the statement:

```
(EXECUTE VAR: $rtn OBJ: $x set_translation(5.0,725.0,10.0))
```

to translate the geometry 5 units along X axis, 725 units along Y axis, and 10 units along Z axis.

In addition to the above, the statement:

```
(MODIFY (OBJ: $x  
(length $len)  
(width $wid)  
(height $hgt)  
) 1000 0.001)
```

modifies the attributes of the geometry, primarily the length, width, and height.

The geometry classes implemented in CONGEN are classified as follows:

- **TYPE: 0 - ENGINEERING OBJECT** is the top level class. It provides the generic interface for all classes. In addition, it also has the information about the bounding box containing all the engineering objects. This class contains the union, difference, and intersection operations.
- **TYPE: 1 - CUBOID**
- **TYPE: 2 - CONE**
- **TYPE: 3 - CYLINDER**
- **TYPE: 4 - STRAIGHT LINE**
- **TYPE: 5 - LINE RECTANGULAR CROSSECTION**
- **TYPE: 6 - LINE CIRCULAR CROSSECTION**
- **TYPE: 7 - LINE RECTANGULAR SOLID**
- **TYPE: 8 - LINE RECTANGULAR HOLLOW**
- **TYPE: 9 - LINE CIRCULAR SOLID**
- **TYPE: 10 - LINE CIRCULAR HOLLOW**

## 7.1 The Problem

---

- **TYPE: 11 - T BEAM**
- **TYPE: 12 - C BEAM**
- **TYPE: 13 - I BEAM**
- **TYPE: 14 - L BEAM**
- **TYPE: 15 - SURFACE**
- **TYPE: 16 - SURFACE RECTANGULAR PLATE**
- **TYPE: 17 - SURFACE CIRCULAR PLATE**
- **TYPE: 18 - SURFACE RECTANGULAR PLATE HOLLOW**
- **TYPE: 19 - SURFACE RECTANGULAR PLATE SOLID**
- **TYPE: 20 - SURFACE CIRCULAR PLATE HOLLOW**
- **TYPE: 21 - SURFACE CIRCULAR PLATE SOLID**
- **TYPE: 22 - ENGINEERING ASSEMBLY**
- **TYPE: 23 - TRUSS**
- **TYPE: 24 - FRAME**

More information about the classes can be found in [26]. The geometry objects are categorized into line and surface forms. Line forms provide a basic abstraction for straight and curved lines, whereas the surface forms can be derived into planar or curved. The abstractions of the classes mentioned above are shown in figure 7-5. The GAB classes with their associated solid and hollow section derivations provide an evolution of shape descriptions from linear elements, such as from lines to beams, from surfaces to slabs.

The geometry classes that are used in this application are mainly type 4 (Straight Line) to represent the concept of beams, columns, and piers, and type 7 (Line Rectangular Solid) to represent the walls, foundation mat, and foundation pads. However, most Civil Engineering structural members can easily be represented using these classes.

Whereas the GAB classes provide the abstraction of the geometry, the rules in COSMOS provide the execution of the methods embedded within the geometry classes itself. For example, the following rule creates the geometry of a northeast column:

## 7.1 The Problem

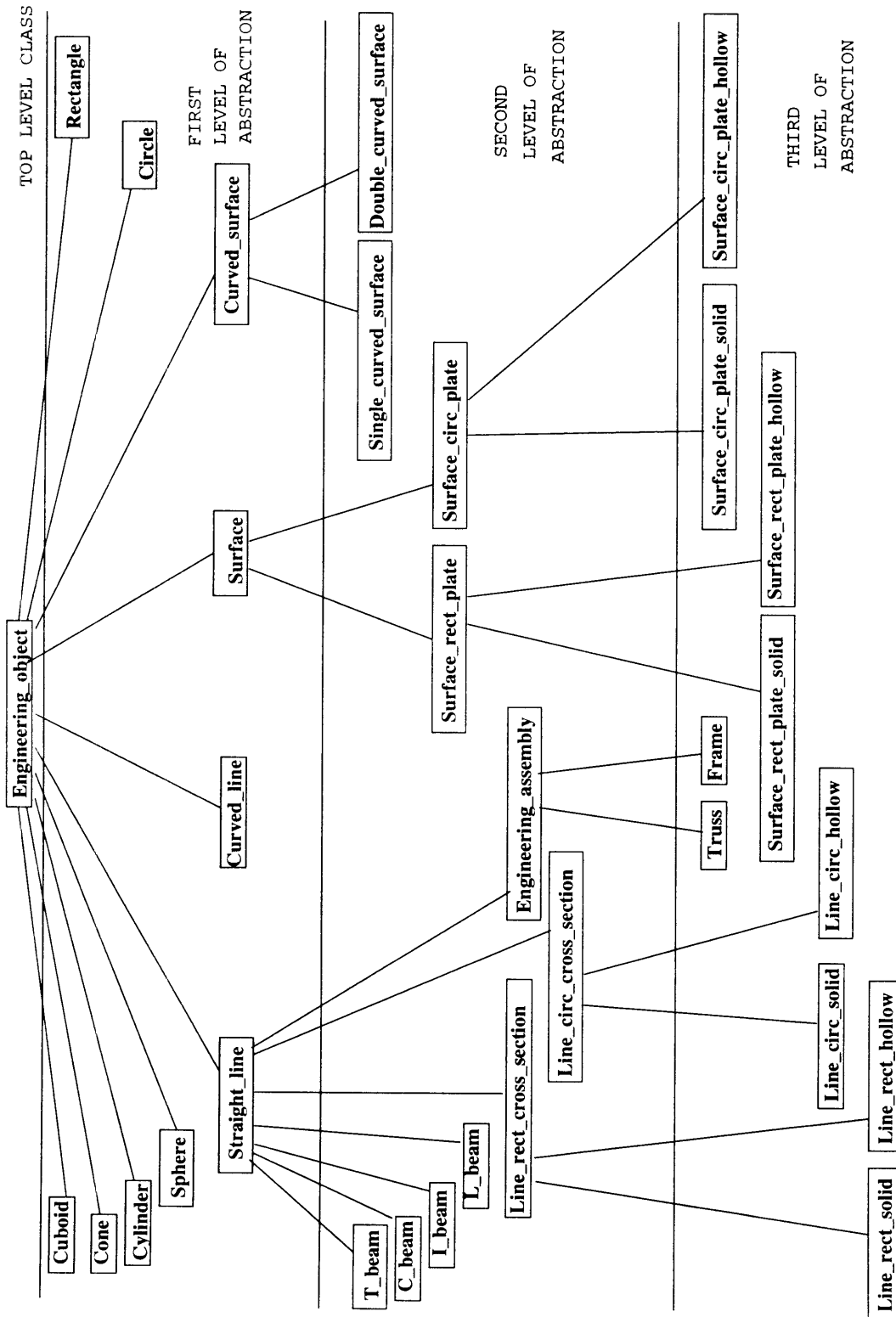


Figure 7-5: The abstraction of GAB geometry classes in CONGEN.

## 7.1 The Problem

---

```
(RULE: ne_geometry 10
IF
((CLASS: Column OBJ: $x
(instancename == "ne_column")) AND
(CLASS: Column OBJ: $x
(((c_length == $len) AND
(c_width == $wid)) AND
((length != $len) AND
(c_depth == $hgt)))
))
THEN (
(EXECUTE VAR: $rtn OBJ: $x create_geometry(7,'negeom'))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(365.0,725.0,10.0))
)
COMMENT:" Rule to set up the geometry of the ne_column")
```

The explanation of each part of the rule is as follows:

- The first condition checks whether there is any instance of the “Column” class with the name of “ne\_column”.
- The second condition passes the values of the instance to \$len, \$wid, and \$hgt. In addition, the condition also checks whether the “length” attribute of “ne\_column” Geometry has been changed to reflect the value from the “Column” class.

The check is useful to avoid endless loop of this rule, because whenever the rule is true, it will always be fired. By adding the condition:

```
(length != $len)
```

the rule will stop firing if the Geometry’s “length” is equal to Column’s “c\_length”.

- The first action to do if both conditions are true is to create this instance’s geometry with the name “negeom”.



## 7.1 The Problem

---

- The second action modifies the attribute values of the “Geometry” to be similar to the “Column” instance values.
- The last two actions provide the extent of the rotation and translation of the “Column” geometry.

You should note that “Geometry” class and “Column” class are not mutually exclusive. “Column” class, or any other CONGEN user-defined classes automatically have “Geometry” class attached to it. This attachment is necessary to interface CONGEN classes with GNOMES functionality. When a CONGEN class object is instantiated, its “Geometry” class is still empty. To activate the instance’s geometry, we use the method *create-geometry(..)* to reserve storage space for “Geometry” attributes of the instance.

For a more complete list of geometry creation rules for this application including the walls, foundations, and trusses, refer to appendix E.

### 7.1.5 Analysis

Ideally, the analysis of the structure produced by this application must be performed by a specialized application, such as GROWLTIGER from MIT. We are in the process of integrating such an application within CONGEN itself by spawning an external process and accessing the external application via dynamic methods.

The current version of CONGEN, provides dynamic methods to be defined within the classes itself. It also enables the passing of variables to a method, and dynamically add method in the runtime. This capability will enable the integration of external methods and other structural analysis application into CONGEN.

Since the first version of this application uses the version of CONGEN without this capability, we decided to use the rule syntax of COSMOS to perform simple calculations for the members in the system such as:

```
(RULE: calculatetotalload 900
IF
(CLASS: Column OBJ: $x
```

## 7.2 The Implementation

---

```
((deadload == $d1) AND
 (liveload == $l1)) AND
 (totalload == 0.0))
)
THEN (
(BIND VAR:$d1 $d1*1.4)
(BIND VAR:$l1 $l1*1.7)
(MODIFY (OBJ: $x
(totalload $d1+$l1)
) 1000 0.001)
)
COMMENT:"Rule to calculate Column totalload")
```

The above example lists the process of calculating the total load passed to a Column by the formula:

$$\text{total load} = 1.4 * \text{dead load} + 1.7 * \text{live load}$$

The *BIND* statement provides the capability of storing values into a temporary variable.

## 7.2 The Implementation

The preliminary information needed to build the cabin design application is explained in the previous section. In this section, we show how to create the complete application.

### 7.2.1 Preparation

Before creating the tutorial application, you must prepare:

1. Have a specialized database volume prepared by your database administrator.
2. Set the `.cshrc` environment variable in the root directory to point to your database volume :

```
setenv EVOLID (the volume number of the database)
setenv CONDIR (the directory where your congen is installed)
setenv CONGEN_USER_AR (the directory where the class archive is installed)
setenv INCDIR (the directory where the congen include source files is placed)
```

## 7.2 The Implementation

---

3. Set the `.sm-config EXODUS` configuration file in the root directory to point to your database volume: `client*mount: (volume number) (port number)@(serverhost)`.
4. Create a special directory to run this tutorial such as: `mkdir /mit/gnomes/congen/tutorial4`.
5. Create a special directory to store the classes generated by this application: `mkdir /mit/gnomes/congen/ar`. Note that this directory must be named `ar` - archive. The `CONGEN_USER_AR` environment variable must point to this directory.
6. Change to the special directory that you have specified (`/mit/gnomes/congen/tutorial4`), and run `CONGEN` within that directory. This directory will store the rulefiles needed for this **CABIN DESIGN** application.

All of these actions can be done with the help of your database administrator and system manager. For more information about the environment variables, please refer to Appendix C. After you have completed these tasks, you are ready to build the application.

**NOTE:** The following information about the `CONGEN` application can be emulated using a C script. All the goals, plans, and classes can be entered and compiled using the script instead of entering them manually. This feature will help shorten the time needed to reenter all the information into the database if the database is corrupted. For more information about the script, please refer to Appendix A. The complete listing of the Cabin Design script can be seen in Appendix E.

### 7.2.2 Creating a new application

Create the application by:

- Select **FILE** → **NEW** in the MAIN CONSOLE window.
- Enter *Cabin* and press **OK**.
- Save the application via **FILE** → **SAVE** in the MAIN CONSOLE window.

## 7.2 The Implementation

---

### 7.2.3 Creating classes and rulesets

#### Creating Classes

For this complete application, you need the following classes:

- *Cabin*. This class contains the crucial information of the cabin to be designed, such as the dimensions of the cabin given by the architects, the number of beams, columns, and girders, the type of foundation, etc.
- *Site*. This class contains the information about the site of the cabin including the soil condition, and the location of the site. Some of the information in this class will be used to determine the type of cabin's foundation.
- *Cabin-part*. This class acts as the container for the parts being created in the design process. The "Cabin-part" class was more of a convenience to distinguish the parts and the cabin itself.

If the parts are attached via the *make\_part* method, the SYNTHESIZER window will show explicitly the link between the parts and the Cabin instance. By using "Cabin-part" class to contain the link, the SYNTHESIZER window will only show the link between the instances of "Cabin" and "Cabin-part".

This approach will help decrease the number of objects shown in the SYNTHESIZER window. For clearer view of the difference, refer to figure 7-6 and figure 7-7.

- *Column*. This class acts as the representation of a column with its attributes.
- *Beam*. The representation of a beam with its attributes.
- *Slab*. The representation of a slab with its attributes.
- *Wall*. The representation of a wall with its attributes.
- *Wall-opening*. The representation of a wall opening with its attributes.
- *Truss-member*. The representation of a truss member with its attributes.

## 7.2 The Implementation

---

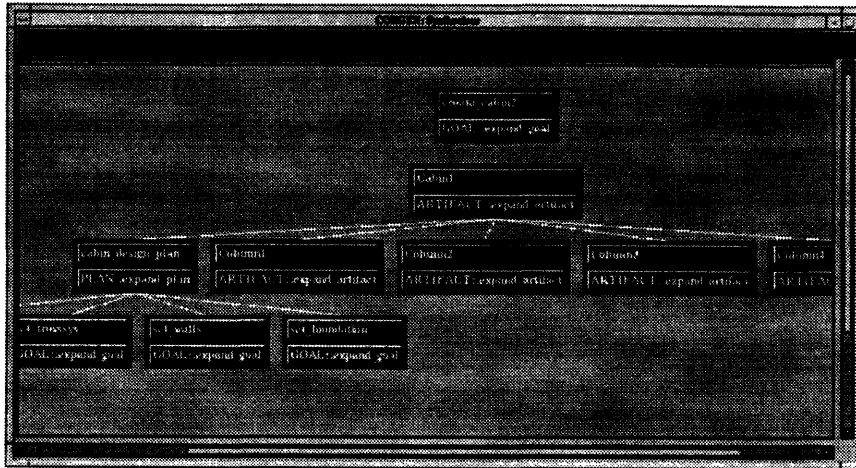


Figure 7-6: Cabin class expanded without using the Cabin-part - notice all the parts linked to Cabin class filling the window.

- *Truss\_system*. The representation of a truss system with its attributes.
- *Mat\_found*. The representation of a mat foundation with its attributes.
- *Spread\_found*. The representation of a spread foundation with its attributes.
- *Strip-footing*. The representation of a strip footing / wall foundation with its attributes.

As an example, to create one of the classes, you should do the following:

### 1. Creating “Cabin” class.

Select **KNOWLEDGE** → **PRODUCT DEFINITIONS**, in the **PRODUCT KNOWLEDGE** window, select **FILE** → **NEW CLASS**. Enter *Cabin* and press **OK**.

In the **CLASS EDITOR** window, select **EDIT** → **PUBLIC**, then **FILE** → **NEW** in the next window.

Enter:

```
float ca_length;  
float ca_height;
```

## 7.2 The Implementation

---

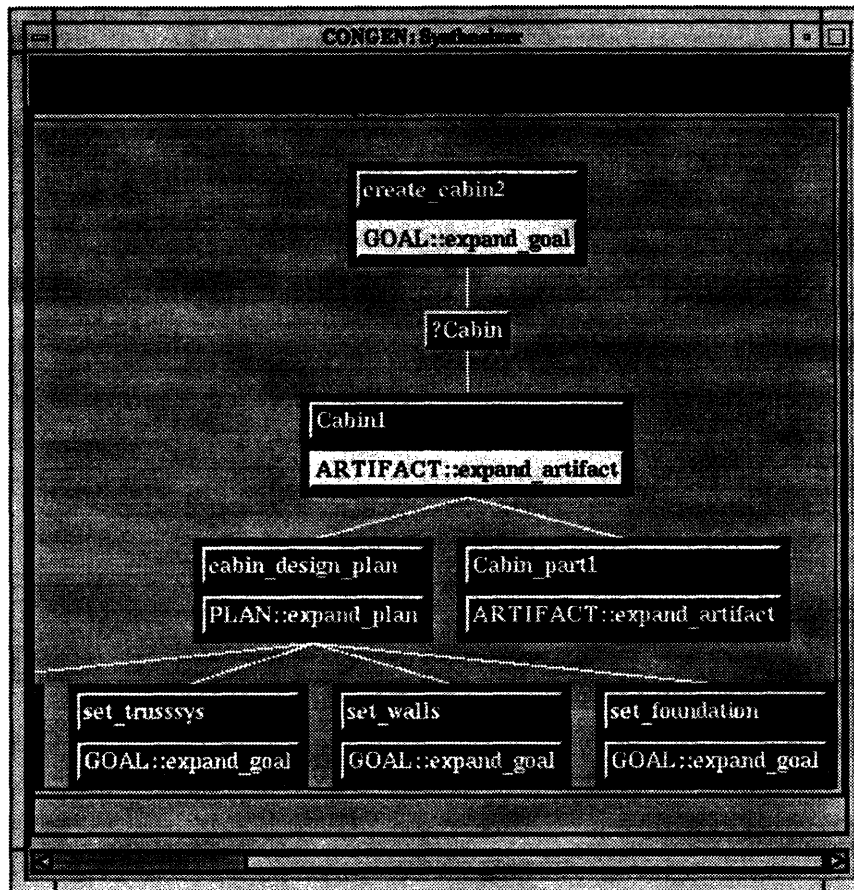


Figure 7-7: Cabin class expanded using the Cabin\_part. Cabin\_part contains all the instances shown in the preceding figure.

```
float ca_width;
```

```
...
```

The remaining attributes to be entered can be examined from figure 7-8. For more complete list of attributes and the header files of all the classes, refer to appendix E.


Press **SAVE**, **QUIT**, and **OK** in the WARNING window.

Exit the editor by selecting **FILE** → **QUIT** in the CLASS EDITOR window, and **FILE** → **QUIT** to go back to PRODUCT KNOWLEDGE window.

**NOTE:** In compiling all the classes for this application, it is better to add three classes at a time and compile, instead of adding all classes and try to compile ev-

## 7.2 The Implementation

---



```
#ifndef Cabin_H
#define Cabin_H

#include "artifact.h"
#include <E/dbStrings.h>
#include "dbstr.h"
dbclass Cabin;public Artifact{
private:
public:
static int art_count ;
static int create_instance(char* a,char* iname,int pers);
static Artifact* copy_instance(Artifact* i);
dbchar #soil_condition;
dbchar #site_location;
dbfloat ca_length;
dbfloat ca_width;
dbfloat ca_height;
dbfloat ca_roof_height;
dbfloat ca_roof_span;
dbint numcolumns;
dbint numbeams;
dbint numpurlins;
dbint numtruss;
dbint numgirders;
dbchar #trusstype;
dbchar #foundationtype;
dbchar #southwalltype;
dbchar #northwalltype;
dbchar #eastwalltype;
dbchar #westwalltype;
dbchar #w_wall_has_opening;
dbchar #e_wall_has_opening;
dbchar #s_wall_has_opening;
dbchar #n_wall_has_opening;

Cabin(char*, char*);
virtual Val direct_get_value (dbchar*);
virtual int direct_put_value (dbchar*,char*);
};
#endif

-----Emacs: Cabin.h (C++)-----HLL-----
Loading ~/c++-mode.el...done
```

Figure 7-8: Cabin class attributes.

everything at once. There is a bug with the current version of EXODUS that causes CONGEN to hang whenever there are too many classes to be compiled and committed.

### Creating rulesets

For your convenience, we have included the listing of all the rulefiles in Appendix E. They are:

## 7.2 The Implementation

---

- *cabin-eff.rul.* This rulefile instantiates the Cabin artifact and sets the initial information about the Cabin itself such as the dimensions needed for the other parts.
- *set-columns-eff.rul.* This rulefile instantiates the Column artifacts with the geometry information.
- *set-girders.rul.* This rulefile controls the selection of the number of girders used in the structure.
- *set-girders-eff.rul.* This rulefile instantiates the girders with the geometry information.
- *set-supporting-beams.rul.* This rulefile controls the selection of the number of supporting beams used in the structure.
- *set-supporting-beams-eff.rul.* This rulefile instantiates the supporting beams with the geometry information.
- *set-trusses.rul.* This rulefile controls the selection of the number of truss system used in the structure.
- *set-trusses-eff.rul.* This rulefile instantiates the truss systems and their members with each geometry information.
- *set-purlins.rul.* This rulefile controls the selection of the number of purlins used in the roof of the structure.
- *set-purlins-eff.rul.* This rulefile instantiates the purlins with the geometry information.
- *set-north-wall.rul.* This rulefile controls the selection of the type of the north wall used in the structure.
- *set-north-wall-eff.rul.* This rulefile instantiates the north wall, openings if necessary, with the geometry information.



## 7.2 The Implementation

---

- *set\_south\_wall.rul.* This rulefile controls the selection of the type of the south wall used in the structure.
- *set\_south\_wall\_eff.rul.* This rulefile instantiates the south wall, openings if necessary, with the geometry information.
- *set\_east\_wall.rul.* This rulefile controls the selection of the type of the east wall used in the structure.
- *set\_east\_wall\_eff.rul.* This rulefile instantiates the east wall, openings if necessary, with the geometry information.
- *set\_west\_wall.rul.* This rulefile controls the selection of the type of the west wall used in the structure.
- *set\_west\_wall\_eff.rul.* This rulefile instantiates the west wall, openings if necessary, with the geometry information.
- *set\_foundation.rul.* This rulefile controls the selection of the type of foundation used in the structure depending on the site, the number of columns and openings on the walls.
- *set\_foundation\_eff.rul.* This rulefile instantiates the parts of the foundation with the geometry information.

If we observe the list of the rulefiles closely, it is clear that there are two kinds of rulefiles: the selection rulefile and the effects rulefile. The selection rulefile acts as the controller of the decision making process. The choices that the selection rulefiles provide will be based on the active context of the knowledge base. On the other hand, the effects rulefile takes the decision and fires some other rules based on what will happen to the knowledge base if one decision has been made.

An example of a selection rulefile is as follows:

## 7.2 The Implementation

---

```
(RULE: three 20
IF
(CLASS: Declist OBJ: $y
((set_columns == "6") AND
 (set_trusses != "3"))
)
THEN (
(MODIFY (OBJ: $y
(set_trusses "3")
) 1000 0.001)
)
COMMENT:" If the number of columns is six, then we can use three trusses")

(RULE: two 20
IF
(CLASS: Declist OBJ: $y
((set_columns != "6") AND
 (set_trusses != "2"))
)
THEN (
(MODIFY (OBJ: $y
(set_trusses "2")
) 1000 0.001)
)
COMMENT:" If the number of columns is not six, then we can use only 2 trusses")

(RULE: three_or_two 20
IF
(CLASS: Declist OBJ: $y
((set_columns == "6") AND
 (set_trusses != "2"))
)
THEN (
(MODIFY (OBJ: $y
(set_trusses "2")
) 1000 0.001)
)
COMMENT:" If the number of columns is six, then we can use two trusses")
```

Basically, these selection rules dictate that if we have more than four columns in the structure, we can have either two or three trusses. If we only have four columns, then the structure can only support two trusses. This rulefile is a very simple example of how the

## 7.2 The Implementation

---

“rules of thumb” in designing a structure can be applied to the design application. These rules are attached to the GOAL: *set\_trusses* shown in figure 7-16.

An example of an effect rulefile is as follows:

```
(RULE: createtwopurlins 1000
IF
((CLASS: Cabin OBJ: $x
((ca_length == $len) AND
(num_purlins == 2))
) AND
(CLASS: Cabin_part OBJ: $y
(instancename == "cabin_parts")
))
THEN (
(MAKE (CLASS: Beam OBJ: e_purlin
(b_length $len)
(b_width 5.0)
(b_depth 5.0)
))
(MAKE (CLASS: Beam OBJ: w_purlin
(b_length $len)
(b_width 5.0)
(b_depth 5.0)
))
(EXECUTE VAR:$rtn OBJ: $y make_part('eastpurlin','Beam','e_purlin'))
(EXECUTE VAR:$rtn OBJ: $y make_part('westpurlin','Beam','w_purlin'))
)
COMMENT:"Rule to create the two purlins")

(RULE: e_purlin_geometry 10
IF
((CLASS: Beam OBJ: $x
(instancename == "e_purlin")) AND
(CLASS: Beam OBJ: $x
(((b_length == $len) AND
(b_width == $wid)) AND
((length != $len) AND
(b_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(4,'epurlgeom'))
(MODIFY (OBJ: $x
```

## 7.2 The Implementation

---

```
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,0.0,90.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(275.0,5.0,300.0))
)
COMMENT:" Rule to set up the geometry of the e_purlin")
...
```

The first rule above states that if the decision is to build two purlins, then create the two-purlin artifacts, and attach the purlins to the “Cabin\_part” instance (referred by **OBJ: \$y**). After the two purlins have been created, the geometry of one of the purlins is created using another rule. The above rules are attached to the *GOAL: set\_purlins* shown in figure 7-17.

### 7.2.4 Making the Application

After defining all the classes and rulesets, it is time to compile the classes to make it ready for CONGEN. When the compilation is finished, please remember to save the application permanently to the database. If by any chance CONGEN hangs, you should press Ctrl-C to halt the operation and start typing everything again.

**NOTE:** In this version of CONGEN, you are allowed to create a script to shortcut the manual entries of the goals, plans and classes. This script is basically an external method to invoke internal CONGEN functions to define the application directly without the user interface. Please refer to Appendix E for further information about the script.

### 7.2.5 Creating goals and plans for the application

#### Goals

When the Product knowledge (i.e. Classes, Rulefiles, etc) input process is finished, the next step is to enter the Process knowledge consisting of the sequence of goals, plans, and artifacts to be created. Basically, all the goals to be entered are as follows:

## 7.2 The Implementation

---

1. *create\_cabin2*. This goal is the root goal of the application. It creates the Cabin artifact which has the initial specifications that will affect other parts.
2. *set\_beam\_column\_grid*. This goal is to expand the process of setting the beam-column grid.
3. *set\_columns*. This goal is to select the number of the principal columns in the structure. Its choices are: 0, 4, 6 columns.
4. *set\_beams*. This goal is to expand the process of setting the beams in the structure.
5. *set\_girders*. This goal is to select the number of girders for the structure. Its choices are: 0, 2 girders.
6. *set\_supporting\_beams*. This goal is to select the number of supporting beams aside of the girders. Its choices are: 0, 1, 2, 3, 6, 7.
7. *set\_trussys*. This goal is to expand the process of setting the roof trusses of the structure.
8. *set\_trusses*. This goal is to set the number of trusses available for the structure. Its choices are: 2, 3 trusses.
9. *set\_purlins*. This goal is to set the number of purlins created for the roof. Its choices are: 0, 2, 4 purlins.
10. *set\_walls*. This goal is to expand the process of setting the wall system of the structure.
11. *set\_north\_wall*. This goal is to select the type of the north wall. Its choices are: Brick, Opening, Shear.
12. *set\_south\_wall*. This goal is to select the type of the south wall. Its choices are: Brick, Opening, Shear.
13. *set\_east\_wall*. This goal is to select the type of the east wall. Its choices are: Brick, Opening, Shear.

## 7.2 The Implementation

---

14. *set-west-wall*. This goal is to select the type of the west wall. Its choices are: Brick, Opening, Shear.
15. *set-foundation*. This goal is to select the type of the foundation of the structure. Its choices are: Mat, Spread, Strip.

The detailed information about the goals follows:

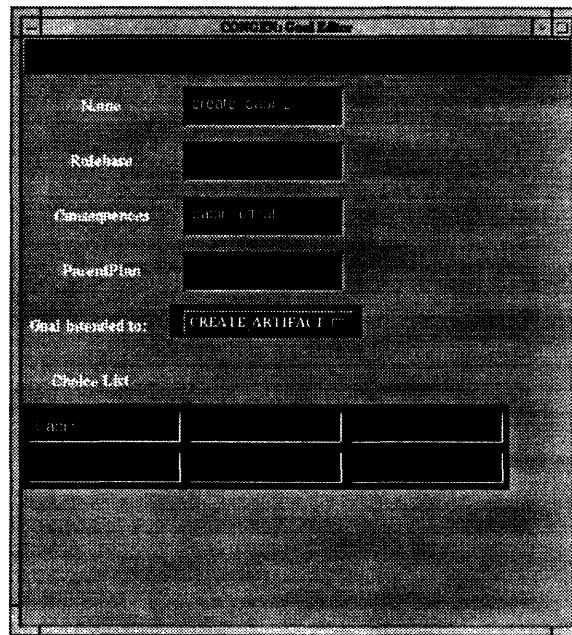


Figure 7-9: The create\_cabin goal description.

## 7.2 The Implementation

---

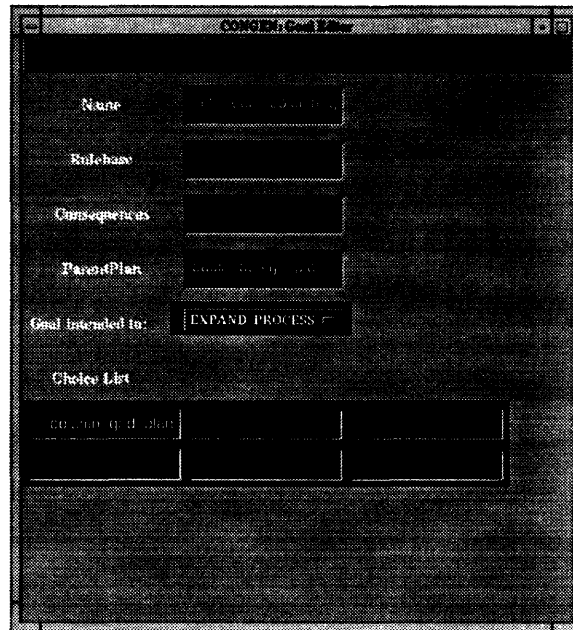


Figure 7-10: The set\_beam\_column\_grid goal description.

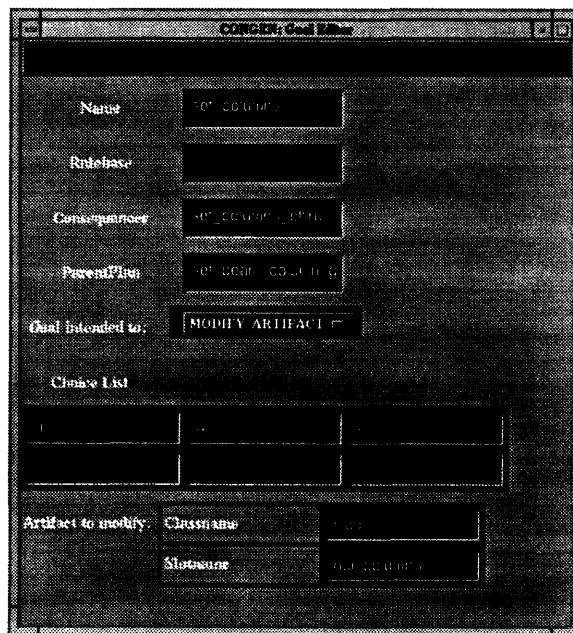


Figure 7-11: The set\_column goal description.

## 7.2 The Implementation

---

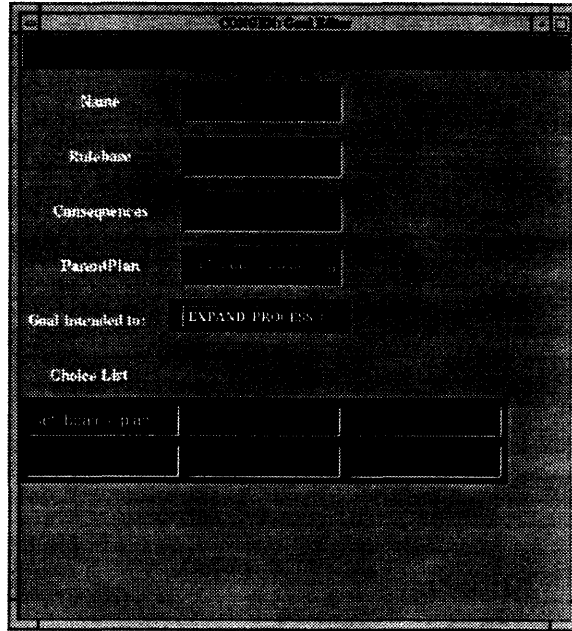


Figure 7-12: The set\_beams goal description.

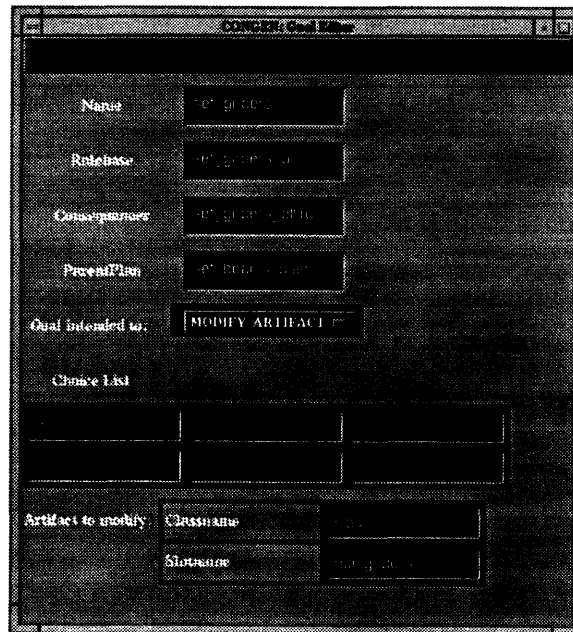


Figure 7-13: The set\_girders goal description.



## 7.2 The Implementation

---

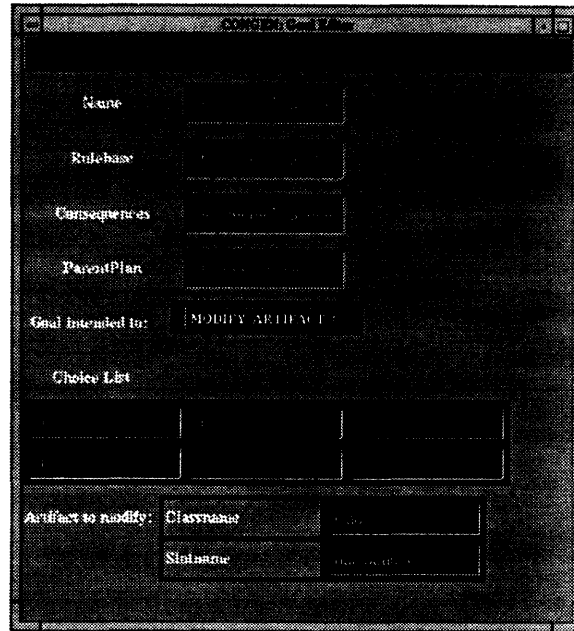


Figure 7-14: The set\_supporting\_beams goal description.

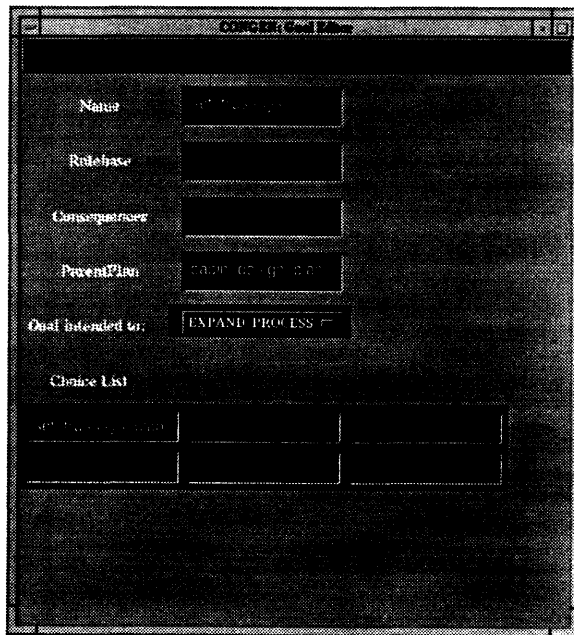


Figure 7-15: The set\_trussys goal description.

## 7.2 The Implementation

---

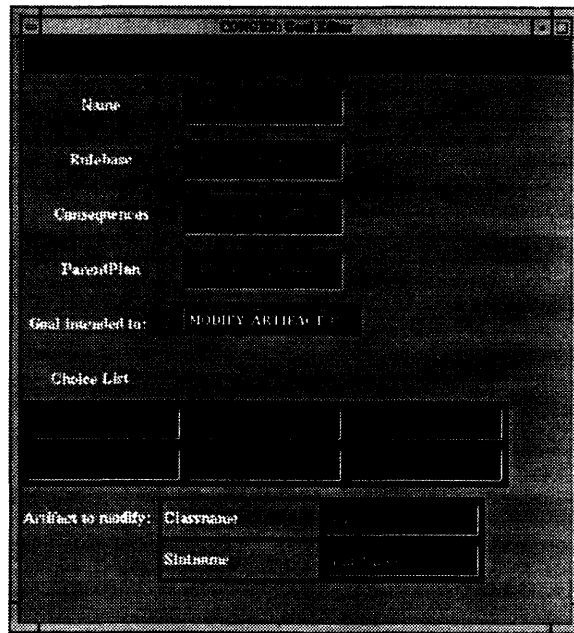


Figure 7-16: The set-trusses goal description.

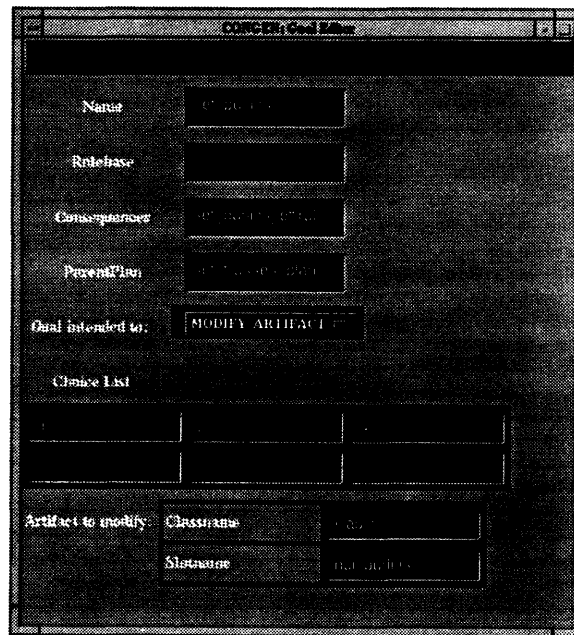


Figure 7-17: The set-purlins goal description.

## 7.2 The Implementation

---

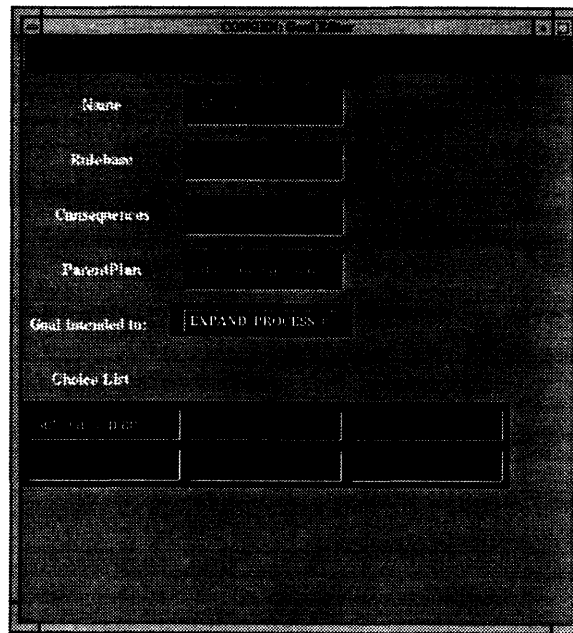


Figure 7-18: The set\_walls goal description.

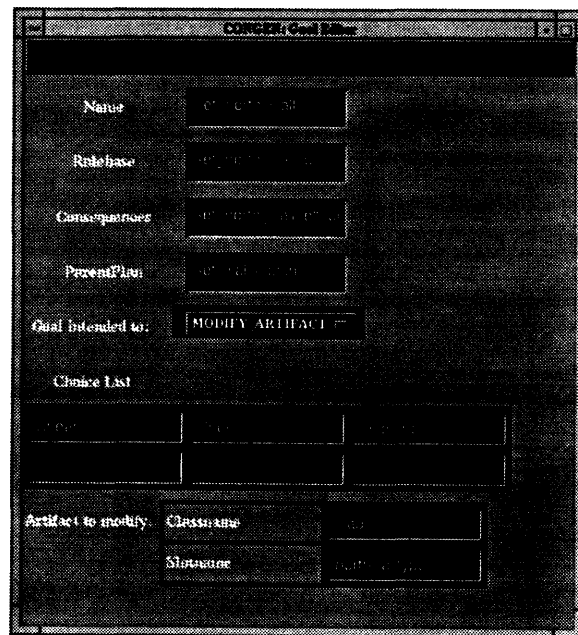


Figure 7-19: The set\_north\_wall goal description.

## 7.2 The Implementation

---

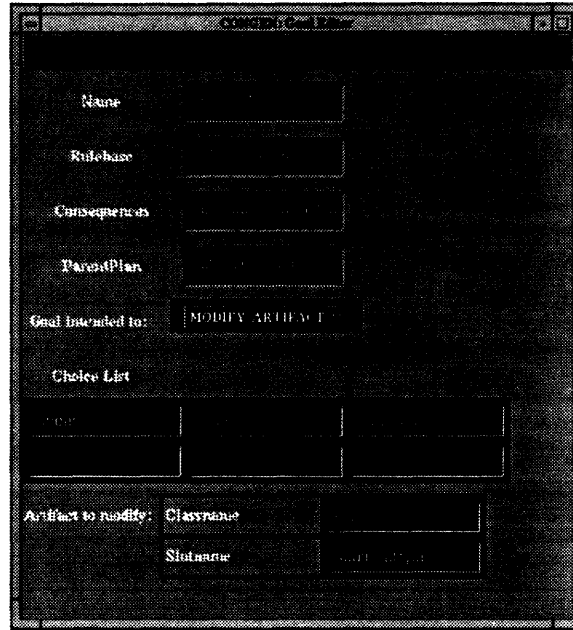


Figure 7-20: The set\_south\_wall goal description.

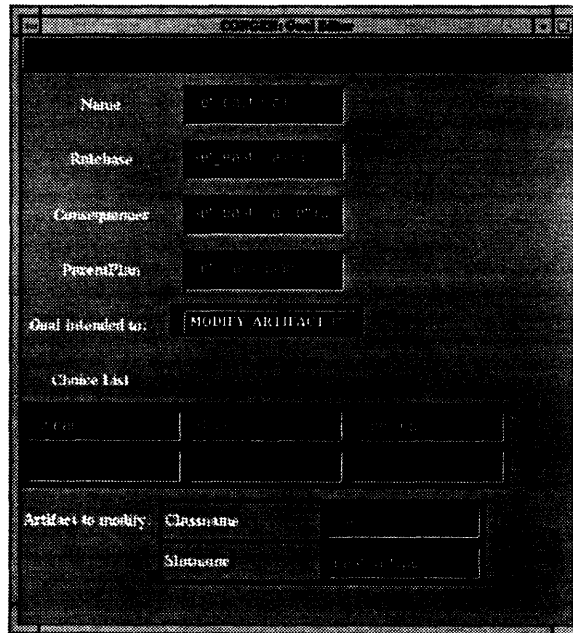


Figure 7-21: The set\_east\_wall goal description.

## 7.2 The Implementation

---

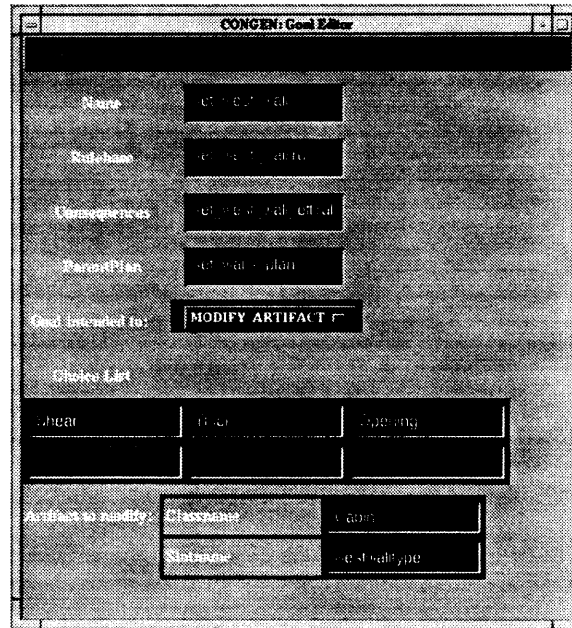


Figure 7-22: The set\_west\_wall goal description.

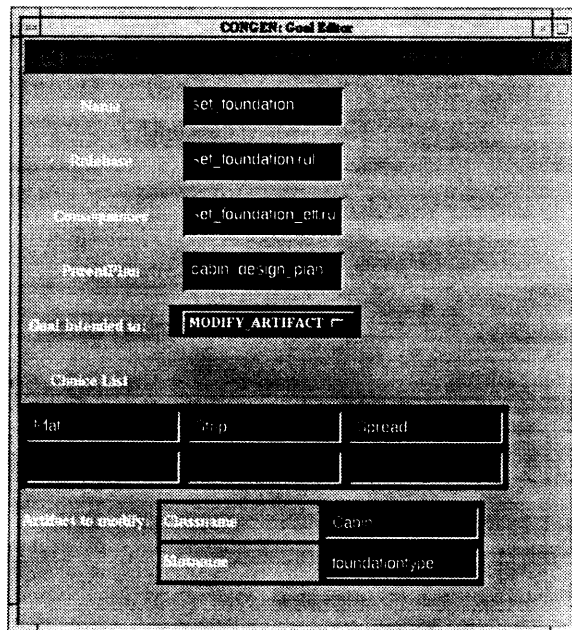


Figure 7-23: The set\_foundation goal description.

## 7.2 The Implementation

---

### Plans

As soon as the goals have been entered, the plans must be entered as well:

1. *cabin\_design\_plan*. This is the plan that controls the sequence of designing parts of the cabin.

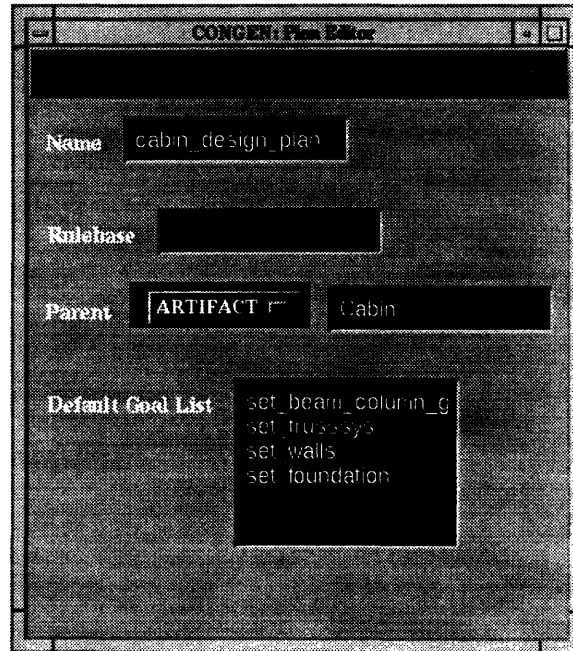


Figure 7-24: The cabin\_design\_plan description.

2. *set\_beam\_column\_grid\_plan*. This is the plan that controls the sequence of setting the beam-column grid of the structure.
3. *set\_beams\_plan*. This is the plan that controls the beam system in the cabin.
4. *set\_walls\_plan*. This is the plan that manages the type of each direction of the walls in the cabin.
5. *set\_trussys\_plan*. This is the plan that provides the sequence of the truss system design of the cabin.

## 7.2 The Implementation

---

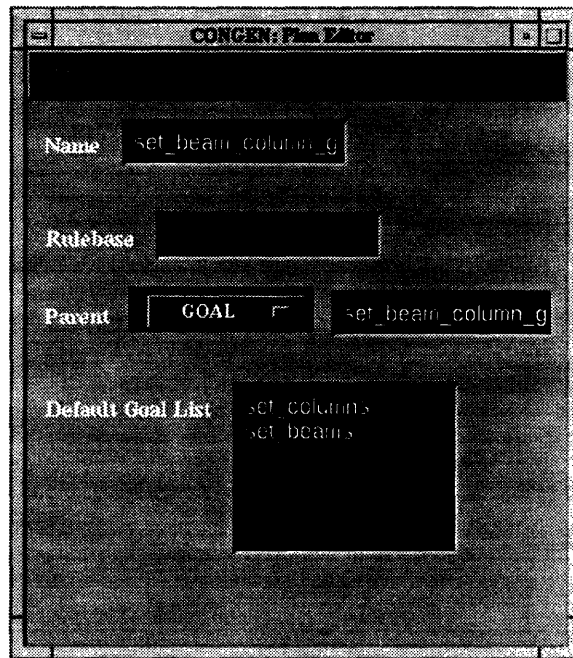


Figure 7-25: The set\_beam\_column\_grid\_plan description.

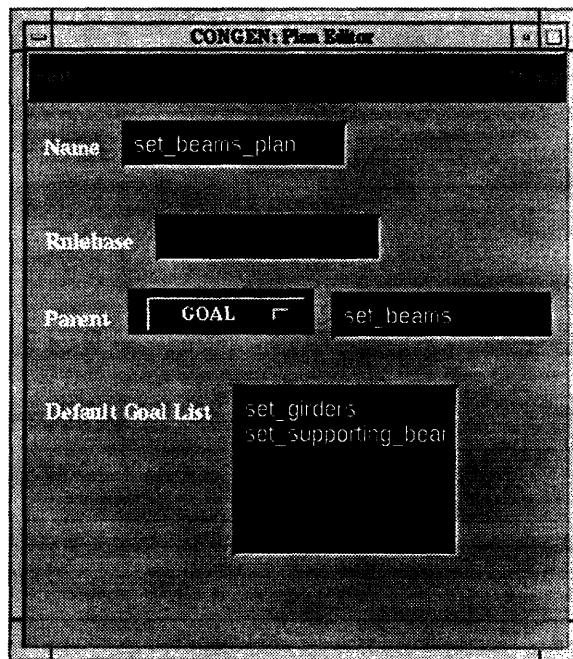


Figure 7-26: The set\_beams\_plan description.

## 7.2 The Implementation

---

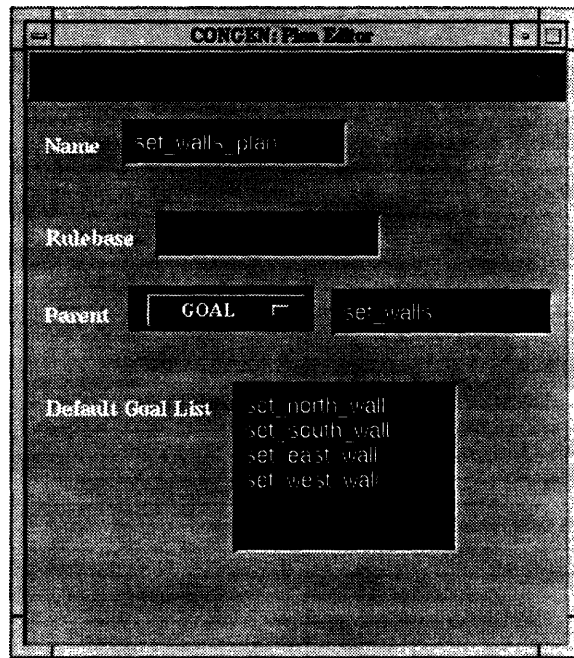


Figure 7-27: The set\_walls\_plan description.

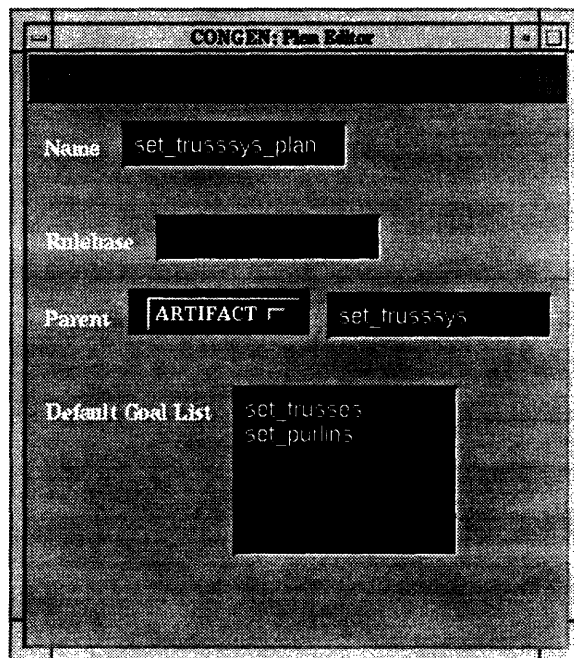


Figure 7-28: The set\_trussys\_plan description.



## 7.2 The Implementation

---

### 7.2.6 Executing the Synthesizer

When all the knowledge elements are entered and ready to be executed, the flow of the application should be checked using the Synthesizer. At this phase, you should have saved all the information to the database. In the previous chapter we showed how to specify the initial values for an artifact, either by using rulefiles or through the Specification Editor. In this case, we are using the rulefile to control the initial specifications of the Cabin. The advantage of using rulefile instead of SPECIFICATIONS window is that the rulefile is independent of a particular program instantiation. Therefore, if the application is erased accidentally, the rulefile will still contain the initial specification values.

To run the application, execute the Synthesizer from the MAIN CONSOLE window. Press the button *GOAL:: EXPAND-GOAL* in the *create-cabin2* and then click on the *?CABIN* button. After you have done this, you have created an instance of the Cabin with its initial specifications.

After instantiating the Cabin artifact, you should expand the process furthermore by expanding the artifact which will reveal the *cabin-design-plan* that consists of the sequence of design from left to right: *set-beam-column-grid* goal, *set-trussys* goal, *set-walls* goal, and *set-foundation* goal. Refer to figure 7-29 for the example of the Synthesizer execution.

Pursuing a design alternative such as the combination of the beam-column grid will yield something like figure 7-30.

You may continue executing Synthesizer by clicking at the choices presented by the goals, or by clicking at goals and plans to expand the processes. You can always access the editor of the corresponding Synthesizer object by clicking on the upper button, for example: if you click on the *Cabin1* button, the ARTIFACT EDITOR window will pop out, as seen in figure 7-31.

Following this example and traversing every path available for the application will create a complete design of the cabin based on your configuration of choices. Additionally, you can check whether the right rulefiles have been fired from the XTERM window.

### Context Switching within the application

## 7.2 The Implementation

---

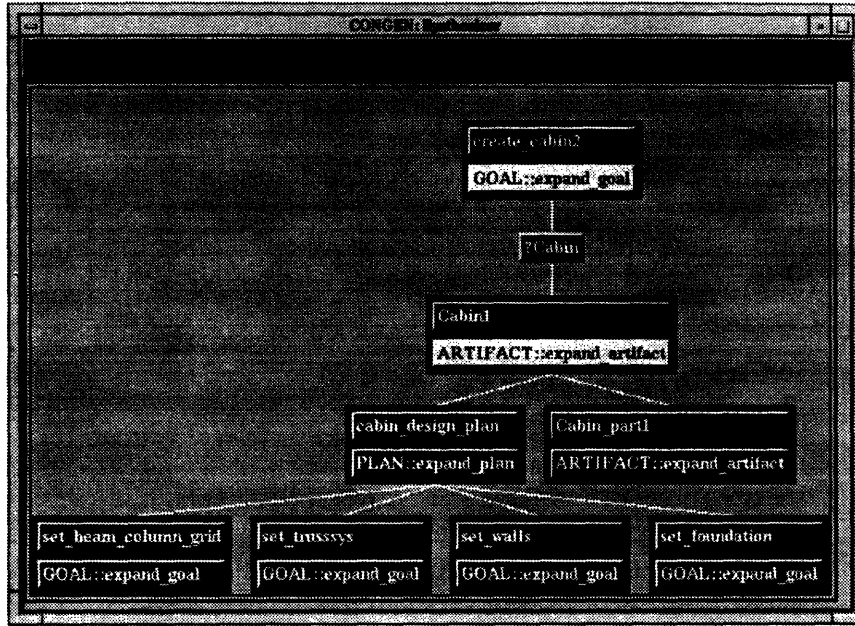


Figure 7-29: The Synthesizer first pass of the Cabin Design application.

CONGEN incorporates a powerful tool within its knowledge-base: the Context switching facility. As explained, Contexts contain the complete information about the state of the knowledge-base whenever a decision has been made such as shown in figure 7-32.

When you summon the DDH EDITOR window, you are presented with four subwindows inside the editor. The lower right window shows the list of Contexts created so far. This figure shows that there are a lot of Contexts created for the course of the design process, such as: *set\_columns.4* which refers to the state of the knowledge-base when the choice of creating four columns was made. By clicking on the *set\_columns.4* in this subwindow, the CONTEXT BROWSER window pops up showing the paths that have been traversed and choices made such as shown in figure 7-33.

By switching Contexts, you can pursue other design alternatives by starting the Synthesizer from that particular Context. The steps are as follows:

1. Quit the Synthesizer.
2. Summon the DDH EDITOR window (**KNOWLEDGE** → **PROCESS KNOWLEDGE** menu in the MAIN CONSOLE window).

## 7.2 The Implementation

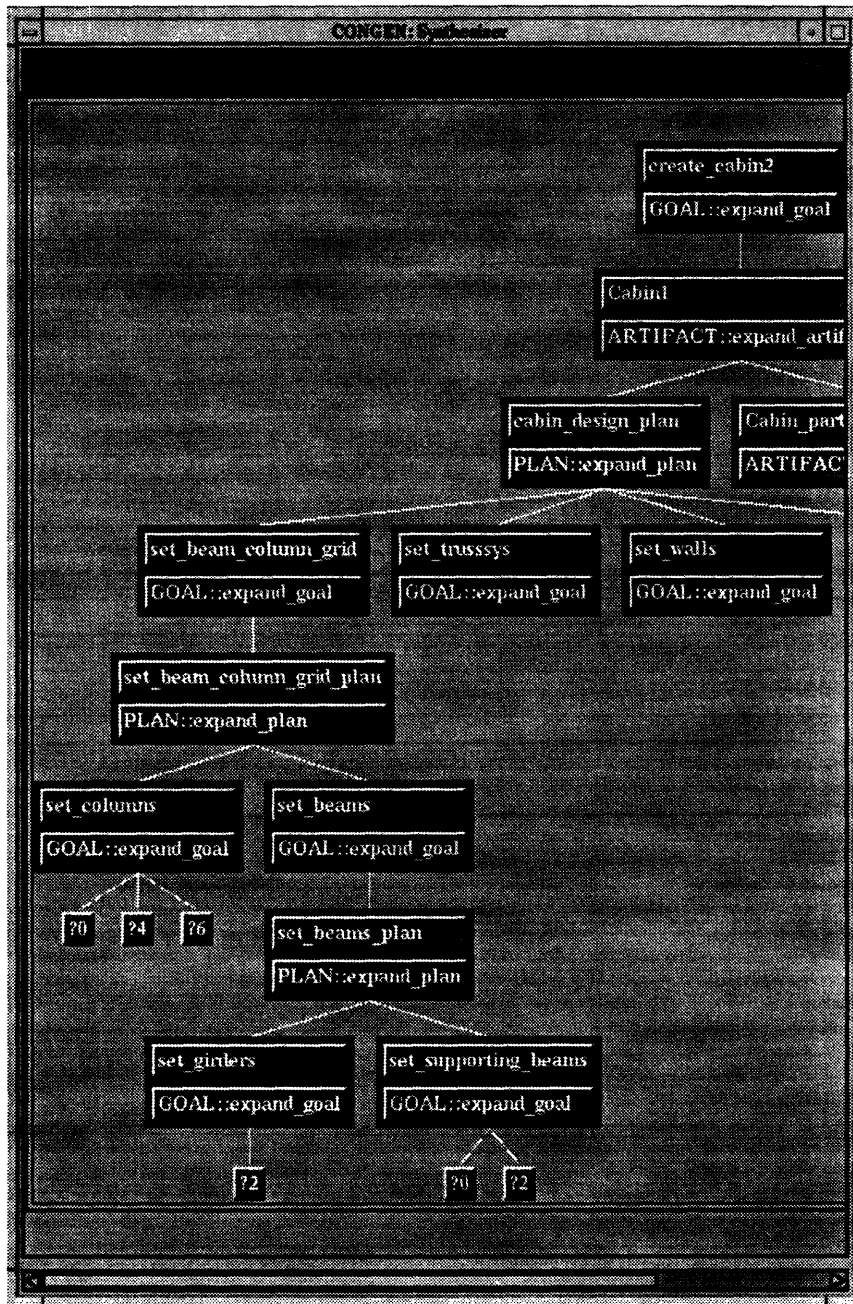


Figure 7-30: The Synthesizer beam-column grid pass of the Cabin Design application.

## 7.2 The Implementation

---

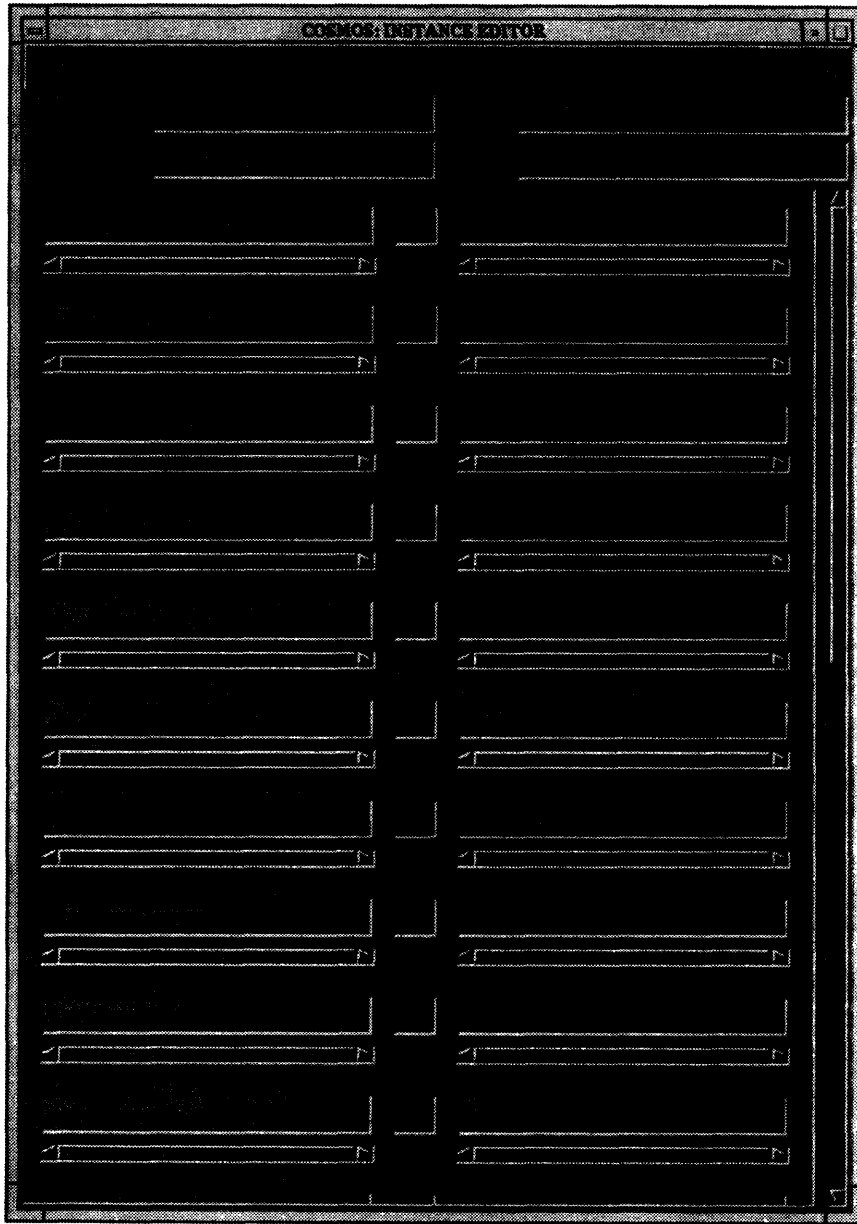


Figure 7-31: The Synthesizer pops out the editor for the artifact Cabin of the Cabin Design application.

## 7.2 The Implementation

---

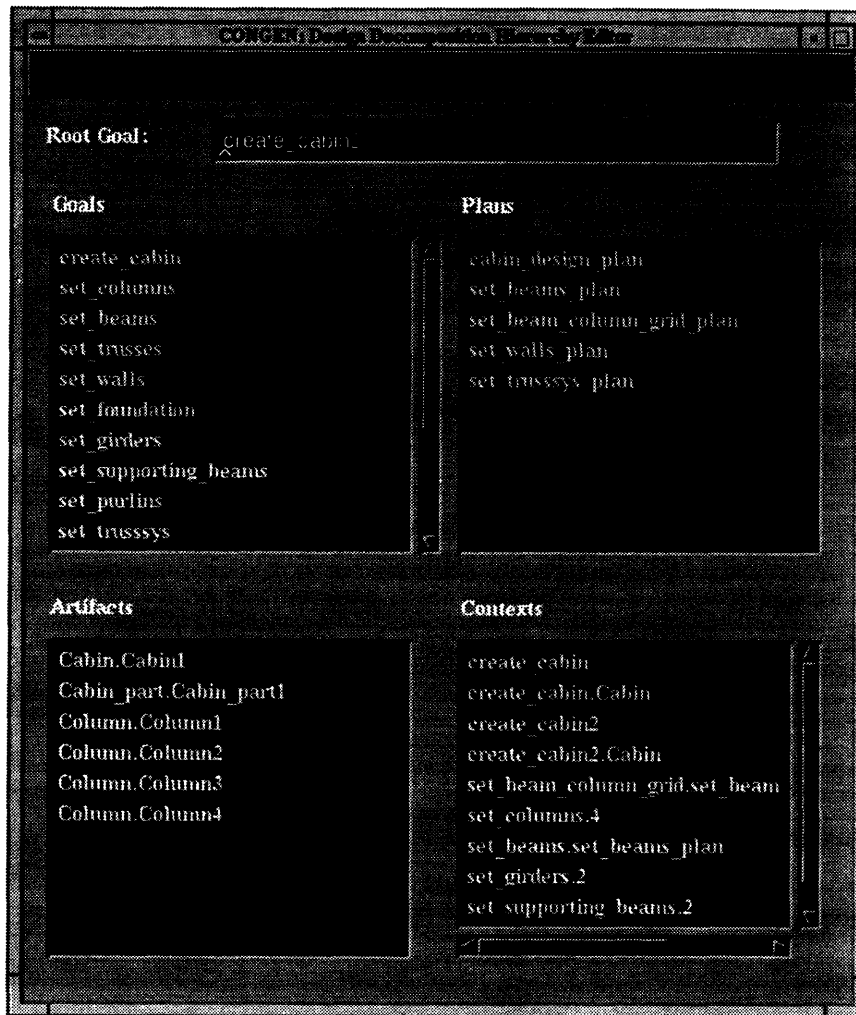


Figure 7-32: The DDH EDITOR window.

3. Click on the Context that you want to start your design alternative from.
4. Go back to the MAIN CONSOLE window and execute the Synthesizer again.

When you execute the Synthesizer after you select a new Context, you are given a chance to pursue a different path from the one you have traversed before. For example, after you pursue a certain path like the one shown in figure 7-34, you may want to change the decision of using different number of columns. To do this, you should perform the steps outlined above and choose the Context of *set-beam-column-grid.set-beam-column-grid-plan* in order to reset the choice of *set-columns.4* Context. The state of knowledge-base in the

## 7.2 The Implementation

---

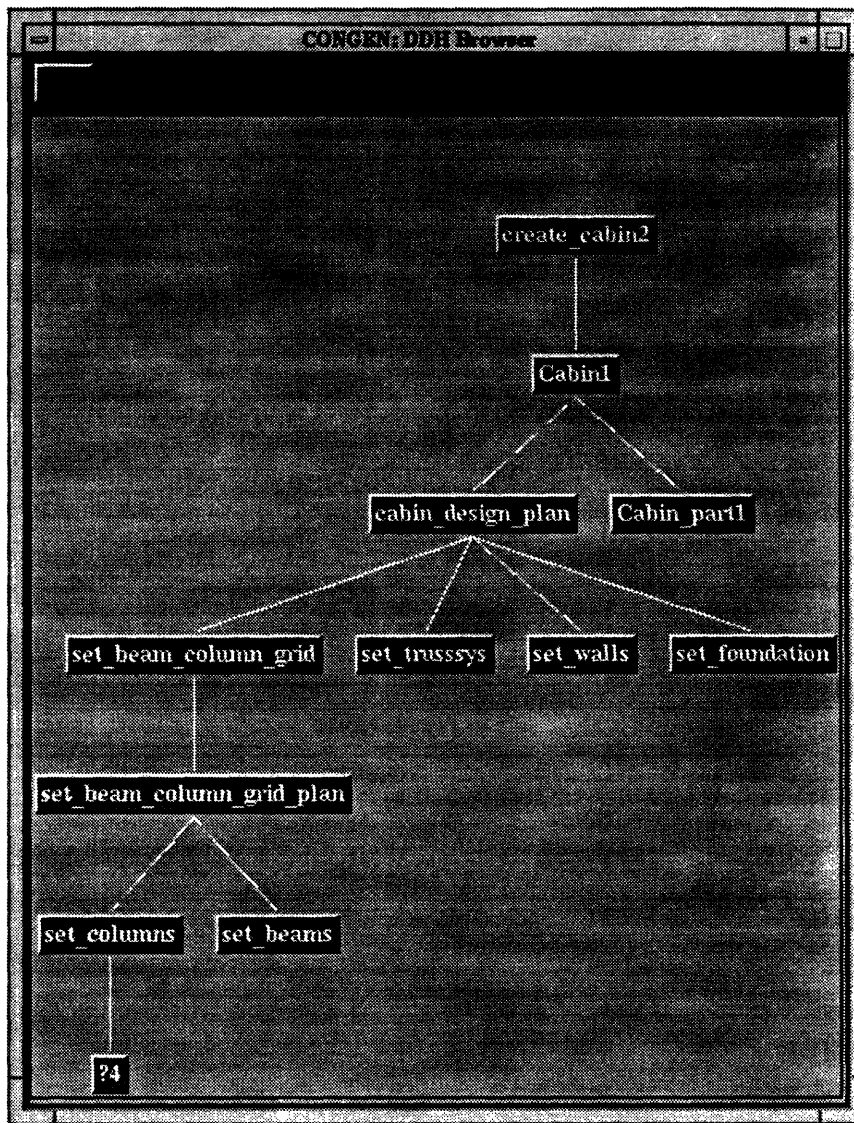


Figure 7-33: The CONTEXT BROWSER window for the Context: set.columns.4.

new Context is shown in figure 7-35.

After you have selected the new Context, then you are ready to select the new configuration by executing the Synthesizer once again.

## 7.2 The Implementation

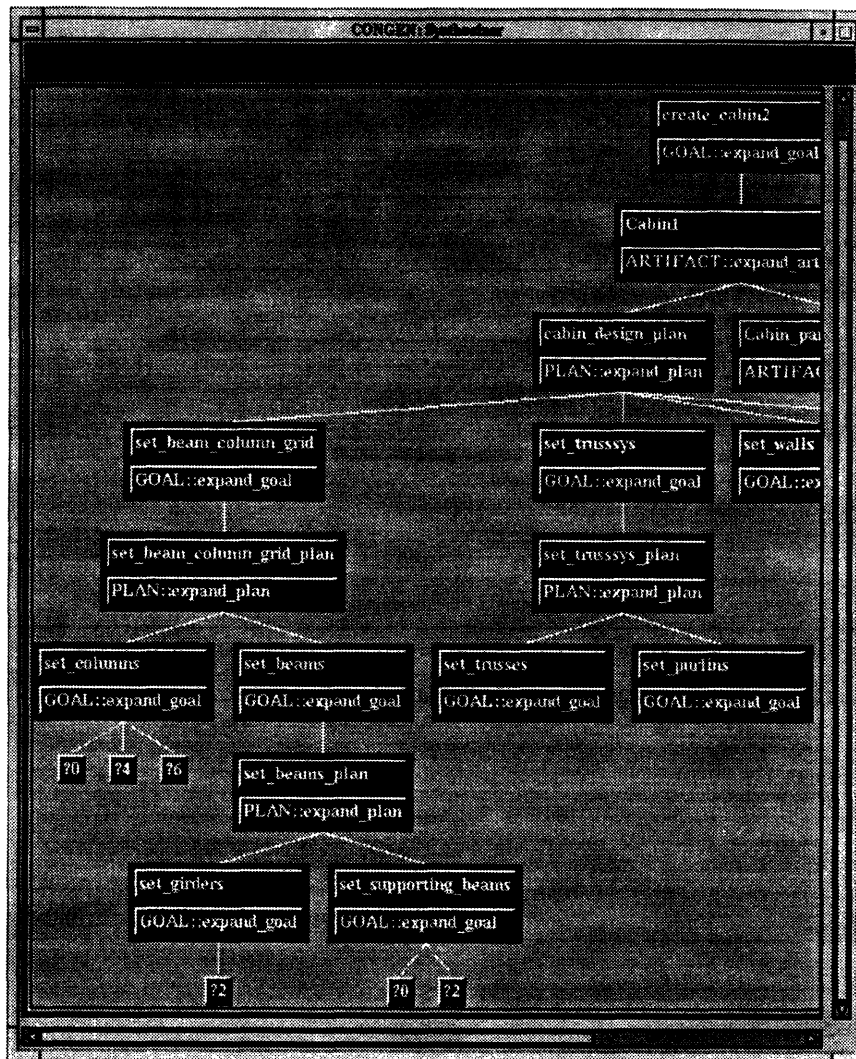


Figure 7-34: The SYNTHESIZER window for the Context: set\_trussys.set\_trussys-plan.

### 7.2.7 Executing GNOMES and displaying the geometry

To execute GNOMES Geometric Modeler, select **EXECUTE** → **GEOMETRIC MODELER** in the MAIN CONSOLE window. After the GNOMES window is shown, go back to the SYNTHESIZER window and select **GEOMETRY** → **SHOW GEOMETRY**. After selecting, you should go back to the GNOMES window, and the various configurations will be shown, such as those in figure 7-36, figure 7-37, and figure 7-38.

### 7.3 Summary

---

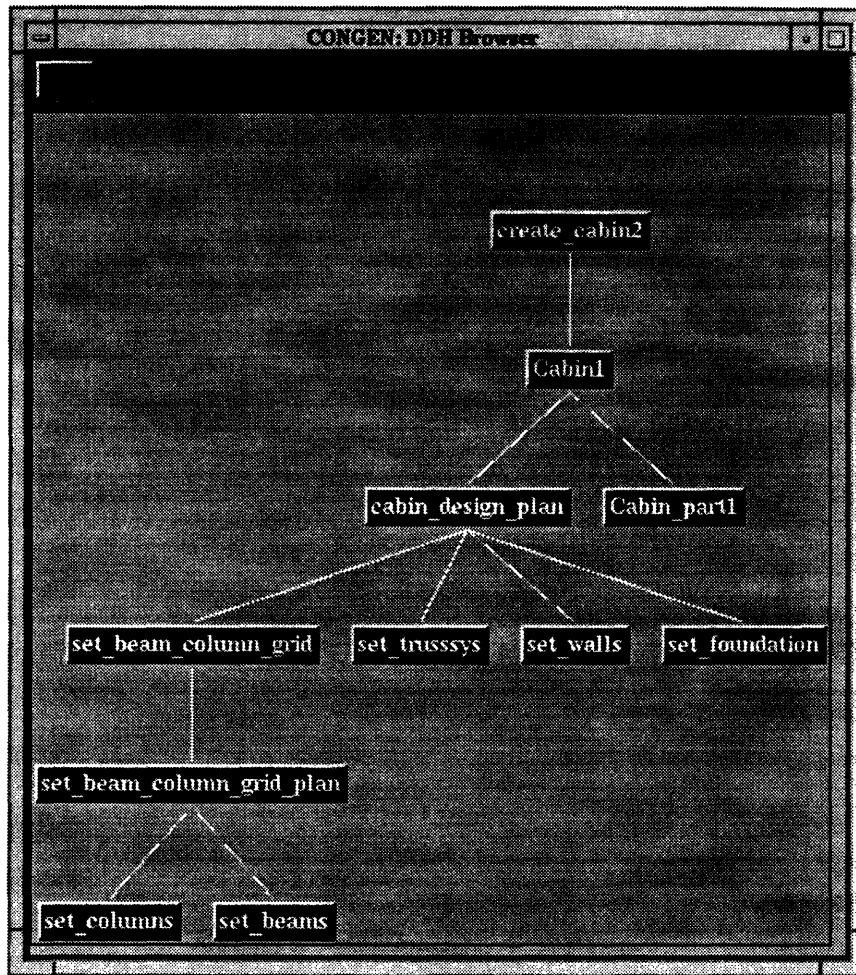


Figure 7-35: The CONTEXT BROWSER window for *set\_beam\_column\_grid.set\_beam\_column\_grid\_plan* .

### 7.3 Summary

Tutorial 4 presents a complete Cabin Design application consisting of a representative design example. Most concepts outlined in the previous chapters have been used extensively and expanded to develop this application. By finishing this tutorial, you should be able to develop an application of your own.

The next chapter deals with CONGEN limitations and future research directions.



### 7.3 Summary

---

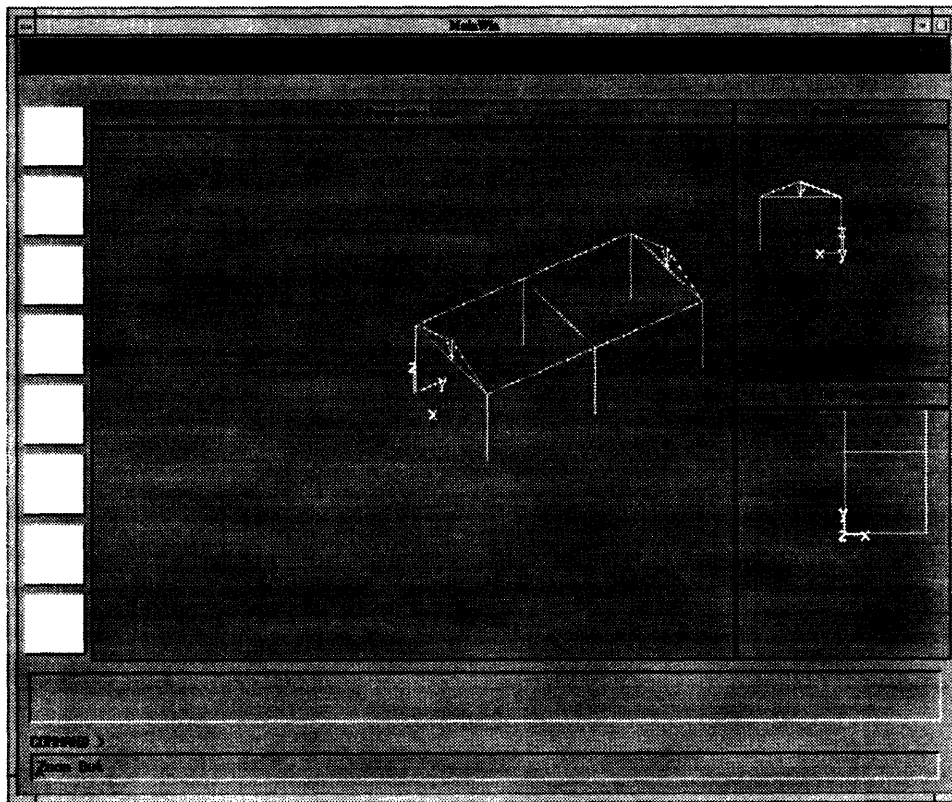


Figure 7-36: The Geometric Modeler for Cabin configuration with 6 columns, 2 girders, 1 supporting beam, and two trusses.

### 7.3 Summary

---

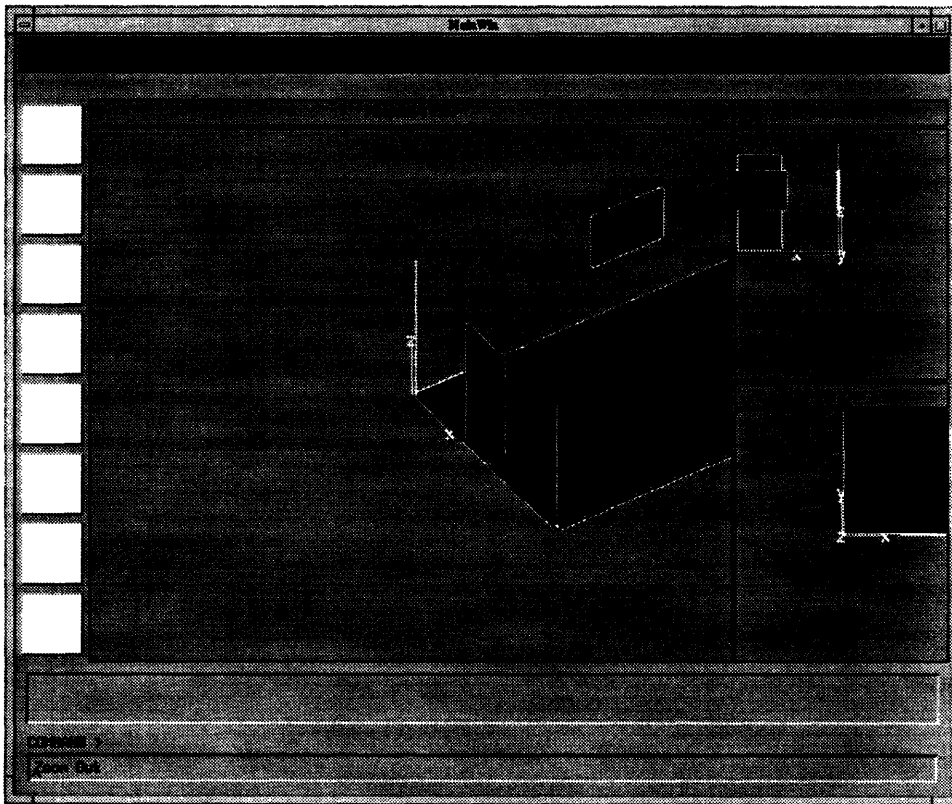


Figure 7-37: The Gnomes window showing configuration with 4 columns, wall openings, and mat foundation.

7.3 Summary

---

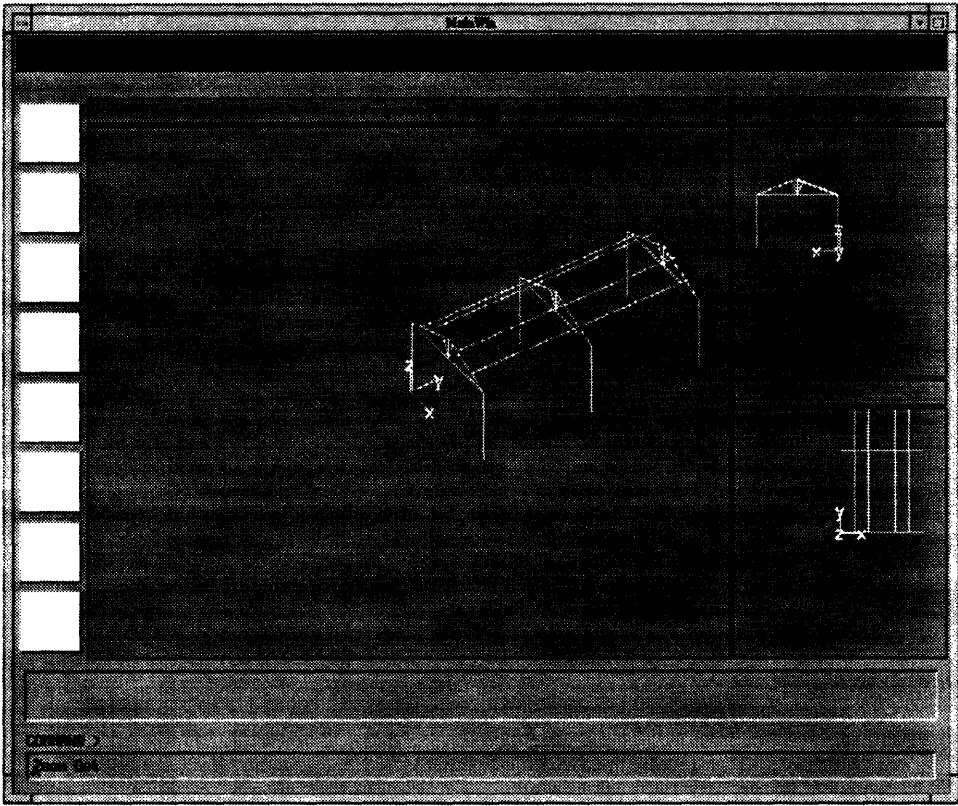


Figure 7-38: The GNOME window showing configuration with 6 columns, 3 trusses and 4 purlins.

---

## Chapter 8

# Summary and Future Work

This thesis has presented a guide to the implementation of CONGEN along with a real world example. The central idea is to provide a documentation for the users of CONGEN to develop their own applications. This chapter presents some conclusions drawn from the experience in developing CONGEN applications. Section 8.1 summarizes the overall implementation. Section 8.2 concludes the thesis with some recommendations on future developments of CONGEN.

### 8.1 Summary

One of the critical stages in the design process is conceptual design. Subsequent design stages depends heavily on the conceptual design solutions. Later on, the alternative solutions will complement the initial design with more detailed information and further modifications. A good conceptual design support usually leads to a fully functional product satisfying various constraint requirements. Performing the conceptual design tasks in the early design stages will smooth the final design's criteria satisfaction process [11].

This thesis has presented an approach towards building a specific domain application on top of a knowledge-based design support system, mainly structural engineering area. The conceptual design presented here does not concern itself with the quantitative analysis process because the detail data may not be available at the early stages of the design

## 8.1 Summary

---

process. Conceptual design phase distinguishes itself from the other stages because it primarily deals with qualitative criteria to simulate the behavior of the artifacts such as material selection, spatial relationships, and rule-of-thumb domain knowledge [9]. The later design stages then proceeds with evaluating and testing the artifacts quantitatively such as the structural analysis of each structural member / artifact.

The structural design application developed on CONGEN utilizes the empirical knowledge of how a structure is conceptualized in the initial stages. For example, the knowledge involved may require that “For weak soil and poor condition of the site, it is better to use mat foundation than spread foundation”. This kind of knowledge is very helpful for the designer to consider the basic artifacts’ behaviors. The overall behaviors of all artifacts will affect the whole structure’s behavior as well.

In developing the application on CONGEN, I believe that CONGEN holds a very bright potential in its role as the platform of conceptual design process. The support of knowledge available to the designers and the visualization capability makes the conceptual design process more flexible and adaptive to changes in the functional requirements. The embedded knowledge guides the designers in producing a satisfactory design while the visualization tool simulates real life artifact models in its final form.

Another important point that has to be noted is that CONGEN is highly flexible. CONGEN is domain independent and can be used in any domain to build its applications. Moreover, CONGEN can represent any visual representation of the artifacts with its geometric modeler. Design engineers from Mechanical Engineering, for example, can utilize CONGEN as their choice of platform in order to develop a conceptual design application. CONGEN’s modeling capabilities such as storing the domain knowledge, providing the constraint management scheme, and modeling the design visually make it ideal for developing conceptual design system.

Moreover, CONGEN is highly modularized. This aspect supports future expansion and reallocation of existing modules within the framework of CONGEN. More modules can be attached or decoupled from CONGEN’s framework according to the required specifications of the design application and the specific needs of the users. Currently we are in the process

## 8.1 Summary

---

of integrating GROWLTIGER, a structural analysis program developed at MIT to support the Cabin Design Application.

The issues arising in the process of developing Cabin Design Application can be categorized as follows:

1. *Knowledge Acquisition.* The source of knowledge used in the structural engineering application comes from two primary sources: an expert structural engineer and textbooks. The developer's lack of experience in knowledge acquisition made the development process harder to accomplish because the knowledge sources are either too complete, or incomplete. Textbooks provide a wealth of knowledge, yet with various approaches and different techniques. The textbooks can not provide a simple implementable procedure to solve a problem in the application.

On the other hand, the initial discussions of Cabin Design development saw a lot of design procedure hierarchies being proposed and dropped. The experience of the real life engineer provide a better guidance in selecting the best design logic at the end. Had we had more experience in knowledge acquisition, the task of synthesizing information and knowledge from the source would have been easier to complete.

2. *Lack of Documentation.* Implementing an application on top of CONGEN without any practical documentation proved to be quite challenging. The only documentation available was the theoretical implementation of the CONGEN designer. CONGEN was never tested by a real user before. This results in confusion of the end-user in handling even simple tasks in CONGEN. The need for clear and concise documentation of CONGEN was realized immediately after getting involved with CONGEN for the first time.
3. *Evolving Application Structure.* One of the advantages of CONGEN is that it permits flexibility in representing the design process hierarchy. The concepts of goals, plans, artifacts, and contexts provide essential building blocks for the application. However, this advantage proved to be somewhat ambiguous, because the application structure tends to evolve from time to time. The application evolution is needed in

## 8.2 Future Work

---

order to overcome CONGEN limitations and different approaches to a subproblem. Therefore, it is very important for the user to define clear functional descriptions of the product-process knowledge in the initial stages of application development. The clear segregation will help the user avoid spending too much time searching different approaches to a subproblem.

## 8.2 Future Work

As an evolving design support system, CONGEN will still inherit many enhancements and shortcomings in the future. Right now, the version of CONGEN has incorporated the new version of COSMOS. With its modularized architecture, independent modules can be upgraded and attached to CONGEN easily. However, the disadvantage is that the bugs and errors of the newly attached module are also integrated into CONGEN.

These limitations severely crippled the capability of CONGEN to provide more flexibility to the application development process. In addition to the limitation, there are several enhancements in various areas of CONGEN which need to be incorporated such as:

1. The need for online-help.
2. The need for utility to migrate databases / applications from one volume to another.
3. The need for functions to print the contents of database to a text file to be evaluated.
4. The need for to represent the complete list of attributes of an instance in the instance editor inherited either from its own class definition or the parent class attributes.
5. The need for informing the user whenever the user switches a context to a different one.
6. The need to expand basic data types in CONGEN to cover arrays, class objects along with their special access mechanisms in the ruleset of COSMOS.

The continual observation in implementing Cabin Design application on CONGEN sheds light on some further inquiries of future CONGEN development:

## 8.2 Future Work

---

1. *Speed.* CONGEN allows the user to switch contexts easily to reflect the various design stages. However, the context switching dictates that every object existing in the current context be copied to the new context generated by a decision. Therefore, most computational resources in CONGEN are geared towards copying instances within the database whenever a decision making point is reached. Moreover, the geometric modeler also contributes to the decrease of processing speed of CONGEN. It will be worthwhile to pursue several alternatives in improving the computational speed of CONGEN, such as decoupling the geometric modeler and CONGEN, or investigating the database mechanism closely to identify weak areas in persistent data operations.
2. *Transaction Management* - The transaction management in CONGEN is based on one big transaction. The major transaction is started when CONGEN starts, or after the user saves the application. It ends when the user quits (aborts the transaction) the application or saves the application (commits the changes to the database). However, with multiple applications, the need to define a better transaction management scheme is acknowledged. For example, a user is working in application A. Then the user decides to open another existing application or creates a new application without saving the application A. When the user commits the transaction in the new application, all changes in application A will also be committed. This means that the previous application's operation is not controlled sufficiently because the changes committed are not limited to the current application. Therefore, a new transaction management scheme for CONGEN is needed to support better control for the persistence of the application data.
3. *Versioning Capability.* Versioning capability is needed in order to share objects across different applications, or even different departments. Collaborative engineering mechanism requires a superior object management capability to avoid confusion among engineers. Versioning will enable the users to store gradual enhancements for an object. In addition, it will help the users trace who does what changes on a shared



## 8.2 Future Work

---

object.

4. *More Elaborate Testing.* In essence, the Cabin Design application only represents a small portion of the domain knowledge of structural engineering. To build a more complete one with bigger scope of design stages, CONGEN must be able to perform effectively and efficiently real time. The issues of performance degradation of a big application and the more various representations of an artifact's geometry must be solved immediately. The Cabin Design application should become the starting point for the future exploitation of CONGEN, where real world applications will test the limits of CONGEN further. When CONGEN can overcome the limitations, it will prove to be an invaluable conceptual design tool for the designers.
5. *More Efficient Object-Oriented Database Management System.* CONGEN utilizes EXODUS as the database platform. However, the developers of EXODUS at Wisconsin-Madison has decided not to further explore the EXODUS research. There are still lots of areas that can be made more efficient in EXODUS, such as the database limitations in migrating data from one volume to another. Therefore, the idea to migrate the whole CONGEN application to a commercial database with better speed and handling of the objects is worth considering.

---

# Bibliography

- [1] A. Goldman, K. Hussein, G. Margelis, J. Sugiono, and Y. C. Huang. GRAPHITI Detailed Design. Project for Course: *Computer Aided Engineering II: Software Engineering for CAE Systems*, 1.552 of M.I.T., May 1994.
- [2] A. Goldman, K. Hussein, G. Margelis, J. Sugiono, and Y. C. Huang. GRAPHITI Functional Specifications. Project for Course *Computer Aided Engineering II: Software Engineering for CAE Systems*, 1.552 of M.I.T., April 1994.
- [3] Adeli, H. and Balasubramanyam, K. *Expert Sytems for Structural Design - A New Generation*, Prentice-Hall, Englewood Cliffs, 1988.
- [4] Adedeji, B., *Expert Systems Applications in Engineering and Manufacturing*, Prentice-Hall, Englewood Cliffs, 1992.
- [5] Addis, W., *Structural Engineering - the nature of theory and design*, Ellis Horwood Ltd., Great Britain, 1990.
- [6] Agbayani N., *DFRAME: An Object-oriented Plane Frame LRFD Design Program with Novel Design Algorithm*, S.M Thesis, Department of Civil Engineering MIT, June 1991.
- [7] 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.
- [8] Ambrose. J., *Building Structures 2nd Ed.*, John Wiley & Sons, New York, 1993.

## BIBLIOGRAPHY

---

- [9] Arockiasamy, M., *EXPERT SYSTEMS Applications for Structural, Transportation, and Environmental Engineering*, CRC Press, Boca Raton, 1993.
- [10] Bowles, J., *Foundation Analysis and Design*, McGraw-Hill Book Co., New York, 1982.
- [11] Bozzo, L. and Fenves, G., "Qualitative Reasoning About Structural Behavior for Conceptual Design, " Structural Engineering Mechanics and Materials Report No. UCB/SEMM-92/26, Department of Civil Engineering U.C. Berkeley, California 1992.
- [12] Brown D. and Chandrasekaran B., "Investigating Routine Design Problem Solving," *AI in Engineering Design*, Vol. III, Editors: Tong, C. and Sriram, D., Academic Press, 1992.
- [13] Cheong, K.W., *A Knowledge-Based Framework for Conceptual Design*, S.M Thesis, Department of Civil Engineering, M.I.T., March 1991.
- [14] Cherneff, J., "Knowledge Based Interpretation of Architectural Drawing, " IESL Research Report R90-13, Intelligent Engineering Systems Laboratory, M.I.T., 1990.
- [15] Coduto, D., *Foundation Design - Principles and Practices*, Prentice-Hall, Englewood Cliffs, 1994.
- [16] 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.
- [17] Gorti, S.R., *From Symbol to Form: A Framework for Design Evolution*, Ph.D Thesis, Department of Civil Engineering, M.I.T., September 1994.
- [18] Hakim, M., "A Representation for Evolving Engineering Design Product Models," Technical Report, Dept. of Civil Engg., CMU, 1992.
- [19] 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

---

- [20] Johnson, A.L., "Functional Modelling: A New Development in Computer-Aided Design," *Intelligent CAD, II*, Yoshikawa, H. and Holden, T. (Editors), IFIP, 1990.
- [21] Li, H., *A Non-Manifold Geometry Modeler: An Object Oriented Approach*, S.M Thesis, Department of Civil Engineering, M.I.T., Feb. 1993.
- [22] Lin, T. and Stotesbury, S., *Structural Concepts and Systems for Architects and Engineers*, Prentice-Hall, Englewood Cliffs, 1981.
- [23] MacGinley T. and Choo, B., *Reinforced Concrete - Design Theory and Examples*, E & F.N. Spon, Great Britain, 1990.
- [24] MacGregor, J., *Reinforced Concrete - Mechanics and Design*, Prentice-Hall, Englewood Cliffs, 1988.
- [25] Mantyla,M., "A Modeling System for Top-down Design of Assembled products," *IBM Journal of Research and Development*, Volume 34, Number 5, Sept. 1990
- [26] Margelis, G., *Geometric Abstractions for Conceptual Design*, S.M. Thesis, Intelligent Systems Laboratory, Dept. of Civil and Environmental Engg., MIT, 1994.
- [27] Mittal, S. and Araya, A., "A Knowledge-Based Framework for Design," *AI in Engineering Design*, Vol. III, Editors: Tong, C. and Sriram, D., Academic Press, 1992.
- [28] Mukerjee,A., "Qualitative Geometric Design," Symposium on Solid Modeling Foundations and CAD/CAM Applications, Editors: Rossignac,J. and Turner,J., ACM Press, 1991.
- [29] Murthi, S.S. , and Addanki, S., "PROMPT: An Innovative Design Tool," *AAAI-87*, pp. 637-642, 1987.
- [30] Murthy, S., "*Synergy in Cooperating Agents: Designing Manipulators from Task Specifications*," , Ph.D. Thesis, CMU, Sept 1992
- [31] Nilson, A. and Winter, G., *Design of Concrete Structures*, McGraw-Hill Book Co., New York, 1986.

## BIBLIOGRAPHY

---

- [32] Parker, H., and Ambrose, J., *Simplified Engineering for Architects and Builders*, John Wiley & Sons, 1984.
- [33] 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.
- [34] Rossignac, J., *Advanced Representations for Geometric Structures*, Seminar given at IESL, Department of Civil Engineering, M.I.T., December 10, 1992.
- [35] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenzen, W., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [36] Salmon, C. and Johnson, J., *Steel Structures Design and Behavior 2nd Ed.*, Harper & Row, New York, 1980.
- [37] Schodek, D. L., *Structures*, Prentice-Hall, Englewood Cliffs, 1980.
- [38] Serrano, D., *Constraint Management in Conceptual Design*, PhD Thesis, MIT, Oct 1987.
- [39] Smithers, T., "AI-Based Design versus Geometry-Based Design," *Computer Aided Design* 21(3): 141-150., 1989.
- [40] Sriram, D., "Knowledge-based Approaches for Structural Design," *Topics in Engineering Vol.1*, Computational Mechanics Publications, Boston 1987.
- [41] Sriram, D., *Intelligent Systems for Engineering: Knowledge-based and Neural Networks*, Technical report, IESL, MIT, 1988.
- [42] 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.

## BIBLIOGRAPHY

---

- [43] 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.
- [44] Sriram, D. and Logcher, R., "The MIT DICE Project," *IEEE Computer*, Special Issue on Concurrent Engineering, pp. 64-65, January 1993.
- [45] 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.
- [46] Sriram, D., Wong, A., and He, L., "An Object-Oriented Non-manifold Geometric Engine," *CAD Journal*, to be published.
- [47] Steele, G. J., *The Definition and Implementation Of a Computer Programming Language Based on Constraints*, PhD Thesis, MIT, Aug 1980.
- [48] Stefik, M. and Bobrow, D.G., "Object-Oriented programming: Themes and Variations," *AI Magazine*, 1986.
- [49] Steinberg, L., "Design as Top-Down Refinement Plus Constraint Propagation," *AI in Engineering Design*, Vol. III, Editors: Tong, C. and Sriram, D., Academic Press, 1992.
- [50] Swenson, E. and Logcher, R., "Knowledge Acquisition in Conceptual Project Scheduling," "IESL Research Report R92-25, Intelligent Engineering Systems Laboratory, M.I.T, 1992.
- [51] Taranath, B., *Structural Analysis and Design of Tall Buildings*, McGraw-Hill Book Co., New York, 1988.
- [52] Tong, C. and Sriram, D., *Introduction to Artificial Intelligence in Engineering Design*, Vol. 1, Academic Press Incorporated, 1992.
- [53] White, R., Gergely, P. and Sexsmith, R., *Structural Engineering*, John Wiley & Sons, Inc., New York, 1972.

## BIBLIOGRAPHY

---

- [54] Wong, A. and Sriram, D., "Geometric Modeling Facilities for Product Modeling," Intelligent Engineering Systems Laboratory, M.I.T, 1993.
- [55] 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.
- [56] Wong, A. and Sriram, D., "SHARED: An Information Model for Cooperative Product Development," *Research in Engineering Design*, Fall 1993.
- [57] 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.
- [58] Wong, A. and Sriram, D. "An Extended Object Model for Design Representation," to be submitted to *IEEE Transactions on Knowledge and Data Engineering*.
- [59] Wright, E. *Structural Design by Computer*, Van Nostrand Reinhold, Great Britain, 1976.

---

## Appendix A

# REFERENCE MANUAL

### A.1 Reserved Keywords

The reserved keywords are:

- *attribute: object\_type* used by the Geometry class in CONGEN.
- *attribute: length* used by the Geometry class in CONGEN.
- *attribute: width* used by the Geometry class in CONGEN.
- *attribute: height* used by the Geometry class in CONGEN.
- *attribute: xpos* used by the Geometry class in CONGEN.
- *attribute: ypos* used by the Geometry class in CONGEN.
- *attribute: xpos* used by the Geometry class in CONGEN.
- *attribute: thetax* used by the Geometry class in CONGEN.
- *attribute: thetay* used by the Geometry class in CONGEN.
- *attribute: thetaz* used by the Geometry class in CONGEN.
- *attribute: sizex* used by the Geometry class in CONGEN.
- *attribute: sizey* used by the Geometry class in CONGEN.
- *attribute: sizez* used by the Geometry class in CONGEN.



## A.2 Available Methods

---

In addition to this, users are strongly advised not to use the following scheme in naming an instance of a class. For example, if the user owns a class “Column”, and wants to name a created instance of the class, DO NOT use the naming Column1, or Column2, etc.

CONGEN uses this naming convention to refer to the instances that it produces in the process. Whenever Synthesizer instantiates an instance of a class, the name of the instance follows the rule: ;classname;inum; e.g. Column1. Therefore, instead of using the following scheme:

```
MAKE (CLASS:Column OBJ:Column1 ... -> WRONG!
```

you should do:

```
MAKE (CLASS:Column OBJ:Col1 or Col_1 or Column_1
```

## A.2 Available Methods

Every method listed below can be summoned via rulefiles. Please remember not to put any space between the arguments of the method.

- *make\_part(relationshipname,childclassname,childinstancename)* This method is used to link a parent and a child with *part-of* relationship. Example:

```
Class Wheel; // Child class  
Class Car; // Parent class
```

in the rulefile:

```
(EXECUTE VAR: $rtn OBJ:Car1 make_part('leftwheel','Wheel','Wheel1'))
```

- *show\_geometry()* This method automatically updates the geometric modeler display if it is already active. Example:

```
(EXECUTE VAR: $rtn OBJ:Car1 show_geometry())
```

## A.2 Available Methods

---

- *create\_geometry(geometry\_type,instancename)* This method creates a geometry to be linked to an artifact, and assigns an instancename to the geometry. The type of geometry can be found in the Cabin Design Application Tutorial in Geometry section. Example:

```
(EXECUTE VAR: $rtn OBJ:Car1 create_geometry(7,'cargeom')
```

The type 7 denotes LINE RECTANGULAR SOLID geometry. To create a line geometry, use type 4.

- *set\_translation(Xunit,Yunit,Zunit)* This method translates a geometry as much as Xunit along X axis, Yunit along Y axis, and Zunit along Z axis. The translation units must be float, not integer. Example:

```
(EXECUTE VAR: $rtn OBJ:Car1 set_translation(100.0,100.0,200.0))
```

- *set\_rotation(Xdegree,Ydegree,Zdegree)* This method rotates a geometry as much as Xdegree along X axis, Ydegree along Y axis, and Zdegree along Z axis. The degrees must also be float, not integer. The direction of the rotation follows the rule of right thumb pointing to the positive direction of the axis and fingers denoting the direction. Example:

```
(EXECUTE VAR: $rtn OBJ:Car1 set_rotation(90.0,90.0,180.0))
```

- *propagate\_attribute\_values(childclass,childinstancename)* This method acts to propagate parent's geometry values to the children. This method is important in ensuring that the children have the same geometry as the parent. Example:

```
(RULE: create_columns22 10000
IF
(CLASS: pier OBJ: $x
((no_of_col == 2) AND
(instancename == "pier2"))
```

### A.3 Dynamic Methods

---

```
)
THEN (
(MAKE (CLASS:column OBJ: pier2col1
(width 3)
(ypos 22)
))
(EXECUTE VAR: $rtn OBJ: $x make_part(''col1'', ''column'', ''pier2col1''))
(EXECUTE VAR: $rtn OBJ: $x propagate_attribute_values(''column'', ''pier2col1''))
(EXECUTE VAR: $rtn OBJ: $x create_geometry(7, ''pier2col1''))
)
COMMENT:" Creating a column, link it to the pier, propagate the
geometry value from pier to column, and create the column geometry")
```

The above rules create a column for the pier, set up *part-of* link between the column and the pier, propagate the pier geometry values to the column, and lastly, create the geometry of the column itself based on the pier's geometry values.

- *notify\_constraint\_violation(string-message)* This method acts as the notifier to the user if the constraint is violated or there's any violation happening in the knowledge base. Example:

```
(EXECUTE VAR: $rtn OBJ: $x notify_constraint_violation(''The wheel is too small''))
```

### A.3 Dynamic Methods

The current version of CONGEN provides a powerful tool in supporting the users' need - dynamic methods. Dynamic methods are methods provided by the user, compiled and linked by CONGEN at runtime. Users do not have to shutdown CONGEN to compile and add new methods to their own applications.

Dynamic methods are defined in two ways:

1. Member functions defined within a class.
2. Independent external C functions.

The syntax to summon the dynamic method is as follows:

```
EXECUTE VAR: $return-identifier OBJ: $caller-identifier method(arguments)
```

### A.3 Dynamic Methods

---

- *\$return\_identifier* points to the variable name which will hold the return value of the method, i.e.: \$rtn.
- *\$caller\_identifier* points to the object summoning this method i.e.: \$x, \$y, etc.
- *method* represents the method name as defined by the user such as: make\_part, calculate\_load, etc.
- *arguments* are the arguments needed by the method.

The current implementation of dynamic methods only supports six basic data types to be parsed as arguments:

1. integer,
2. character,
3. float,
4. double, and
5. PString - Persistent string implementation in EXODUS.
6. PStringList - Persistent list of strings.

An example of a dynamic method implementation is as follows:

- The header file:

---

```
#ifndef truss_H
#define truss_H

#include "Artifact.h"
#include "exemplar.h"
dbclass truss:public Artifact{

private:

public:
dbvoid analyze();
int get_nodes();
```

10

### A.3 Dynamic Methods

---

```
int get_mem_no();
dbvoid set_no_nodes(dbint);
int no_nodes;
int no_mems;
int mem_inc[5][2];
int supported[4][2];
double jcoord[4][2];
double member_area[5];
double mem_youngs[5];
double loads[8];

...
};
#endif
```

---

- The program file:

---

```
#include "/mit/jdchiou/congen/ar/truss.h"
#include "meta_class.h"
#include "ccString.h"
#include "data_manager.h"
#include "InstanceManager.h"
#include <iostream.h>
#include <math.h>

// All the declaration of class Truss is put here by CONGEN (code automatically generated)
...

// The following methods are other external methods entered in CONGEN

dbint truss::get_nodes(){
return no_nodes;
}
dbint truss::get_mem_no(){
return no_mems;
}
dbvoid truss::set_no_nodes(dbint i){
no_nodes = i;
}

// The following methods are inserted manually using a text editor (EMACS)

void truss(int no_mems, int no_nodes, int mem_inc[][2],
double jcoord[][2], int supported[][2], double *loads, double
*member_area, double *mem_youngs) ;

dbvoid truss::analyze()
// analyze function calls the "truss" C function from inside
```

---

### A.3 Dynamic Methods

---

```
// this is a dummy function to implement external method invocation
{

// the body of function "analyze"
...

// prepare data to be passed to the function "truss" below
truss(no_mems,no_nodes,mem_incl,jcoord1,supported1,loads1,member_area1,mem_youngs1);
}

// Function "truss" declaration in C
// This function must be inserted manually by the user using
// a text editor.
void truss(int no_mems, int no_nodes, int mem_inc[][2],
           double jcoord[][2], int supported[][2], double *loads, double
           *member_area, double *mem_youngs)
{
// the body of function "truss"..
}
```

- 
- The rulefile:

```
(RULE: create_geomtry 10
IF
(CLASS: truss OBJ: $x
(no_mems == 0)
)
THEN (
(EXECUTE VAR:$rtn OBJ:$x set_no_nodes ($rtn-1) )
(EXECUTE VAR:$rtn OBJ:$x analyze () )
)
COMMENT:"Rule to execute methods defined inside the class truss")
```

---

The external methods can be written in two ways:

1. By entering the method and edit the method inside CONGEN itself (from the PRODUCT KNOWLEDGE menu). This approach is effective when the method to be inserted is relatively short.
  2. By entering the method manually - editing the class files (.e and .h) using a text editor i.e.: EMACS. This approach is effective when you need to insert a long function.
-

## A.4 Application Script

---

In the example, the function *get.nodes()* from the above example is entered using CONGEN's Product Knowledge facility. The function is very short, so it is better to use CONGEN's facility. On the other hand, the function *analyze()* is quite long. It is not effective to use CONGEN's facility to enter the source code. Therefore, we use EMACS to edit the "truss.h" and "truss.e" files manually.

## A.4 Application Script

Application scripts are provided to help the user shortcut the tedious process of entering many goals, plans, and classes manually.

The steps to prepare an application script are as follows:

1. Prepare the entries for goals and plans before writing the script.
2. Use the template from Cabin Design application script listed in Appendix E. This template will help explain the basic entry rules for the goals and plans. You must use a different name for your own script. In addition, you should change the name of the goals or plans and each field entry as outlined in the example.
3. Prepare the classes to be entered, complete with their attributes.
4. Enter the classes as written in the Cabin Design sample. You should not enter all attributes into the script. The attributes should be put in another file, for example: Cabin class is linked to Cabin.attr file containing all the Cabin attributes.
5. Enter the subclass relationships between the classes. The relationship examples can be seen at the end of the Cabin Design script.
6. Compile the script, i.e.: **make create\_cabin.o**.
7. Execute CONGEN, and in the MAIN CONSOLE window, select the submenu **HELP**. After you click on **HELP** a window should be presented to you as shown in figure A-1. You must enter the name of the script program without the extension *.o*.
8. Wait for the script to finish executing.

## A.4 Application Script

---

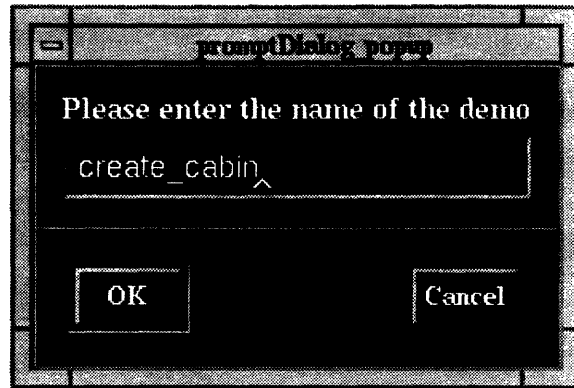


Figure A-1: The HELP window with the *create\_cabin* script entry.

9. Make the application / compile the classes using the PRODUCT KNOWLEDGE window menu **FILE** → **MAKE APP**.

**NOTE:** The statements: *E\_start\_transaction* and *E\_commit\_transaction* should be used often in the application. Due to a bug in EXODUS that crashes CONGEN whenever there are too many objects to be committed, you should put the statement pairs more often in the script code.



---

## Appendix B

# COSMOS Knowledge Base Rule in CONGEN

### B.1 COSMOS Rule Grammar

The BNF notation for the knowledge base in COSMOS is given below.

```
<knowledge base> ::= <knowledge base> <commentblock> |  
                  <knowledge base> <rule>  
<commentblock> ::= /* <comment> */  
<comment>      ::= identifier | <comment> identifier  
<rule>         ::= (RULE: <rulename> <ruleprior> if <condblock>  
                  then <actblock> <comblock>)  
<rulename>     ::= identifier  
<ruleprior>    ::= number  
<condblock>    ::= (<condblock> OR <condblock>) |  
                  (<condblock> AND <condblock>) |  
                  (CLASS: identifier OBJ: $identifier <slot_ops>)  
<slot_ops>     ::= (<slot_ops> OR <slot_ops>) |  
                  (<slot_ops> AND <slot_ops>) |  
                  (identifier <lop> <expr>)
```

## B.1 COSMOS Rule Grammar

---

**<lop>** ::= GT | GE | LT | LE | EQ | NE

**<expr>** ::= **number** |  
string |  
\$identifier|  
VAR |  
(<expr>) |  
<expr> <aop> <expr> |

**<aop>** ::= + | - | \* |

**<actblock>** ::= (<actionstatement>+)

**<actionstatement>** ::= (MODIFY OBJ: \$ identifier [(identifier<expr>)]+) <lsno><lno> |  
(MAKE CLASS: identifier OBJ: \$identifier [(identifier <expr>)]+) |  
(REMOVE OBJ: \$identifier) |  
(PRINT <expr> [,<expr>]\*) |  
(READ PROMPT: string, VAR: \$identifier, TYPE:identifier) |  
(BIND VAR: \$identifier<expr>) |  
(EXECUTE VAR: identifier OBJ: identifier method) |  
(DISPLAY string)

**<lsno>** ::= **number**

**<lno>** ::= **number**

**<comblock>** ::= COMMENT: string

### B.1.1 Explanation of the Grammar

The knowledge base (rule base) is composed of comments and rules as given by the grammar rule given below.

**<knowledge base>** ::= <knowledge base> <commentblock> |  
<knowledge base> <rule>

## B.1 COSMOS Rule Grammar

---

### B.1.2 Comment Block

`<commentblock> ::= /* <comment> */`  
`<comment> ::= identifier | <comment> identifier`

The comments are identifiers enclosed by the characters `'/*'` and `'*/'`. Any text enclosed like this is a comment and are ignored by the parser. This form of comment can occur only outside the rules. They cannot be used to omit parts of rules. The grammar for a rule is as follows:

### B.1.3 Rule

`<rule> ::= (RULE: <rulename> <ruleprior> if <condblock>`  
`then <actblock> <comblock>)`

The rules are the main part of the knowledge base. A matched set of parenthesis delimits a rule. Inside parenthesis, the rule begins with the keyword `'RULE:'`. The rule is made of the following components. The keyword `'RULE:'`, rule name, rule priority, the if portion or antecedent or condition part, the then portion or consequent or action part, and the comment part.

### B.1.4 Rule Name

`<rulename> ::= identifier`

The rule name is an identifier **uniquely** labeling a rule. Each rule in the rule base has a **unique** rule name. It is an identifier beginning with a character and can be 25 characters long. The rule name must consist of alphanumeric characters, and should **not** contain any blanks.

## B.1 COSMOS Rule Grammar

---

### B.1.5 Rule Priority

**<ruleprior> ::= number**

The rule priority is a number. The higher the number, the higher the priority of the rule.

### B.1.6 Condition Block

The unit condition block element is delimited by parenthesis and the portion inside the parenthesis has three parts.

1. the class name beginning with 'CLASS:' keyword and class name identifier
2. the object name beginning with 'OBJ:' keyword and object identifier **prefixed** by the '\$' symbol.
3. the conditional expression. It is recursively made of the unit condition block elements interconnected by conjunctive or disjunctive logical connectives represented by the keywords 'AND' and 'OR' respectively. The negation operator NOT is also used for recursive definition of the condition block.

**<condblock> ::= (<condblock> OR <condblock>) |  
(<condblock> AND <condblock>) |  
(CLASS:identifier OBJ:\$identifier <slot\_ops>)**

### B.1.7 Test Expression

The unit test expression is an identifier and an arithmetic expression connected by comparative operators. The comparative operators used are *i*, *i*=, *i*, *i*=, ==, != to represent greater-than, greater-than-or-equal, less-than, less-than-or-equal, equal, not-equal respectively. The grammar for the test expression is given below.

**<slot\_ops> ::= (<slot\_ops> OR <slot\_ops>) |**

## B.1 COSMOS Rule Grammar

---

```
( <slot_ops> AND <slot_ops> ) |  
  ( identifier <lop> <expr> )  
<lop> ::= GT | GE | LT | LE | EQ | NE
```

While, in general test expressions are used to match values to attribute variables of classes, they can be used to construct more complex pattern matching cases.

For example:

**attribute op val** : If **val** is not defined, and **op** is '==', the comparison is reduced to a **binding**, wherein the variable **val** is bound to equal **attribute**. This fact may be used to perform matches of the kind **attribute1 op attribute2**, by reducing the conditional to a **BOOLEAN AND** of the two conditions, **attribute1 == \$val** and **attribute2 op \$val**. **The scope of such bound variables is, however, limited to the rule !**

Note that when **val** is an arithmetic expression, the syntax for the conditional would be as in **attribute op {2.0 + \$val}**, to use an example.

### B.1.8 Arithmetic Expression

The arithmetic expression is formed of number, string, object, variable, or these things except strings combined by the binary arithmetic operators of addition, subtraction, multiplication and division. The unary operator used is the negation. The **NOT** unary operator is the only allowed logical operator. The string is represented by characters enclosed in double quotes "".

```
<expr> ::= number | string | $ identifier | VAR |  
          ( <expr> ) | <expr> <aop> <expr>  
<aop> ::= + | - | * |
```

### B.1.9 Example of Condition Block

An example of a condition block will be:

## B.1 COSMOS Rule Grammar

---

```
((CLASS:car OBJ:buick
    (spare_wheel LT 1)) AND
    (service_shop NE "available")) OR
    (car_junked EQ "true")) AND
((CLASS: ..... ) OR
(CLASS: .....)))
```

**IMPORTANT NOTE:** Be sure that the parenthesis are used only for couples, for example (COND1), ((COND1) AND (COND2)), (((COND1) AND (COND2)) AND (COND3)). You cannot use conditional statements in more than two conditions at one point of testing within the same parenthesis.

### B.1.10 Action Block

The next part of the rule is the consequent or the action part of the rule. Each rule has one action block. The action block is composed of one or more action statements. When the condition part is satisfied, the action statements are executed in the textual sequence. There are seven different types of action statements:

1. *MODIFY* - This statement modifies the values of the attribute in a class.
2. *MAKE* - This statement creates an instance of a class. You can also attach initial values of attributes within the *MAKE* statement.
3. *REMOVE* - This statement removes an object from the working memory.
4. *READ* - This statement reads input from a popped dialog window.
5. *BIND* - This statement provides a variable name to be bound to a
6. *EXECUTE* - This statement enables the user to execute an external method. Please refer to Appendix A for further information.
7. *DISPLAY* - This statement displays a picture taken from an Xwindow screendump.

The action statements are expressed by the following grammar rules.

## B.1 COSMOS Rule Grammar

---

```
<actblock> ::= (<actionstatement>+)  
<actionstatement> ::= (MODIFY OBJ:$identifier  
                        [(identifier <expr>)]+) <lsno> <lnno> |  
                        (MAKE CLASS: identifier OBJ: $identifier  
                        [(identifier;expri)]+) |  
                        (REMOVE OBJ: $identifier ) |  
                        (READ PROMPT: string,VAR: $identifier,TYPE:identifier ) —  
                        (BIND VAR: $identifier <expr>) |  
                        (EXECUTE VAR:identifier OBJ:identifier method) |  
                        (DISPLAY string)
```

### B.1.11 Example of Action Block

Examples of the Action Block are provided below.

```
THEN (  
    (MODIFY (OBJ: $x  
            (problem "dead_battery")) 1000 0.001 )  
    (MAKE (CLASS: Aclass OBJ:$x  
          (problem "dead_battery")))  
    (REMOVE OBJ: $x)  
    (READ PROMPT: "read a value", VAR:$x,TYPE:whatever)  
    (BIND VAR: $num 550.34)  
    (EXECUTE VAR: $rtn OBJ: $x method)  
    (DISPLAY "/pathname/filename.xwd")  
)
```

### B.1.12 Inline Comment Block

There is another form of comment used in COSMOS which forms the part of rule. This forms one of the segments of a rule. This comment is preceded by the keyword

## B.2 Helpful hints in building and running rules in COSMOS

---

'COMMENT:' followed by the string representing the comment. The grammar for the comment is given below.

```
::= <comblock>
```

## B.2 Helpful hints in building and running rules in COSMOS

- **VERY IMPORTANT:** A rule will always be fired whenever the condition it tests is true. There are some ways to overcome this problem, for example,

```
IF (CLASS: Column OBJ: \ $x
((c\_length == $len) AND
  (length != $len))
THEN
(MODIFY OBJ: \ $x
  (length $len) 1000 0.001)
```

In this case, length is a variable modified by the rule. Initially, length is not equal to the column length, therefore this rule is fired. After the rule is fired, length is set to the value of column length, therefore this rule is not fired for the second time. If the (length != \$len) is omitted, then the condition will always be true, and CONGEN will be caught in an infinite loop.

- The above behavior of the rulefile is very helpful in doing iterations, for example, you have to evaluate every instance from an unknown quantity. One way to iterate through all the instances is to make a rulefile that intercepts any of the instance immediately, and then set a flag such as *dbint processed*; to 1, and 0 if not yet intercepted. For example:

---

```
(RULE: createsource 1000
IF
```

---



## B.2 Helpful hints in building and running rules in COSMOS

---

```
(CLASS: source OBJ: $x
(s_int == 0)
)
THEN (
(MAKE (CLASS: source OBJ: sourcenew
(s_int 1)
(s_char "source1")
(s_float 0.0)
(processed 0)
))
(MAKE (CLASS: source OBJ: sourcenew2
(s_int 2)
(s_char "source2")
(s_float 0.0)
(processed 0)
))
)
COMMENT:"Rule to create the source"

(RULE: readsources 1000
IF
(CLASS: source OBJ: $x
(((processed == 0) AND
(s_char == $name)) AND
((s_int == $int) AND
(s_float == $float)))
)
THEN (
(READ PROMPT:"We found a source, would you name a new target?" VAR: $tname TYPE: s)
(BIND VAR: $newint 5/2)
(MAKE (CLASS: target OBJ: dummy
(t_int $int)
(t_char $tname)
(t_float $newint)
(instancename $tname)
))
(MODIFY (OBJ:$x
(processed 1)
) 1000 0.001)
)
COMMENT:"Rule to read in targets"

(RULE: readtargets 1000
IF
(CLASS: target OBJ: $x
(instancename == "try")
)
THEN (
(READ PROMPT:"We found the target, would you name a new target?" VAR: $tname TYPE: s)
(MODIFY (OBJ:$x
(processed 1)
) 1000 0.001)
)
)
```

## B.2 Helpful hints in building and running rules in COSMOS

---

COMMENT:"Rule to read in targets")

---

- Be sure to check whether the classnames and the attributes to corresponding classes exist, because when they do not exist, the rule is never fired.
- Use DISPLAY statement to show a screen capture from a window (.xwd suffix). The DISPLAY statement is very helpful in showing a screen capture of a geometry without having to execute the Geometric Modeler.
- Remember that after MODIFY statement, you have to add more parameters as listed above such as " 10000 0.001) "- Check the OBJ identifier of the classes where you build your conditions. There has been some occasions when the OBJ identifiers are the same for the class, and CONGEN doesn't react to the rule.
- Check on the parenthesis coupling for the conditional statement. You have to make sure that cparenthesis match with each other, and there are no more than two conditions inside a conditional composite.
- When you use strings in the rules, be sure to check the quotes whether they match or not.
- OR statement can also be implemented using two rules complementing each other. For example if A OR B, can be implemented as IF A .... and IF B .... while they perform the same exact operations.
- Check on the executed methods, whether they exist or not in the classes or in the libraries.
- Remember that when you use the equal operator (==), it behaves as a conditional operator and assignment operator if the right hand side is a new variable.

---

## Appendix C

# Installation, Configuration & Troubleshooting

### C.1 Installation

#### C.1.1 How to Obtain CONGEN

CONGEN can be obtained by anonymous ftp from `iesl.mit.edu` from the directory `pub/dice/congen` as the file `congen.tar.Z`. For further information, contact `vemula@mit.edu`.

#### C.1.2 Requirements

Your site must meet the following minimum requirements before installing CONGEN:

- CONGEN runs on the family of Sun SPARC stations under Sun OS 4.1. CONGEN runs as a client/server system requiring the EXODUS server process and the CONGEN client process. Each process can run on the same or different machine.
- A minimum of 45 mb of disk space is needed to install all the modules in CONGEN including COSMOS, GNOMES, COPLAN, and CONGEN itself.
- Installed XWindows Version 11 Release 5.
- Installed EXODUS Object-oriented Database Management System.

## C.1 Installation

---

### C.1.3 Pre-installation

Before you install the program, there are several steps to be taken to ensure a successful installation:

1. Ensure that the directories where you are installing CONGEN are read/writeable.
2. Use the uncompress command

```
uncompress congen.tar.Z
```

to expand the file.

3. Use the tar command to extract the installation files in the target directory:

```
tar -xvf congen.tar
```

### C.1.4 Compiling CONGEN

All the source code is in the directory src, library archives are in the directory lib, and the bin contains the congen executable. There are README files in each of the subdirectories to help identify the wanted files.

### C.1.5 Environment Variables

Environment variables which need to be set are:

```
setenv DISPLAY <machine-name>:0.0 (enter your machine name instead)
```

```
setenv HOOPS_PICTURE X11/<machine-name>:0.0
```

```
setenv EVOLID (the volume number of the database)
```

```
    i.e.: 4008
```

```
setenv CONDIR (the directory where your congen is installed)
```

```
    i.e.: /mit/congen/bin
```

```
setenv CONGEN_USER_AR (the directory where the class archive is installed)
```

```
    i.e.: /mit/jsugiono/congen/ar
```

## C.1 Installation

---

`setenv INCDIR` (the directory where the `congen` include sourcefiles is placed)  
i.e.: `/mit/congen/src/include`

The Xresource files used by CONGEN is the file `.Xdefaults` in the home directory

The database volume needs to be set up according to the instructions provided in the EXODUS OODBMS release. Note that if the Exodus storage manager and the E directories are not installed, the program will neither run, nor will any of the source code compile since the application needs to use the `eg++` compiler provided by the EXODUS OODBMS release.

The EXODUS OODBMS source code and documents can be obtained from the University of Wisconsin, Madison. The Object Management (OBM) part of COSMOS interacts with the EXODUS storage manager server in order to create and retrieve the persistent data. For each class the user defines through the COSMOS User Interface, the OBM generates code in the language E (which is a superset of C++ with support for persistence). This code is compiled by the E compiler and linked in at run-time.

The COSMOS user needs to install both EXODUS v.3.1 and E-2.3.3 before using COSMOS. The EXODUS Storage Manager and the E Compiler can be obtained by anonymous ftp from `ftp.cs.wisc.edu`.

### C.1.6 Database Environment Variable

The EXODUS database variable to be set are:

- In *.cshrc* file - `setenv EVOLID` (Database Volume ID).
- In *.sm-config* file - `client*mount: (DB Vol ID) (Port number)@(servername)`.

---

## Appendix D

# Tutorial 2 & 3 Listings

### D.1 TUTORIAL 2 - SIMPLE SLAB

#### D.1.1 Main Listing of the rulefile Tutorial\_2.rul = Alternative 1 - One Goal, One Plan rulefile.

---

```
(RULE: createslab 0
IF
(CLASS: Slab OBJ: $x
((s_length == 0) AND
(s_width == 0))
)
THEN (
(MODIFY (OBJ:$x
(s_length 40)
) 1000 0.001)
(MODIFY (OBJ:$x
(s_width 20)
) 1000 0.001)
)
COMMENT:"Rule to create a slab for tutorial 2")

(RULE: create_geometry 10
IF
(CLASS: Slab OBJ: $x
(((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
)
THEN (
(EXECUTE VAR:$rtn OBJ:$x create_geometry (16,"slabgeom2"))
```

## D.1 TUTORIAL 2 - SIMPLE SLAB

---

```
(MODIFY (OBJ:$x
(plane 2)
)1000 0.001)
(MODIFY (OBJ:$x
(length $len)
)1000 0.001)
(MODIFY (OBJ:$x
(width $wid)
)1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation (0.0,36.0,0.0) )
(EXECUTE VAR:$rtn OBJ: $x set_rotation (90.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x show_geometry ( ))
)
COMMENT:"Rule_to_set_up_the_slab_geometry")
```

---

30  
40

## D.1 TUTORIAL 2 - SIMPLE SLAB

---

### D.1.2 Alternative 2 - One Goal, One Plan, Two Subgoals

**modslab.rul**

---

```
(RULE: createslab 0
IF
(CLASS: Slab OBJ: $x
((s_length == 0) AND
(s_width == 0))
)
THEN (
(MODIFY (OBJ:$x
(s_length 40)
) 1000 0.001)
(MODIFY (OBJ:$x
(s_width 20)
) 1000 0.001)
)
COMMENT:"Rule to create a simple slab for Tutorial 2")
```

---



## D.1 TUTORIAL 2 - SIMPLE SLAB

---

### geoslab.rul

---

```
(RULE: create_geometry 10
IF
(CLASS: Slab2 OBJ: $x
(((s_length == $len) AND
(s_width == $wid) ) AND
(length != $len))
)
THEN (
(EXECUTE OBJ: $x create_geometry(16,"slabgeom2"))
(MODIFY (OBJ:$x                                10
(plane 2)
)1000 0.001)
(MODIFY (OBJ:$x
(length $len)
)1000 0.001)
(MODIFY (OBJ:$x
(width $wid)
)1000 0.001)
(EXECUTE OBJ: $x set_translation(0.0,36.0,0.0))
(EXECUTE OBJ: $x set_rotation(90.0,0.0,0.0))      20
(EXECUTE OBJ: $x show_geometry())
)
COMMENT:"Rule to set up the slab geometry")
```

---

## D.2 TUTORIAL 3 - BOX APPLICATION

---

### D.2 TUTORIAL 3 - BOX APPLICATION

#### D.2.1 Main Listing

**create\_ass\_eff.rul**

---

```
(RULE: createass 1000
IF
(CLASS: Assembly3 OBJ: $x
((a_length == 0) AND
(a_width == 0))
)
THEN (
(MODIFY (OBJ:$x
(a_length 1000)
) 1000 0.001)
(MODIFY (OBJ:$x
(a_width 500)
) 1000 0.001)
(MODIFY (OBJ:$x
(a_depth 300)
) 1000 0.001)
)
COMMENT:"Rule_to_create_a_simple_assembly_box_for_test_3")
```

---

10

## D.2 TUTORIAL 3 - BOX APPLICATION

---

### create\_top.rul

---

```
(RULE: createtop 1000
IF
((CLASS: Slab3 OBJ: $x
((s_length == 0) AND
(s_width == 0))
)
AND
(CLASS: Assembly OBJ: $y
(((a_length == $len) AND
(a_width == $wid)) AND
(a_depth == $dep))
))
THEN (
(MODIFY (OBJ:$x
(s_length $len)
(s_width $wid)
(s_depth 2)
) 1000 0.001)
)
COMMENT:"Rule to create the top cover for test 3")

(RULE: top_geometry 10
IF
(CLASS: Slab3 OBJ: $x
(((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
)
THEN (
(EXECUTE VAR: $rtn OBJ: $x create_geometry(16,"slabgeom2"))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(0.0,0.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(0.0,0.0,30.0))
(EXECUTE VAR: $rtn OBJ: $x show_geometry( ))
)
COMMENT:" Rule to set up the geometry of the top ")
```

---

## D.2 TUTORIAL 3 - BOX APPLICATION

---

### create\_bottom.rul

---

```
(RULE: createbottom 1000
IF
((CLASS: Slab3 OBJ: $x
((s_length == 0) AND
(s_width == 0))
)
AND
(CLASS: Assembly OBJ: $y
(((a_length == $len) AND
(a_width == $wid)) AND
(a_depth == $dep))
))
THEN (
(MODIFY (OBJ:$x
(s_length $len)
(s_width $wid)
(s_depth 2)
) 1000 0.001)
)
COMMENT:"Rule to create the bottom cover for test 3"

(RULE: bottom_geometry 10
IF
(CLASS: Slab3 OBJ: $x
(((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
)
THEN (
(EXECUTE VAR: $rtn OBJ: $x create_geometry(16,"slabgeom2"))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(0.0,0.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(0.0,0.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the bottom ")
```

---

## D.2 TUTORIAL 3 - BOX APPLICATION

---

### create\_east.rul

---

```
(RULE: createeast 1000
IF
((CLASS: Slab3 OBJ: $x
(s_length == 0) AND
(s_width == 0))
)
AND
(CLASS: Assembly OBJ: $y
(((a_length == $len) AND
(a_width == $wid)) AND
(a_depth == $dep))
))
THEN (
(MODIFY (OBJ: $x
(s_length $len)
(s_width $dep)
(s_depth 2)
) 1000 0.001)
)
COMMENT:"Rule to create the east cover for test 3")
```

10  
20

```
(RULE: east_geometry 10
IF
(CLASS: Slab3 OBJ: $x
(((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
)
THEN (
(EXECUTE VAR: $rtn OBJ: $x create_geometry(16,"slabgeom2"))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(90.0,0.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(0.0,0.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the east ")
```

30  
40

---

## D.2 TUTORIAL 3 - BOX APPLICATION

---

### create\_west.rul

---

```
(RULE: createwest 1000
IF
((CLASS: Slab3 OBJ: $x
(s_length == 0) AND
(s_width == 0))
)
AND
(CLASS: Assembly OBJ: $y
(((a_length == $len) AND
(a_width == $wid)) AND
(a_depth == $dep))
))
THEN (
(MODIFY (OBJ: $x
(s_length $len)
(s_width $dep)
(s_depth 2)
) 1000 0.001)
)
COMMENT:"Rule to create the west cover for test 3")

(RULE: west_geometry 10
IF
(CLASS: Slab3 OBJ: $x
(((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
)
THEN (
(EXECUTE VAR: $rtn OBJ: $x create_geometry(16,"slabgeom2"))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(90.0,0.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(0.0,50.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the west ")
```

---

## D.2 TUTORIAL 3 - BOX APPLICATION

---

### create\_north.rul

---

```
(RULE: createnorth 1000
IF
((CLASS: Slab3 OBJ: $x
((s_length == 0) AND
(s_width == 0))
)
AND
(CLASS: Assembly OBJ: $y
(((a_length == $len) AND
(a_width == $wid)) AND
(a_depth == $dep))
))
THEN (
(MODIFY (OBJ:$x
(s_length $dep)
(s_width $wid)
(s_depth 2)
) 1000 0.001)
)
COMMENT:"Rule to create the north cover for test 3")

(RULE: north_geometry 10
IF
(CLASS: Slab3 OBJ: $x
(((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
)
THEN (
(EXECUTE VAR: $rtn OBJ: $x create_geometry(16,"slabgeom2"))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(100.0,0.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the north ")
```

---

## D.2 TUTORIAL 3 - BOX APPLICATION

---

### create\_south.rul

---

```
(RULE: createsouth 1000
IF
((CLASS: Slab3 OBJ: $x
((s_length == 0) AND
(s_width == 0))
)
AND
(CLASS: Assembly OBJ: $y
(((a_length == $len) AND
(a_width == $wid)) AND
(a_depth == $dep))
))
THEN (
(MODIFY (OBJ:$x
(s_length $dep)
(s_width $wid)
(s_depth 2)
) 1000 0.001)
)
COMMENT:"Rule to create the south cover for test 3")

(RULE: south_geometry 10
IF
(CLASS: Slab3 OBJ: $x
(((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
)
THEN (
(EXECUTE VAR: $rtn OBJ: $x create_geometry(16,"slabgeom2"))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(0.0,0.0,0.0))
(EXECUTE VAR: $rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the south ")
```



## D.2 TUTORIAL 3 - BOX APPLICATION

---

### D.2.2 Alternative - tut3\_long.rul

---

```
(RULE: createass 1000
IF
(CLASS: Assembly3 OBJ: $x
((a_length == 0) AND
(a_width == 0))
)
THEN (
(MODIFY (OBJ:$x
(a_length 1000)
(a_width 500)
(a_depth 300)
) 1000 0.001)
)
COMMENT:"Rule to create a simple assembly box for test 3")

(RULE: createslabs 1000
IF
(CLASS: Assembly3 OBJ: $y
(((a_length == $len) AND
(a_width == $wid)) AND
(a_depth == $dep))
)
THEN (
(MAKE (CLASS: Slab OBJ: top
(s_length $len)
(s_width $wid)
(s_depth 2)
))
(MAKE (CLASS: Slab OBJ: bottom
(s_length $len)
(s_width $wid)
(s_depth 2)
))
(MAKE (CLASS: Slab OBJ: north
(s_length $dep)
(s_width $wid)
(s_depth 2)
))
(MAKE (CLASS: Slab OBJ: south
(s_length $dep)
(s_width $wid)
(s_depth 2)
))
(MAKE (CLASS: Slab OBJ: east
(s_length $len)
(s_width $dep)
(s_depth 2)
))
(MAKE (CLASS: Slab OBJ: west
```

## D.2 TUTORIAL 3 - BOX APPLICATION

---

```
(s_length $len)
(s_width $dep)
(s_depth 2)
))
)
COMMENT:"Rule to create the covers for alternative tutorial")

(RULE: top_geometry 10
IF
((CLASS: Slab OBJ: $x
(instancename == "top")) AND
(CLASS: Slab OBJ: $x
((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(16,"slabgeomtop"))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(0.0,0.0,30.0))
(EXECUTE VAR:$rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the top ")

(RULE: bottom_geometry 10
IF
((CLASS: Slab OBJ: $x
(instancename == "bottom")) AND
(CLASS: Slab OBJ: $x
((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(16,"slabgeombottom"))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(0.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the bottom ")

(RULE: east_geometry 10
IF
((CLASS: Slab OBJ: $x
```

## D.2 TUTORIAL 3 - BOX APPLICATION

---

```
(instancename == "east")) AND
(CLASS: Slab OBJ: $x
((s_length == $len) AND
 (s_width == $wid)) AND
 (length != $len))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(16,"slabgeomeast"))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(90.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(0.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the east " )

(RULE: west_geometry 10
IF
((CLASS: Slab OBJ: $x
(instancename == "west")) AND
(CLASS: Slab OBJ: $x
((s_length == $len) AND
 (s_width == $wid)) AND
 (length != $len))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(16,"slabgeonwest"))
(MODIFY (OBJ: $x
(plane 2)
(length $len)
(width $wid)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(90.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(0.0,50.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the west " )

(RULE: north_geometry 10
IF
((CLASS: Slab OBJ: $x
(instancename == "north")) AND
(CLASS: Slab OBJ: $x
((s_length == $len) AND
 (s_width == $wid)) AND
 (length != $len))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(16,"slabgeomnorth"))
(MODIFY (OBJ: $x
(plane 2)
```

## D.2 TUTORIAL 3 - BOX APPLICATION

---

```
(length $len)
(width $wid)
) 1000 0.001) 160
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(100.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the north ")

(RULE: south_geometry 10
IF
((CLASS: Slab OBJ: $x
(instancename == "south")) AND 170
(CLASS: Slab OBJ: $x
((s_length == $len) AND
(s_width == $wid)) AND
(length != $len))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(16,"slabgeomsouth"))
(MODIFY (OBJ: $x
(plane 2)
(length $len) 180
(width $wid)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(0.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x show_geometry())
)
COMMENT:" Rule to set up the geometry of the south ")
```

---

---

## Appendix E

# CABIN DESIGN elements

### E.1 Cabin Artifact Vocabulary

#### E.1.1 Structural Member Superclass

The following is the base class description of all the structural engineering classes. The attributes contains all the information needed to analyze the structural feasibility of the whole assembly.

```
dbclass Struct_mbr
{
public:
  dbint concrete_grade;
  dbint reinforcement_grade;
  dbint dead_load;
  dbint live_load;
  dbint earthquake_load;
  dbint wind_load;
  PStringList parent_names;
  dbfloat X_coordinate;
  dbfloat Y_coordinate;
  dbfloat Z_coordinate;
  dbfloat X_rotation;
  dbfloat Y_rotation;
  dbfloat Z_rotation;
};
```

### Description of Basic Structural Member Attributes

- *Struct\_mbr.concrete\_grade.* Concrete compressive cylinder stress in newton/sq. mm - number.
- *Struct\_mbr.reinforcement\_grade.* Steel yield stress in newton/sq. mm.
- *Struct\_mbr.dead\_load.* Applied permanent dead loads including member's weight in newton/sq. mm - number.
- *Struct\_mbr.live\_load.* Load on a structure or member produced by the external environment and intended occupancy or use in newton/sq. mm - number.
- *Struct\_mbr.earthquake\_load.* Earthquake equivalent lateral load in newton/sq. mm - number.
- *Struct\_mbr.wind\_load.* Wind equivalent lateral load in newton/sq. mm - number.
- *Struct\_mbr.parent\_names.* The list of the parents which this member belongs to.
- *Struct\_mbr.X\_coordinate.* X coordinate of member's local reference frame origin with respect to the building reference frame in mm - number.
- *Struct\_mbr.Y\_coordinate.* Y coordinate of member's local reference frame origin with respect to the building reference frame in mm - number.
- *Struct\_mbr.Z\_coordinate.* Z coordinate of member's local reference frame origin with respect to the building reference frame in mm - number.
- *Struct\_mbr.X\_rotation.* Rotation of member's local reference frame about X axis in degrees - number.
- *Struct\_mbr.Y\_rotation.* Rotation of member's local reference frame about Y axis in degrees - number.
- *Struct\_mbr.Z\_rotation.* Rotation of member's local reference frame about Z axis in degrees - number.

#### E.1.2 Joint

The class Joint corresponds to the node information in the structural analysis process such as the connection type etc.

## E.1 Cabin Artifact Vocabulary

---

```
dbclass Joint : public Struct_mbr
{
  public:
    dbfloat point_loading;
    PString connection_type;
    PStringList parts_attached;
};
```

### Description of Basic Joint Attributes

- *Joint.point\_loading*. The member's point force magnitude applied to the area in Newton / sq.mm - number.
- *Joint.connection\_type*. The member's connection type at this joint.
- *Joint.parts\_attached*. The list of other members attached to this Joint.

### E.1.3 Linear Members

The class Linear Member corresponds to the structural members that we regard as having a linear load such as columns, piers, beams, truss members etc.

```
dbclass Linear_mbr: public Struct_mbr
{
  public:
    PString start_joint;
    PString end_joint;
    dbfloat axial_loading;
};
```

### Description of Linear Members Attributes

- *Linear\_mbr.start\_joint*. The member's joint name at start (lower 3D point).
- *Linear\_mbr.end\_joint*. The member's joint name at end (higher 3D point).
- *Linear\_mbr.axial\_loading*. The member's linear force magnitude applied linearly in Newton / sq.mm - number.

## E.1 Cabin Artifact Vocabulary

---

### E.1.4 Area Members

The class Area Member corresponds to the structural members that we regard as having an area load such as walls, footings, slabs, etc.

```
dbclass Area_mbr: public Struct_mbr
{
  public:
    PStringList corner_joints;
    dbfloat area_loading;
};
```

#### Description of Area Members Attributes

- *Area\_mbr.corner\_joints*. The list of member's corner joint names.
- *Area\_mbr.area\_loading*. The member's linear force magnitude applied to the area in Newton / sq.mm - number.

### E.1.5 Beams

```
dbclass Beam : public Linear_mbr // Generic Beam
{
  public:
    dbfloat b_length;
    dbfloat b_width;
    dbfloat b_depth;
    dbfloat b_effective_depth;
    dbfloat b_moment;
    dbfloat b_reinforcement_area;
    PString b_material;
};
```

#### Description of Attributes

- *Beam.b\_length*. Distance between centers of adjacent supporters along the X axis of beam local reference frame in mm - number.



## E.1 Cabin Artifact Vocabulary

---

- *Beam.b\_width*. Dimension of beam cross section along the Y axis of beam local reference frame in mm - number.
- *Beam.b\_depth*. Overall depth of the beam cross section along the Z axis of beam local reference frame in mm - number.
- *Beam.b\_effective\_depth*. Depth to the centerline of the tension steel along the Z axis of beam local reference frame in mm - number.
- *Beam.b\_moment*. Capacity (resisting moment) of the RC beam in newton mm - number.
- *Beam.b\_reinforcement\_area*. Area of ribbed tension steel reinforcement in sq. mm for RC beam - number.
- *Beam.b\_material*. Beam material (RC, Steel, etc) - text.

### E.1.6 Columns

```
dbclass Column : public Linear_mbr // Generic column
{
public:
  dbfloat c_width;
  dbfloat c_length;
  dbfloat c_depth;
  dbfloat c_effective_length;
  dbfloat c_bar_diameter;
  dbfloat c_link_diameter;
  dbfloat c_moment;
  dbfloat c_steel_area;
  dbfloat c_concrete_area;
  PString c_material;
};
```

#### Description of Attributes

- *Column.c\_width*. Dimension of column cross section along Y axis of column reference frame in mm - number.
- *Column.c\_depth*. Dimension of column cross section along the Z axis of column local reference frame in mm - number.
- *Column.c\_length*. Distance between the ends of column along the X axis of column local reference frame in mm - number.

## E.1 Cabin Artifact Vocabulary

---

- *Column.c\_effective\_length.* Buckling length along X axis of column local reference frame in mm - number.
- *Column.c\_bar\_diameter.* Diameter of column reinforcement bar within in mm - number.
- *Column.c\_link\_diameter.* Diameter of column reinforcement lateral tie in mm - number.
- *Column.c\_moment.* Moment applied to the center of the column in newton - number.
- *Column.c\_steel\_area.* RC column total steel area in sq. mm - number.
- *Column.c\_concrete\_area.* RC column total concrete area in sq. mm - number.
- *Column.c\_material.* Material of column - text.

### E.1.7 Piers

```
dbclass Pier : public Linear_mbr // Generic Pier
{
public:
  dbfloat p_width;
  dbfloat p_length;
  dbfloat p_depth;
  dbfloat p_effective_length;
  dbfloat p_moment;
  dbfloat p_steel_area;
  dbfloat p_concrete_area;
  PString p_material;
};
```

#### Description of Attributes

- *Pier.p\_width.* Dimension of Pier cross section along Y axis of Pier reference frame in mm - number.
- *Pier.p\_depth.* Dimension of Pier cross section along the Z axis of Pier local reference frame in mm - number.
- *Pier.p\_length.* Distance between the ends of Pier along the X axis of Pier local reference frame in mm - number.

## E.1 Cabin Artifact Vocabulary

---

- *Pier.p-effective-length*. Buckling length along X axis of Pier local reference frame in mm - number.
- *Pier.p-moment*. Moment applied to the center of the Pier in newton - number.
- *Pier.p-steel-area*. RC Pier total steel area in sq. mm - number.
- *Pier.p-concrete-area*. RC Pier total concrete area in sq. mm - number.
- *Pier.p-material*. Material of Pier - text.

### E.1.8 Truss\_member

```
dbclass Truss_member : public Linear_mbr
{
public:
  dbfloat t_axial_load;
  dbfloat t_length;
  dbfloat t_angle_leg;
  dbfloat t_thickness;
  dbfloat t_width;
  dbfloat t_cross_section_area;
  PString t_material;
};
```

#### Description of Attributes

- *Truss\_member.t-axial-load*. Axial load applied on truss member in newton - number.
- *Truss\_member.t-length*. Dimension of truss member along X axis of member local reference frame in mm - number.
- *Truss\_member.t-angle-leg*. Angle leg section of truss member in degrees - number.
- *Truss\_member.t-thickness*. Dimension of truss member along Z axis of member local reference frame in mm - number.
- *Truss\_member.t-width*. Dimension of truss member along Y axis of member local reference frame in mm - number.
- *Truss\_member.t-cross-section-area*. Cross sectional area of truss member in sq. mm - number.
- *Truss\_member.t-material*. The material of the Truss member.

## E.1 Cabin Artifact Vocabulary

---

### E.1.9 Truss\_system

```
dbclass Truss_system : public Area_mbr
{
public:
  dbfloat ts_pitch;
  dbfloat ts_truss_weight;
  dbfloat ts_purlin_weight;
  dbfloat ts_roofing_weight;
  dbfloat ts_total_height;
  dbfloat ts_total_length;
  PString ts_truss_type;
  PString ts_material;
  PStringList ts_memberlist;
};
```

#### Description of Attributes

- *Truss\_system.ts\_pitch*. The pitch angle of the Truss system.
- *Truss\_system.ts\_truss\_weight*. The total weight of the Truss system in newton - number.
- *Truss\_system.ts\_purlin\_weight*. The total weight of the purlins in newton - number.
- *Truss\_system.ts\_roofing\_weight*. The total weight of the roofings in newton - number.
- *Truss\_system.ts\_total\_height*. The total vertical dimension of truss system along Z axis of truss local reference frame - number.
- *Truss\_system.ts\_total\_length*. The total vertical dimension of truss system along X axis of truss local reference frame - number.
- *Truss\_system.ts\_truss\_type*. The type of the Truss system (e.g., gable-truss, ..).
- *Truss\_system.ts\_material*. The material of the Truss system.
- *Truss\_system.ts\_memberlist*. The list of all the members attached to this Truss system.

## E.1 Cabin Artifact Vocabulary

---

### E.1.10 Slabs

```
dbclass Slab : public Area_mbr // Generic Slab
{
public:
  dbfloat s_clear_xspan;
  dbfloat s_effective_xspan;
  dbfloat s_clear_yspan;
  dbfloat s_effective_yspan;
  dbfloat s_depth;
  dbfloat s_effective_depth;
  PString s_slabtype;
  dbfloat s_bending_moment;
  dbfloat s_reinforcement_area;
  PString s_material;
};
```

#### Description of Attributes

- *Slab.s\_clear\_xspan*. Distance between opposite faces of support along the X axis of slab local reference frame in mm - number.
- *Slab.s\_effective\_xspan*. Distance between center to center of beams supporting slab along the X axis of slab local reference frame in mm - number.
- *Slab.s\_clear\_yspan*. Distance between opposite faces of support along the Y axis of slab local reference frame in mm - number.
- *Slab.s\_effective\_yspan*. Distance between center to center of beams supporting slab along the Y axis of slab local reference frame in mm - number.
- *Slab.s\_depth*. Overall depth (thickness) of slab along the Z axis of slab local reference frame in mm - number.
- *Slab.s\_effective\_depth*. Depth to the centerline of the steel (Z axis direction) in mm - number.
- *Slab.s\_slabtype*. Type of the slab - one way or two way or flat slab.
- *Slab.s\_moment*. Moment of resistance of the slab in newton mm - number.
- *Slab.s\_reinforcement\_area*. Area of steel reinforcement in sq. mm - number.
- *Slab.s\_material*. Slab material (RC, Steel, etc).

### Ribbed\_slab

```
dbclass Ribbed_slab : public Slab // One way spanning ribbed slab
{
  public:
    dbfloat rib_width;
    dbfloat rib_depth;
    dbfloat topping;
};
```

#### Description of Attributes

- *Ribbed\_slab.rib\_width*. Breadth of the rib in mm - number.
- *Ribbed\_slab.rib\_depth*. Depth of the rib in mm - number.
- *Ribbed\_slab.topping*. Depth of the topping of the slab in mm - number.

### Waffle\_slab

```
dbclass Waffle_slab : public Slab // Two-way spanning ribbed slab
{
  public:
    dbfloat rib_width;
    dbfloat rib_depth;
    dbfloat rib_spacing;
    dbfloat topping;
};
```

#### Description of Attributes

- *Waffle\_slab.rib\_width*. Breadth of the rib in mm - number.
- *Waffle\_slab.rib\_depth*. Depth of the rib in mm - number.
- *Waffle\_slab.rib\_spacing*. Distance between the centers of the rib in mm - number.
- *Waffle\_slab.topping*. Thickness of slab in along z axis of slab local reference frame in mm - number.

## E.1 Cabin Artifact Vocabulary

---

### E.1.11 Walls

```
dbclass Wall : public Area_mbr // Generic Wall
{
public:
  PString w_type;
  dbfloat w_width;
  dbfloat w_height;
  dbfloat w_length;
  dbfloat w_axial_load;
  dbfloat w_reinforcement_area;
  dbfloat w_slenderness_ratio;
  PString w_material;
};
```

#### Description of Attributes

- *Wall.w\_type*. The type of the wall (brick or Shear or Opening).
- *Wall.w\_width*. Dimension of wall along Y axis of wall local reference frame in mm - number.
- *Wall.w\_height*. Dimension of wall along Z axis of wall local reference frame in mm - number.
- *Wall.w\_length*. Dimension of wall along X axis of wall local reference frame in mm - number.
- *Wall.w\_axial\_load*. Vertical concentrated load applied on wall along Z axis of wall local reference frame in newton - number.
- *Wall.w\_reinforcement\_area*. Area of steel reinforcement in sq mm - number.
- *Wall.w\_slenderness\_ratio*. Slenderness (height/thickness) ratio for wall - percentage.
- *Wall.w\_material*. Material of wall - text.

### E.1.12 Wall-opening

```
dbclass Wall_opening : public Area_mbr // Openings in the wall (door,etc.)
{
public:
```

## E.1 Cabin Artifact Vocabulary

---

```
dbfloat wo_length;  
dbfloat wo_width;  
dbfloat wo_depth;  
};
```

### Description of Attributes

- *Wall.opening.wo\_width.* Dimension of wall along Y axis of wall local reference frame in mm - number.
- *Wall.opening.wo\_depth.* Dimension of wall along Z axis of wall local reference frame in mm - number.
- *Wall.opening.wo\_length.* Dimension of wall along X axis of wall local reference frame in mm - number.

### E.1.13 Strip footing

```
dbclass Strip_footing : public Area_mbr // Wall foundation  
{  
public:  
dbfloat f_axial_load;  
dbfloat f_horizontal_load;  
dbfloat f_bearing_load;  
dbfloat f_moment_load;  
PString f_material;  
PString f_connection;  
dbfloat f_base_width;  
dbfloat f_base_depth;  
dbfloat f_base_length;  
dbfloat f_pier_width;  
dbfloat f_pier_length;  
dbfloat f_pier_depth;  
};
```

### Description of Attributes

- *Strip\_footing.f\_base\_width.* Dimension of foundation along Y axis of foundation of local reference frame in mm - number.



## E.1 Cabin Artifact Vocabulary

---

- *Strip footing.f\_base\_length.* Dimension of foundation along X axis of foundation of local reference frame in mm - number.
- *Strip footing.f\_base\_depth.* Dimension of foundation along Z axis of foundation of local reference frame in mm - number.
- *Strip footing.f\_axial\_load.* Vertical concentrated load applied on foundation in newton - number.
- *Strip footing.f\_horizontal\_load.* Concentrated load applied along the x axis of foundation local reference frame in newton - number.
- *Strip footing.f\_bearing\_load.* Soil bearing capacity in vertical direction in newton/sq. mm - number
- *Strip footing.f\_moment\_load.* Moment applied to the foundation in newton - number.
- *Strip footing.f\_material.* Material of the Foundation element - text.
- *Strip footing.f\_connection.* Connection type from foundation to the member supported - text.
- *Strip footing.f\_pier\_width.* The width of the pier supporting foundation (Y axis direction) in mm - number.
- *Strip footing.f\_pier\_length.* The length of the pier supporting foundation (X axis direction) in mm - number.
- *Strip footing.f\_pier\_depth.* The depth of the pier supporting foundation (Z axis direction) in mm - number.

### E.1.14 Mat\_found

```
dbclass Mat_found : public Area_mbr // Mat Foundation
{
public:
    dbfloat f_base_width;
    dbfloat f_base_length;
    dbfloat f_base_depth;
    dbfloat f_pier_width;
    dbfloat f_pier_length;
    dbfloat f_pier_depth;
    dbfloat f_axial_load;
    dbfloat f_horizontal_load;
    dbfloat f_bearing_load;
    dbfloat f_moment_load;
```

## E.1 Cabin Artifact Vocabulary

---

```
PString f_material;  
PString f_connection;  
};
```

### Description of Attributes

- *Mat\_found.f-base-width.* Dimension of foundation along Y axis of foundation of local reference frame in mm - number.
- *Mat\_found.f-base-length.* Dimension of foundation along X axis of foundation of local reference frame in mm - number.
- *Mat\_found.f-base-depth.* Dimension of foundation along Z axis of foundation of local reference frame in mm - number.
- *Mat\_found.f-pier-width.* The width of the pier supporting Mat (Y axis direction) in mm - number.
- *Mat\_found.f-pier-length.* The length of the pier supporting Mat (X axis direction) in mm - number.
- *Mat\_found.f-pier-depth.* The depth of the pier supporting Mat (Z axis direction) in mm - number.
- *Mat\_found.f-axial-load.* Vertical concentrated load applied on foundation in newton - number.
- *Mat\_found.f-horizontal-load.* Concentrated load applied along the x axis of foundation local reference frame in newton - number.
- *Mat\_found.f-bearing-load.* Soil bearing capacity in vertical direction in newton/sq. mm - number
- *Mat\_found.f-moment-load.* Moment applied to the foundation in newton - number.
- *Mat\_found.f-material.* Material of the Foundation element - text.
- *Mat\_found.f-connection.* Connection type from foundation to the member supported - text.

#### E.1.15 Spread\_found

```
dbclass Spread_found : public Area_mbr // = Isolated Foundation  
{
```

## E.1 Cabin Artifact Vocabulary

---

```
public:
  dbfloat f_base_width;
  dbfloat f_base_length;
  dbfloat f_base_depth;
  dbfloat f_pier_width;
  dbfloat f_pier_length;
  dbfloat f_pier_depth;
  dbfloat f_axial_load;
  dbfloat f_horizontal_load;
  dbfloat f_bearing_load;
  dbfloat f_moment_load;
  PString f_material;
  PString f_connection;
};
```

### Description of Attributes

- *Spread\_found.f-base-width.* Dimension of foundation along Y axis of foundation of local reference frame in mm - number.
- *Spread\_found.f-base-length.* Dimension of foundation along X axis of foundation of local reference frame in mm - number.
- *Spread\_found.f-base-depth.* Dimension of foundation along Z axis of foundation of local reference frame in mm - number.
- *Spread\_found.f-pier-width.* The width of the pier supporting foundation (Y axis direction) in mm - number.
- *Spread\_found.f-pier-length.* The length of the pier supporting foundation (X axis direction) in mm - number.
- *Spread\_found.f-pier-depth.* The depth of the pier supporting foundation (Z axis direction) in mm - number.
- *Spread\_found.f-axial-load.* Vertical concentrated load applied on foundation in newton - number.
- *Spread\_found.f-horizontal-load.* Concentrated load applied along the x axis of foundation local reference frame in newton - number.
- *Spread\_found.f-bearing-load.* Soil bearing capacity in vertical direction in newton/sq. mm - number
- *Spread\_found.f-moment-load.* Moment applied to the foundation in newton - number.

## E.2 Examples of CABIN Rulefiles

---

- *Spread\_found.f.material.* Material of the Foundation element - text.
- *Spread\_found.f.connection.* Connection type from foundation to the member supported - text.

## E.2 Examples of CABIN Rulefiles

### E.2.1 cabin\_eff.rul

---

```
(RULE: createcabin 1000
IF
(CLASS: Cabin OBJ: $x
(ca_length == 0.0)
)
THEN (
(MODIFY (OBJ:$x
(ca_length 720.0)
(ca_width 360.0)
(ca_height 240.0)
(ca_roof_span 360.0)
(ca_roof_height 90.0)
(site_location "above groundwater")
(soil_condition "poor")
) 1000 0.001)
(MAKE (CLASS: Cabin_part OBJ: cabin_parts
(length 0.0)
))
(EXECUTE VAR: $rtn OBJ:$x make_part("partcontainer","Cabin_part","cabin_parts"))
)
COMMENT:"Rule to create the cabin and site information & container of all parts")
```

---

## E.2 Examples of CABIN Rulefiles

---

### E.2.2 set\_columns.rul

---

```
(RULE: createfourcolumns 1000
IF
((CLASS: Cabin OBJ: $x
(ca_height == $hgt) AND
(numcolumns == 4)
) AND
(CLASS: Cabin_part OBJ: $y
(instancename == "cabin_parts")
))
THEN (
(MAKE (CLASS: Column OBJ: ne_column
(c_length $hgt)
(c_width 10.0)
(c_depth 10.0)
(X_coordinate 365.0)
(Y_coordinate 725.0)
(Z_coordinate 10.0)
(X_rotation 0.0)
(Y_rotation 270.0)
(Z_rotation 0.0)
))
(MAKE (CLASS: Column OBJ: nw_column
(c_length $hgt)
(c_width 10.0)
(c_depth 10.0)
(X_coordinate 5.0)
(Y_coordinate 725.0)
(Z_coordinate 10.0)
(X_rotation 0.0)
(Y_rotation 270.0)
(Z_rotation 0.0)
))
(MAKE (CLASS: Column OBJ: se_column
(c_length $hgt)
(c_width 10.0)
(c_depth 10.0)
(X_coordinate 365.0)
(Y_coordinate 5.0)
(Z_coordinate 10.0)
(X_rotation 0.0)
(Y_rotation 270.0)
(Z_rotation 0.0)
))
(MAKE (CLASS: Column OBJ: sw_column
(c_length $hgt)
(c_width 10.0)
(c_depth 10.0)
(X_coordinate 5.0)
(Y_coordinate 5.0)

```

## E.2 Examples of CABIN Rulefiles

---

```
(Z_coordinate 10.0) 50
(X_rotation 0.0)
(Y_rotation 270.0)
(Z_rotation 0.0)
))
(EXECUTE VAR:$rtn OBJ: $y make_part("northeastcol","Column","ne_column"))
(EXECUTE VAR:$rtn OBJ: $y make_part("northwestcol","Column","nw_column"))
(EXECUTE VAR:$rtn OBJ: $y make_part("southeastcol","Column","se_column"))
(EXECUTE VAR:$rtn OBJ: $y make_part("southwestcol","Column","sw_column"))
)
COMMENT:"Rule to create the four columns" 60

(RULE: createsixcolumns 1000
IF
((CLASS: Cabin OBJ: $x
((ca_height == $hgt) AND
(numcolumns == 6))
) AND
(CLASS: Cabin_part OBJ: $y
(instancename == "cabin_parts"))
)) 70
THEN (
(MAKE (CLASS: Column OBJ: ne_column
(c_length $hgt)
(c_width 10.0)
(c_depth 10.0)
(X_coordinate 365.0)
(Y_coordinate 725.0)
(Z_coordinate 10.0)
(X_rotation 0.0)
(Y_rotation 270.0) 80
(Z_rotation 0.0)
))
(MAKE (CLASS: Column OBJ: nw_column
(c_length $hgt)
(c_width 10.0)
(c_depth 10.0)
(X_coordinate 5.0)
(Y_coordinate 725.0)
(Z_coordinate 10.0)
(X_rotation 0.0) 90
(Y_rotation 270.0)
(Z_rotation 0.0)
))
(MAKE (CLASS: Column OBJ: se_column
(c_length $hgt)
(c_width 10.0)
(c_depth 10.0)
(X_coordinate 365.0)
(Y_coordinate 5.0)
(Z_coordinate 10.0) 100
(X_rotation 0.0)
(Y_rotation 270.0)
(Z_rotation 0.0)
```

## E.2 Examples of CABIN Rulefiles

---

```
))
(MAKE (CLASS: Column OBJ: sw_column
(c_length $hgt)
(c_width 10.0)
(c_depth 10.0)
(X_coordinate 5.0)
(Y_coordinate 5.0)
(Z_coordinate 10.0)
(X_rotation 0.0)
(Y_rotation 270.0)
(Z_rotation 0.0)
))
(MAKE (CLASS: Column OBJ: e_column
(c_length $hgt)
(c_width 10.0)
(c_depth 10.0)
(X_coordinate 365.0)
(Y_coordinate 365.0)
(Z_coordinate 10.0)
(X_rotation 0.0)
(Y_rotation 270.0)
(Z_rotation 0.0)
))
(MAKE (CLASS: Column OBJ: w_column
(c_length $hgt)
(c_width 10.0)
(c_depth 10.0)
(X_coordinate 5.0)
(Y_coordinate 365.0)
(Z_coordinate 10.0)
(X_rotation 0.0)
(Y_rotation 270.0)
(Z_rotation 0.0)
))
(EXECUTE VAR:$rtn OBJ: $y make_part("northeastcol","Column","ne_column"))
(EXECUTE VAR:$rtn OBJ: $y make_part("northwestcol","Column","nw_column"))
(EXECUTE VAR:$rtn OBJ: $y make_part("southeastcol","Column","se_column"))
(EXECUTE VAR:$rtn OBJ: $y make_part("southwestcol","Column","sw_column"))
(EXECUTE VAR:$rtn OBJ: $y make_part("eastcol","Column","e_column"))
(EXECUTE VAR:$rtn OBJ: $y make_part("westcol","Column","w_column"))
)
COMMENT:"Rule to create the six columns")

(RULE: ne_geometry 10
IF
((CLASS: Column OBJ: $x
(instancename == "ne_column")) AND
(CLASS: Column OBJ: $x
(((c_length == $len) AND
(c_width == $wid)) AND
((length != $len) AND
(c_depth == $hgt))))
))
THEN (
```

## E.2 Examples of CABIN Rulefiles

---

```
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"negeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(365.0,725.0,10.0))
)
COMMENT:" Rule to set up the geometry of the ne_column"

(RULE: nw_geometry 10
IF
((CLASS: Column OBJ: $x
(instancename == "nw_column")) AND
(CLASS: Column OBJ: $x
(((c_length == $len) AND
(c_width == $wid)) AND
((length != $len) AND
(c_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"nwgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(5.0,725.0,10.0))
)
COMMENT:" Rule to set up the geometry of the nw_column ")

(RULE: se_geometry 10
IF
((CLASS: Column OBJ: $x
(instancename == "se_column")) AND
(CLASS: Column OBJ: $x
(((c_length == $len) AND
(c_width == $wid)) AND
((length != $len) AND
(c_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"segeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(365.0,5.0,10.0))
)

```



## E.2 Examples of CABIN Rulefiles

---

```
COMMENT:" Rule to set up the geometry of the se_column ")

(RULE: sw_geometry 10
IF
((CLASS: Column OBJ: $x
(instancename == "sw_column")) AND
(CLASS: Column OBJ: $x
(((c_length == $len) AND
(c_width == $wid)) AND
((length != $len) AND
(c_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"swgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(5.0,5.0,10.0))
)
COMMENT:" Rule to set up the geometry of the sw_column ")

(RULE: e_geometry 10
IF
((CLASS: Column OBJ: $x
(instancename == "e_column")) AND
(CLASS: Column OBJ: $x
(((c_length == $len) AND
(c_width == $wid)) AND
((length != $len) AND
(c_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"egeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(365.0,365.0,10.0))
)
COMMENT:" Rule to set up the geometry of the e_column ")

(RULE: w_geometry 10
IF
((CLASS: Column OBJ: $x
(instancename == "w_column")) AND
(CLASS: Column OBJ: $x
```

220

230

240

250

260

## E.2 Examples of CABIN Rulefiles

---

```
((c_length == $len) AND
 (c_width == $wid) AND
 ((length != $len) AND
 (c_depth == $hgt)))
))
270
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"wgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(5.0,365.0,10.0))
)
280
COMMENT:" Rule to set up the geometry of the w_column ")
```

---

## E.2 Examples of CABIN Rulefiles

---

### E.2.3 set\_girders.rul

---

```
(RULE: zero 20
IF
(CLASS: Declist OBJ: $y
((set_columns == "0") AND
 (set_girders != "0"))
)
THEN (
(MODIFY (OBJ: $y
(set_girders "0")
) 1000 0.001)
)
COMMENT:" If the number of columns is zero, then use no girders")

(RULE: two 20
IF
((CLASS: Cabin OBJ: $x
(numcolumns != 0))
AND
(CLASS: Declist OBJ: $y
(set_girders != "2"))
)
THEN (
(MODIFY (OBJ: $y
(set_girders "2")
) 1000 0.001)
)
COMMENT:" If the number of columns is more than zero, then can use 2 girders")

(RULE: zero_or_two 20
IF
((CLASS: Cabin OBJ: $x
(numcolumns == 6))
AND
(CLASS: Declist OBJ: $y
(set_girders != "0"))
)
THEN (
(MODIFY (OBJ: $y
(set_girders "0")
) 1000 0.001)
)
COMMENT:" If the number of columns is six, then can use zero girders")
```

---

## E.2 Examples of CABIN Rulefiles

---

### E.2.4 set\_girders\_eff.rul

---

```
(RULE: createtwogirders 1000
IF
((CLASS: Cabin OBJ: $x
((ca_length == $len) AND
(numgirders == 2))
) AND
(CLASS: Cabin_part OBJ: $y
(instancename == "cabin_parts")
))
THEN (
(MAKE (CLASS: Beam OBJ: e_girder
(b_length $len)
(b_width 5.0)
(b_depth 5.0)
))
(MAKE (CLASS: Beam OBJ: w_girder
(b_length $len)
(b_width 5.0)
(b_depth 5.0)
))
(EXECUTE VAR: $rtn OBJ: $y make_part("eastgirder","Beam","e_girder"))
(EXECUTE VAR: $rtn OBJ: $y make_part("westgirder","Beam","w_girder"))
)
COMMENT:"Rule to create the two girders")

(RULE: e_geometry 10
IF
((CLASS: Beam OBJ: $x
(instancename == "e_girder")) AND
(CLASS: Beam OBJ: $x
((b_length == $len) AND
(b_width == $wid) AND
((length != $len) AND
(b_depth == $hgt)))
))
THEN (
(EXECUTE VAR: $rtn OBJ: $x create_geometry(4,"egirdgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(0.0,0.0,90.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(365.0,5.0,250.0))
)
COMMENT:" Rule to set up the geometry of the e_girder")

(RULE: w_geometry 10
IF
```

## E.2 Examples of CABIN Rulefiles

---

```
((CLASS: Beam OBJ: $x
(instancename == "w_girder")) AND
(CLASS: Beam OBJ: $x
(((b_length == $len) AND
 (b_width == $wid)) AND
 (length != $len) AND
 (b_depth == $hgt)))
))
THEN (
(EXECUTE VAR: $rtn OBJ: $x create_geometry(4,"wgirdgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR: $rtn OBJ: $x set_rotation(0.0,0.0,90.0))
(EXECUTE VAR: $rtn OBJ: $x set_translation(5.0,5.0,250.0))
)
COMMENT:" Rule to set up the geometry of the w_girder")
```

---

## E.2 Examples of CABIN Rulefiles

---

### E.2.5 set\_south\_wall\_eff.rul

---

```
(RULE: createsouthshear 1000
IF
((CLASS: Cabin OBJ: $x
(((ca_width == $len) AND
(ca_height == $hgt)) AND
(southwalltype == "Shear"))
) AND
(CLASS: Cabin_part OBJ: $y
(instancename == "cabin_parts")
))
THEN (
(MAKE (CLASS: Wall OBJ: s_wall
(w_length $len)
(w_height $hgt)
))
(EXECUTE VAR:$rtn OBJ: $y make_part("southwall","Wall","s_wall"))
)
COMMENT:"Rule to create the south wall")
10

(RULE: createsouthbrick 1000
IF
((CLASS: Cabin OBJ: $x
(((ca_width == $len) AND
(ca_height == $hgt)) AND
(southwalltype == "Brick"))
) AND
(CLASS: Cabin_part OBJ: $y
(instancename == "cabin_parts")
))
THEN (
(MAKE (CLASS: Wall OBJ: s_wall
(w_length $len)
(w_height $hgt)
))
(EXECUTE VAR:$rtn OBJ: $y make_part("southwall","Wall","s_wall"))
)
COMMENT:"Rule to create the south brick wall")
20

(RULE: createsouthopening 1000
IF
((CLASS: Cabin OBJ: $x
(((ca_width == $len) AND
(ca_height == $hgt)) AND
(southwalltype == "Opening"))
) AND
(CLASS: Cabin_part OBJ: $y
(instancename == "cabin_parts")
))
THEN (
30
40
```

## E.2 Examples of CABIN Rulefiles

---

```
(MAKE (CLASS: Wall OBJ: s_wall                                     50
(w_length $len)
(w_height $hgt)
))
(EXECUTE VAR:$rtn OBJ: $y make_part("southwall","Wall","s_wall"))
)
COMMENT:"Rule to create the south Opening wall")

(RULE: attachsouthopening 1000
IF                                                                    60
(((CLASS: Wall OBJ: $x
(instancename == "s_wall")
) AND
(CLASS: Cabin OBJ: $y
(southwalltype == "Opening")
)) AND
(CLASS: Cabin_part OBJ: $z
(instancename == "cabin_parts")
))
THEN (                                                                    70
(MAKE (CLASS: Wall_opening OBJ: s_opening
(wo_length 215.0)
(wo_width 95.0)
(wo_depth 5.0)
))
(EXECUTE VAR:$rtn OBJ: $x make_part("southopening","Wall_opening","s_opening"))
)
COMMENT:"Rule to create the south opening and attach it to the wall")

(RULE: south_geometry 100
IF
(CLASS: Wall OBJ: $x
(((w_length == $len) AND
(w_height == $wid)) AND
((length != $len) AND
(instancename == "s_wall")))
)
THEN (                                                                    90
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"wallsgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height 5.0)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(90.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(5.0,5.0,10.0))
)
COMMENT:" Rule to set up the geometry of the south ")

(RULE: south_opening_geometry 10
IF
(CLASS: Wall_opening OBJ: $x
```

100

## E.2 Examples of CABIN Rulefiles

---

```
((instancename == "s_opening") AND
  (length != 215.0))
)
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(16,"wallsopengeom"))
(MODIFY (OBJ: $x
(length 215.0)
(width 100.0)
(height 5.0)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,270.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(135.0,3.0,10.0))
)
COMMENT:" Rule to set up the geometry of the south ")
```

110



## E.2 Examples of CABIN Rulefiles

---

### E.2.6 set\_foundation\_eff.rul

---

```
(RULE: creatematfour 1000
IF
(((CLASS: Cabin OBJ: $x
((ca_length == $len) AND
(ca_width == $wid))
) AND
(CLASS: Declist OBJ: $y
((set_foundation == "Mat") AND
(set_columns != "6"))
)) AND
(CLASS: Cabin_part OBJ: $z
(instancename == "cabin_parts")
))
THEN (
(MAKE (CLASS: Mat_found OBJ: mat_pier_ne
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_ne","Mat_found","mat_pier_ne"))
(MAKE (CLASS: Mat_found OBJ: mat_pier_nw
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_nw","Mat_found","mat_pier_nw"))
(MAKE (CLASS: Mat_found OBJ: mat_pier_se
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_se","Mat_found","mat_pier_se"))
(MAKE (CLASS: Mat_found OBJ: mat_pier_sw
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
))
```

## E.2 Examples of CABIN Rulefiles

---

```
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_sw","Mat_found","mat_pier_sw")) 50
(MAKE (CLASS: Mat_found OBJ: mat_base
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_floor","Mat_found","mat_base"))
) 60
COMMENT:"Rule to create the mat foundation with four columns")

(RULE: creatematsix 1000
IF
(((CLASS: Cabin OBJ: $x
((ca_length == $len) AND
(ca_width == $wid))
) AND
(CLASS: Declist OBJ: $y
((set_foundation == "Mat") AND
(set_columns == "6"))
)) AND
(CLASS: Cabin_part OBJ: $z
(instancename == "cabin_parts")
))
THEN (
(MAKE (CLASS: Mat_found OBJ: mat_pier_ne
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
)) 80
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_ne","Mat_found","mat_pier_ne"))
(MAKE (CLASS: Mat_found OBJ: mat_pier_nw
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
)) 90
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_nw","Mat_found","mat_pier_nw"))
(MAKE (CLASS: Mat_found OBJ: mat_pier_se
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
)) 100
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_se","Mat_found","mat_pier_se"))
```

## E.2 Examples of CABIN Rulefiles

---

```
(MAKE (CLASS: Mat_found OBJ: mat_pier_sw
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_sw","Mat_found","mat_pier_sw"))
(MAKE (CLASS: Mat_found OBJ: mat_pier_e
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_e","Mat_found","mat_pier_e"))
(MAKE (CLASS: Mat_found OBJ: mat_pier_w
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_w","Mat_found","mat_pier_w"))
(MAKE (CLASS: Mat_found OBJ: mat_base
(f_base_length $len)
(f_base_width $wid)
(f_base_depth 5.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_floor","Mat_found","mat_base"))
)
COMMENT:"Rule to create the mat foundation with four columns")

(RULE: createspreadfour 1000
IF
(((CLASS: Cabin OBJ: $x
((ca_length == $len) AND
(ca_width == $wid))
) AND
(CLASS: Declist OBJ: $y
((set_foundation == "Spread") AND
(set_columns != "6"))
)) AND
(CLASS: Cabin_part OBJ: $z
(instancename == "cabin_parts")
))
THEN (
(MAKE (CLASS: Spread_found OBJ: spread_pier_ne
```



## E.2 Examples of CABIN Rulefiles

---

```
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pad_se","Spread_found","spread_pad_se"))
(MAKE (CLASS: Spread_found OBJ: spread_pad_sw 220
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pad_sw","Spread_found","spread_pad_sw"))
)
COMMENT:"Rule to create the spread foundation with four columns" 230

(RULE: createspreadsix 1000
IF
(((CLASS: Cabin OBJ: $x
((ca_length == $len) AND
 (ca_width == $wid))
) AND
(CLASS: Declist OBJ: $y
((set_foundation == "Spread") AND
 (set_columns == "6"))
)) AND
(CLASS: Cabin_part OBJ: $z
(instancename == "cabin_parts")
))
THEN (
(MAKE (CLASS: Spread_found OBJ: spread_pier_ne
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0) 250
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_ne","Spread_found","spread_pier_ne"))
(MAKE (CLASS: Spread_found OBJ: spread_pier_nw
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0) 260
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_nw","Spread_found","spread_pier_nw"))
(MAKE (CLASS: Spread_found OBJ: spread_pier_se
(f_base_length 5.0)
```

## E.2 Examples of CABIN Rulefiles

---

```
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_se","Spread_found","spread_pier_se"))
(MAKE (CLASS: Spread_found OBJ: spread_pier_sw
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_sw","Spread_found","spread_pier_sw"))
(MAKE (CLASS: Spread_found OBJ: spread_pier_e
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_e","Spread_found","spread_pier_e")) 290
(MAKE (CLASS: Spread_found OBJ: spread_pier_w
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pier_w","Spread_found","spread_pier_w"))
(MAKE (CLASS: Spread_found OBJ: spread_pad_ne 300
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pad_ne","Spread_found","spread_pad_ne"))
(MAKE (CLASS: Spread_found OBJ: spread_pad_nw 310
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pad_nw","Spread_found","spread_pad_nw"))
(MAKE (CLASS: Spread_found OBJ: spread_pad_se
(f_base_length 5.0)
```

## E.2 Examples of CABIN Rulefiles

---

```
(f_base_width 10.0) 320
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pad_se","Spread_found","spread_pad_se"))
(MAKE (CLASS: Spread_found OBJ: spread_pad_sw
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0) 330
(f_pier_length 5.0)
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pad_sw","Spread_found","spread_pad_sw"))
(MAKE (CLASS: Spread_found OBJ: spread_pad_e
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0) 340
(f_pier_width 2.0)
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pad_e","Spread_found","spread_pad_e"))
(MAKE (CLASS: Spread_found OBJ: spread_pad_w
(f_base_length 5.0)
(f_base_width 10.0)
(f_base_depth 10.0)
(f_pier_length 5.0)
(f_pier_width 2.0) 350
(f_pier_depth 2.0)
))
(EXECUTE VAR:$rtn OBJ: $z make_part("foundation_pad_w","Spread_found","spread_pad_w"))
)
COMMENT:"Rule to create the spread foundation with four columns")

(RULE: createstrip 1000
IF
(((CLASS: Cabin OBJ: $x
((ca_length == $len) AND 360
(ca_width == $wid))
) AND
(CLASS: Declist OBJ: $y
(set_foundation == "Strip")
)) AND
(CLASS: Cabin_part OBJ: $z
(instancename == "cabin_parts")
))
THEN (
(MAKE (CLASS: Strip_footing OBJ: wall_pier_n 370
(f_base_depth 5.0)
(f_base_length $wid)
(f_base_width 15.0)
```

## E.2 Examples of CABIN Rulefiles

---

```
(f_pier_length $wid)
(f_pier_width 5.0)
(f_pier_depth 5.0)
))
(EXECUTE VAR:$rtn OBJ:$z make_part("wall_pier_n","Strip_footing","wall_pier_n"))
(MAKE (CLASS: Strip_footing OBJ: wall_pier_s
(f_base_depth 5.0)
(f_base_length $wid)
(f_base_width 15.0)
(f_pier_length $wid)
(f_pier_width 5.0)
(f_pier_depth 5.0)
))
(EXECUTE VAR:$rtn OBJ:$z make_part("wall_pier_s","Strip_footing","wall_pier_s"))
(MAKE (CLASS: Strip_footing OBJ: wall_pier_e
(f_base_depth 5.0)
(f_base_length $len)
(f_base_width 15.0)
(f_pier_length $len)
(f_pier_width 5.0)
(f_pier_depth 5.0)
))
(EXECUTE VAR:$rtn OBJ:$z make_part("wall_pier_e","Strip_footing","wall_pier_e"))
(MAKE (CLASS: Strip_footing OBJ: wall_pier_w
(f_base_depth 5.0)
(f_base_length $len)
(f_base_width 15.0)
(f_pier_length $len)
(f_pier_width 5.0)
(f_pier_depth 5.0)
))
(EXECUTE VAR:$rtn OBJ:$z make_part("wall_pier_w","Strip_footing","wall_pier_w"))
(MAKE (CLASS: Strip_footing OBJ: wall_pad_n
(f_base_depth 5.0)
(f_base_length $wid)
(f_base_width 15.0)
(f_pier_length $wid)
(f_pier_width 5.0)
(f_pier_depth 5.0)
))
(EXECUTE VAR:$rtn OBJ:$z make_part("wall_pad_n","Strip_footing","wall_pad_n"))
(MAKE (CLASS: Strip_footing OBJ: wall_pad_s
(f_base_depth 5.0)
(f_base_length $wid)
(f_base_width 15.0)
(f_pier_length $wid)
(f_pier_width 5.0)
(f_pier_depth 5.0)
))
(EXECUTE VAR:$rtn OBJ:$z make_part("wall_pad_s","Strip_footing","wall_pad_s"))
(MAKE (CLASS: Strip_footing OBJ: wall_pad_e
(f_base_depth 5.0)
(f_base_length $len)
(f_base_width 15.0)
```

380

390

400

410

420



## E.2 Examples of CABIN Rulefiles

---

```
(f_pier_length $len)
(f_pier_width 5.0)
(f_pier_depth 5.0)
))
(EXECUTE VAR:$rtn OBJ:$z make_part("wall_pad_e","Strip footing","wall_pad_e"))
(MAKE (CLASS: Strip footing OBJ: wall_pad_w
(f_base_depth 5.0)
(f_base_length $len)
(f_base_width 15.0)
(f_pier_length $len)
(f_pier_width 5.0)
(f_pier_depth 5.0)
))
(EXECUTE VAR:$rtn OBJ:$z make_part("wall_pad_w","Strip footing","wall_pad_w"))
)
COMMENT:"Rule to create the Strip footing")

(RULE: Wall_n_pier_geometry 10
IF
((CLASS: Strip footing OBJ: $x
(instancename == "wall_pier_n")) AND
(CLASS: Strip footing OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
(length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"wallbgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(5.0,715.0,5.0))
)
COMMENT:" Rule to set up the geometry of the wall northeast Strip footing ")

(RULE: Wall_s_pier_geometry 10
IF
((CLASS: Strip footing OBJ: $x
(instancename == "wall_pier_s")) AND
(CLASS: Strip footing OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
(length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"wallbsgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)

```

## E.2 Examples of CABIN Rulefiles

---

```
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(5.0,5.0,5.0))
)
COMMENT:" Rule to set up the geometry of the wall northwest Strip_footing ")

(RULE: Wall_e_pier_geometry 10
IF
((CLASS: Strip_footing OBJ: $x
(instancename == "wall_pier_e")) AND
(CLASS: Strip_footing OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt))))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"wallbegeom"))
(MODIFY (OBJ: $x
(length $len)
(width 3.0)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,0.0,90.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(365.0,5.0,5.0))
)
COMMENT:" Rule to set up the geometry of the wall southeast Column ")

(RULE: Wall_w_pier_geometry 10
IF
((CLASS: Strip_footing OBJ: $x
(instancename == "wall_pier_w")) AND
(CLASS: Strip_footing OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt))))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"wallbwgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,0.0,90.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(10.0,5.0,5.0))
)
COMMENT:" Rule to set up the geometry of the wall southwest Column ")

(RULE: Wall_n_pad_geometry 10
```

## E.2 Examples of CABIN Rulefiles

---

```
IF
((CLASS: Strip_footing OBJ: $x
(instancename == "wall_pad_n")) AND
(CLASS: Strip_footing OBJ: $x
(((f_base_length == $len) AND
(f_base_width == $wid)) AND
(length != $len) AND
(f_base_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"pngeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(0.0,710.0,0.0))
)
COMMENT:" Rule to set up the geometry of the wall northeast pad "
```

```
(RULE: Wall_s_pad_geometry 10
IF
((CLASS: Strip_footing OBJ: $x
(instancename == "wall_pad_s")) AND
(CLASS: Strip_footing OBJ: $x
(((f_base_length == $len) AND
(f_base_width == $wid)) AND
(length != $len) AND
(f_base_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"psgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
)
COMMENT:" Rule to set up the geometry of the wall northwest pad "
```

```
(RULE: Wall_e_pad_geometry 10
IF
((CLASS: Strip_footing OBJ: $x
(instancename == "wall_pad_e")) AND
(CLASS: Strip_footing OBJ: $x
(((f_base_length == $len) AND
(f_base_width == $wid)) AND
(length != $len) AND
(f_base_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"pegeom"))
```

## E.2 Examples of CABIN Rulefiles

---

```
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,0.0,90.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(350.0,0.0,0.0))
)
COMMENT:" Rule to set up the geometry of the wall southeast pad ")
```

```
(RULE: Wall_w_pad_geometry 10
IF
((CLASS: Strip_footing OBJ: $x
(instancename == "wall_pad_w")) AND
(CLASS: Strip_footing OBJ: $x
(((f_base_length == $len) AND
(f_base_width == $wid)) AND
((length != $len) AND
(f_base_depth == $hgt))))
))
```

```
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"pwgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(15.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,0.0,90.0))
)
```

```
COMMENT:" Rule to set up the geometry of the wall southwest pad ")
```

```
(RULE: mat_ne_pier_geometry 10
IF
((CLASS: Mat_found OBJ: $x
(instancename == "mat_pier_ne")) AND
(CLASS: Mat_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt))))
))
```

```
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"negeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(366.0,724.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
)
```

## E.2 Examples of CABIN Rulefiles

---

```
COMMENT:" Rule to set up the geometry of the mat northeast pier ")

(RULE: mat_nw_pier_geometry 10
IF
((CLASS: Mat_found OBJ: $x
(instancename == "mat_pier_nw")) AND
(CLASS: Mat_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"nwgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(6.0,724.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
)
COMMENT:" Rule to set up the geometry of the mat northwest pier ")

(RULE: mat_se_pier_geometry 10
IF
((CLASS: Mat_found OBJ: $x
(instancename == "mat_pier_se")) AND
(CLASS: Mat_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"segeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(366.0,4.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
)
COMMENT:" Rule to set up the geometry of the mat southeast pier ")

(RULE: mat_sw_pier_geometry 10
IF
((CLASS: Mat_found OBJ: $x
(instancename == "mat_pier_sw")) AND
(CLASS: Mat_found OBJ: $x
(((f_pier_length == $len) AND
```

## E.2 Examples of CABIN Rulefiles

---

```
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"swgeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_translation(6.0,4.0,5.0))
)
COMMENT:" Rule to set up the geometry of the mat southwest pier ")
```

```
(RULE: mat_e_pier_geometry 10
IF
((CLASS: Mat_found OBJ: $x
(instancename == "mat_pier_e")) AND
(CLASS: Mat_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"egeom"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(366.0,364.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
)
COMMENT:" Rule to set up the geometry of the mat east pier ")
```

```
(RULE: mat_w_pier_geometry 10
IF
((CLASS: Mat_found OBJ: $x
(instancename == "mat_pier_w")) AND
(CLASS: Mat_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"wgeom"))
(MODIFY (OBJ: $x
(length $len)

```

## E.2 Examples of CABIN Rulefiles

---

```
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(6.0,364.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
)
COMMENT:" Rule to set up the geometry of the mat west pier ")

(RULE: mat_base_geometry 10                                     760
IF
((CLASS: Mat_found OBJ: $x
(instancename == "mat_base")) AND
(CLASS: Mat_found OBJ: $x
(((f_base_length == $len) AND
(f_base_width == $wid)) AND
((f_base_depth == $hgt) AND
(height != $hgt)))
))
THEN (                                                         770
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"basegeom"))
(MODIFY (OBJ: $x
(length $len+2)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(366.0,4.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,0.0,90.0))
)
COMMENT:" Rule to set up the geometry of the mat base"         780

(RULE: Spread_ne_pier_geometry 10
IF
((CLASS: Spread_found OBJ: $x
(instancename == "spread_pier_ne")) AND
(CLASS: Spread_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"negeompier"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(366.0,724.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))         800
)
COMMENT:" Rule to set up the geometry of the spr northeast pier ")

(RULE: Spread_nw_pier_geometry 10
```

## E.2 Examples of CABIN Rulefiles

---

```
IF
((CLASS: Spread_found OBJ: $x
(instancename == "spread_pier_nw")) AND
(CLASS: Spread_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
(length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"nwgeompier"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(6.0,724.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
)
COMMENT:" Rule to set up the geometry of the spr northwest pier ")
```

```
(RULE: Spread_se_pier_geometry 10
IF
((CLASS: Spread_found OBJ: $x
(instancename == "spread_pier_se")) AND
(CLASS: Spread_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
(length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"segeompier"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(366.0,4.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
)
COMMENT:" Rule to set up the geometry of the spr southeast pier ")
```

```
(RULE: Spread_sw_pier_geometry 10
IF
((CLASS: Spread_found OBJ: $x
(instancename == "spread_pier_sw")) AND
(CLASS: Spread_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
(length != $len) AND
(f_pier_depth == $hgt)))
))
```



## E.2 Examples of CABIN Rulefiles

---

```
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"swgeompier"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(6.0,4.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
)
COMMENT:" Rule to set up the geometry of the spr southwest pier ")
```

```
(RULE: Spread_e_pier_geometry 10
IF
((CLASS: Spread_found OBJ: $x
(instancename == "spread_pier_e")) AND
(CLASS: Spread_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"egeompier"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(366.0,364.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
)
COMMENT:" Rule to set up the geometry of the spr east pier ")
```

```
(RULE: Spread_w_pier_geometry 10
IF
((CLASS: Spread_found OBJ: $x
(instancename == "spread_pier_w")) AND
(CLASS: Spread_found OBJ: $x
(((f_pier_length == $len) AND
(f_pier_width == $wid)) AND
((length != $len) AND
(f_pier_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"wgeompier"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
```

## E.2 Examples of CABIN Rulefiles

---

```
(EXECUTE VAR:$rtn OBJ: $x set_translation(6.0,364.0,5.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,270.0,0.0))
)
COMMENT:" Rule to set up the geometry of the spr west pier "
```

```
(RULE: Spread_ne_pad_geometry 10                                     920
IF
((CLASS: Spread_found OBJ: $x
(instancename == "spread_pad_ne")) AND
(CLASS: Spread_found OBJ: $x
(((f_base_length == $len) AND
(f_base_width == $wid)) AND
((length != $len) AND
(f_base_depth == $hgt)))
))
THEN (                                                                930
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"negeompad"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(360.0,720.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,90.0,0.0))
)
COMMENT:" Rule to set up the geometry of the spr northeast pad " ) 940
```

```
(RULE: Spread_nw_pad_geometry 10
IF
((CLASS: Spread_found OBJ: $x
(instancename == "spread_pad_nw")) AND
(CLASS: Spread_found OBJ: $x
(((f_base_length == $len) AND
(f_base_width == $wid)) AND
((length != $len) AND
(f_base_depth == $hgt)))
))
THEN (                                                                950
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"nwgeompad"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(0.0,720.0,0.0))          960
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,90.0,0.0))
)
COMMENT:" Rule to set up the geometry of the spr northwest pad "
```

```
(RULE: Spread_se_pad_geometry 10
IF
```

## E.2 Examples of CABIN Rulefiles

---

```
((CLASS: Spread_found OBJ: $x
(instancename == "spread_pad_se")) AND
(CLASS: Spread_found OBJ: $x
(((f_base_length == $len) AND
(f_base_width == $wid)) AND
((length != $len) AND
(f_base_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"segeompad"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_translation(360.0,0.0,0.0))
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,90.0,0.0))
)
COMMENT:" Rule to set up the geometry of the spr southeast pad ")

(RULE: Spread_sw_pad_geometry 10
IF
(CLASS: Spread_found OBJ: $x
(instancename == "spread_pad_sw")) AND
(CLASS: Spread_found OBJ: $x
(((f_base_length == $len) AND
(f_base_width == $wid)) AND
((length != $len) AND
(f_base_depth == $hgt)))
))
THEN (
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"swgeompad"))
(MODIFY (OBJ: $x
(length $len)
(width $wid)
(height $hgt)
) 1000 0.001)
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,90.0,0.0))
)
COMMENT:" Rule to set up the geometry of the spr southwest pad ")

(RULE: Spread_e_pad_geometry 10
IF
(CLASS: Spread_found OBJ: $x
(instancename == "spread_pad_e")) AND
(CLASS: Spread_found OBJ: $x
(((f_base_length == $len) AND
(f_base_width == $wid)) AND
((length != $len) AND
(f_base_depth == $hgt)))
))
```

970

980

990

1000

1010

1020

## E.2 Examples of CABIN Rulefiles

---

```
THEN (  
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"egeompad"))  
(MODIFY (OBJ: $x  
(length $len)  
(width $wid)  
(height $hgt)  
) 1000 0.001)  
(EXECUTE VAR:$rtn OBJ: $x set_translation(360.0,360.0,0.0))  
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,90.0,0.0)) 1030  
)  
COMMENT:" Rule to set up the geometry of the spr east pad ")
```

```
(RULE: Spread_w_pad_geometry 10  
IF  
((CLASS: Spread_found OBJ: $x  
(instancename == "spread_pad_w")) AND  
(CLASS: Spread_found OBJ: $x  
(((f_base_length == $len) AND 1040  
(f_base_width == $wid)) AND  
(length != $len) AND  
(f_base_depth == $hgt)))  
))  
THEN (  
(EXECUTE VAR:$rtn OBJ: $x create_geometry(7,"wgeompad"))  
(MODIFY (OBJ: $x  
(length $len)  
(width $wid)  
(height $hgt) 1050  
) 1000 0.001)  
(EXECUTE VAR:$rtn OBJ: $x set_translation(0.0,360.0,0.0))  
(EXECUTE VAR:$rtn OBJ: $x set_rotation(0.0,90.0,0.0))  
)  
COMMENT:" Rule to set up the geometry of the spr west pad ")
```

1060

### E.3 CABIN calculation rulefiles

---

#### E.3 CABIN calculation rulefiles

All the rulefiles listed below are taken from the analysis of the reinforced concrete structures.

##### E.3.1 Loading.rul

This rulefile passes the loading of the structure from top to bottom.

---

```
(RULE: createclasses 1000
IF
(CLASS: roof OBJ: $x
(deadload == 0.0)
)
THEN (
(MODIFY (OBJ: $x
(deadload 100.1)
(liveload 200.2)
(windload 100.1)
) 1000 0.001)
(MAKE (CLASS: column OBJ: newcolumn
(deadload 10.1)
(liveload 20.2)
(windload 10.1)
))
(MAKE (CLASS: footing OBJ: newfooting
(deadload 1000.1)
(liveload 2000.2)
(windload 1000.1)
))
)
COMMENT:"Rule to create the source")

(RULE: calculateroof 1000
IF
(CLASS: roof OBJ: $x
(((deadload == $dl) AND
(liveload == $ll)) AND
((windload == $wl) AND
(totalload == 0.0))
))
THEN (
(BIND VAR:$dl $dl*1.4)
(BIND VAR:$ll $ll*1.7)
(MODIFY (OBJ: $x
(totalload $dl+$ll)
) 1000 0.001)
)
)
```

### E.3 CABIN calculation rulefiles

---

```
COMMENT:"Rule to calculate roof totalload" 40

(RULE: calculatecolumn 1000
IF
(CLASS: column OBJ: $x
(((deadload == $dl) AND
(liveload == $ll)) AND
((windload == $wl) AND
(totalload == 0.0))
))
THEN ( 50
(MODIFY (OBJ: $x
(totalload $dl+$ll+$wl)
) 1000 0.001)
)
COMMENT:"Rule to calculate column totalload")

(RULE: calculatefooting 1000
IF
(CLASS: footing OBJ: $x
(((deadload == $dl) AND 60
(liveload == $ll)) AND
((earthquakeload == $eql) AND
(totalload == 0.0))
))
THEN (
(MODIFY (OBJ: $x
(totalload $dl+$ll+$eql)
) 1000 0.001)
)
COMMENT:"Rule to calculate footing totalload" 70

(RULE: passroofload 500
IF
((CLASS: roof OBJ: $x
(totalload == $tl)) AND
(CLASS: column OBJ: $y
((passedload == 0.0) AND
(totalload == $ctl)))
)
THEN ( 80
(MODIFY (OBJ: $y
(totalload $ctl+$tl)
(passedload $tl)
) 1000 0.001)
)
COMMENT:"Rule to pass roof totalload and calculate column loads")

(RULE: passcolumnload 100
IF 90
(CLASS: column OBJ: $x
(totalload == $tl)) AND
(CLASS: footing OBJ: $y
```

### E.3 CABIN calculation rulefiles

---

```
((passedload == 0.0) AND
(totalload == $ctl))
)
THEN (
(MODIFY (OBJ: $y
(totalload $ctl+$tl)
(passedload $tl)
) 1000 0.001)
)
COMMENT:"Rule to pass roof totalload and calculate column loads")
```

100

---

## E.3 CABIN calculation rulefiles

---

### E.3.2 Beam.rul

This rulefile calculates the cross section area of the beam according to the loading.

---

```
(RULE: createbeam 1000
IF
(CLASS: beam OBJ: $x
(b_length == 0.0)
)
THEN (
(MODIFY (OBJ: $x
(deadload 100.0)
(liveload 200.0)
(b_length 100.0)
) 1000 0.001)
)
COMMENT:"Rule to create the beam")

(RULE: calculatetotalload 900
IF
(CLASS: beam OBJ: $x
(((deadload == $dl) AND
(liveload == $ll)) AND
(totalload == 0.0))
)
THEN (
(BIND VAR:$dl $dl*1.4)
(BIND VAR:$ll $ll*1.7)
(MODIFY (OBJ: $x
(totalload $dl+$ll)
) 1000 0.001)
)
COMMENT:"Rule to calculate beam totalload")

(RULE: assumewidthdepth 800
IF
(CLASS: beam OBJ: $x
((b_length == $len) AND
(b_width == 0.0))
)
THEN (
(MODIFY (OBJ: $x
(b_width $len/16)
(b_depth $len/16)
) 1000 0.001)
)
COMMENT:"Rule to calculate beam width & depth")

(RULE: calculatemoment 700
IF
(CLASS: beam OBJ: $x
```



### E.3 CABIN calculation rulefiles

---

```
((b_length == $len) AND
 (totalload == $tl) AND
 (moment == 0.0))
)
THEN (
(BIND VAR:$len1 $tl*$len)
(BIND VAR:$len2 $len/8)
(MODIFY (OBJ: $x
(moment $len1*$len2)
) 1000 0.001)
)
COMMENT:"Rule to calculate beam moment")
50

(RULE: calculateeff_depth 600
IF
(CLASS: beam OBJ: $x
((b_depth == $dep) AND
 (b_eff_depth == 0.0))
)
THEN (
(MODIFY (OBJ: $x
(b_eff_depth $dep-2.5)
) 1000 0.001)
)
COMMENT:"Rule to calculate beam effective depth")
70

(RULE: calculatearea 500
IF
(CLASS: beam OBJ: $x
(((moment == $mom) AND
 (b_eff_depth == $efdep)) AND
 (b_reinf_area == 0.0))
)
THEN (
(BIND VAR:$i 0.875*$efdep)
(BIND VAR:$j 0.9*36.0)
(BIND VAR:$k $i*$j)
(MODIFY (OBJ: $x
(b_reinf_area $mom/$k)
) 1000 0.001)
)
COMMENT:"Rule to calculate beam reinforcement area")
80
90
```

---

## E.3 CABIN calculation rulefiles

---

### E.3.3 Column.rul

This rulefile calculates the cross section area of the column according to the loading.

---

```
(RULE: createcolumn 1000
IF
(CLASS: column OBJ: $x
(c_length == 0.0)
)
THEN (
(MODIFY (OBJ: $x
(axialload 100.0)
(c_length 15.0)
(c_eff_length 0.8*15.0)
) 1000 0.001)
)
COMMENT:"Rule to create the column")

(RULE: calculate_reinf_area 900
IF
(CLASS: column OBJ: $x
(((axialload == $al) AND
(c_eff_length == $len)) AND
(c_reinf_area == 0.0))
)
THEN (
(BIND VAR:$i 1.31-2.77)
(BIND VAR:$j $len/12.84)
(BIND VAR:$k $i*$j)
(BIND VAR:$fa 2.77+$k)
(BIND VAR:$pu 1.4*$al)
(MODIFY (OBJ: $x
(c_reinf_area $pu/$fa)
) 1000 0.001)
)
COMMENT:"Rule to calculate column reinforced area")

(RULE: calculatesteelarea 800
IF
(CLASS: column OBJ: $x
((c_reinf_area == $area) AND
(c_steel_area == 0.0))
)
THEN (
(MODIFY (OBJ: $x
(c_steel_area $area*0.04)
) 1000 0.001)
)
COMMENT:"Rule to calculate column initial area")
```

---

## E.3 CABIN calculation rulefiles

---

### E.3.4 Slab.rul

This rulefile calculates the cross section area of the slab according to the loading.

---

```
(RULE: createslab 1000
IF
(CLASS: slab OBJ: $x
(s_length == 0.0)
)
THEN (
(MODIFY (OBJ: $x
(deadload 100.0)
(liveload 50.0)
(s_length 100.0)
) 1000 0.001)
)
COMMENT: "Rule to create the slab")

(RULE: calculatetotalload 900
IF
(CLASS: slab OBJ: $x
(((deadload == $dl ) AND
(liveload == $ll)) AND
(totalload == 0.0))
)
THEN (
(BIND VAR:$dl $dl*1.4)
(BIND VAR:$ll $ll*1.7)
(MODIFY (OBJ: $x
(totalload $dl+$ll)
) 1000 0.001)
)
COMMENT: "Rule to calculate the slab total load")

(RULE: assumedepth 800
IF
(CLASS: slab OBJ: $x
((s_length == $len) AND
(s_width == $wid))
)
THEN (
(MODIFY (OBJ: $x
(s_depth $len/20)
(s_eff_depth $len/20)
) 1000 0.001)
)
COMMENT:"Rule to calculate slab width & depth")

(RULE: calculatemoment 700
IF
(CLASS: slab OBJ: $x
```

### E.3 CABIN calculation rulefiles

---

```
((s_length == $len) AND
 (totalload == $tl) AND
 (moment == 0.0))
)
THEN(
(MODIFY (OBJ: $x
(moment $tl*$len*$len/8)
) 1000 0.001)
)
COMMENT:"Rule to calculate slab width & depth")

(RULE: calculatearea 500
IF
(CLASS: slab OBJ: $x
(((moment == $mom) AND
 (s_depth == $dep)) AND
 (s_reinf_area == 0.0))
)
THEN (
(BIND VAR:$i 0.9)
(BIND VAR:$j 60000.0)
(BIND VAR:$k 0.925)
(BIND VAR:$l 12000.0)
(BIND VAR:$m $dep-1.0)
(BIND VAR:$n $i*$j*$k*$m)
(MODIFY (OBJ: $x
(s_reinf_area $mom*$l/$n)
(s_eff_depth $m)
) 1000 0.001)
)
COMMENT: "Rule to calculate slab reinforcement area")
```

50  
60  
70  
80

---

## E.4 CABIN script

---

### E.4 CABIN script

The following is the script to generate all the information of Cabin Design application. Before the script is executed in CONGEN, it must be compiled successfully.

#### E.4.1 create\_cabin.c

---

```
/*  
 * create_cabin: File to conveniently create the  
 * CABIN DESIGN application, without having to go through all the  
 * elaborate rigmarole of the user interface  
 *  
 * Author: Johanes C. Sugiono  
 * Intelligent Engineering Systems Laboratory  
 * Massachusetts Institute of Technology  
 * Date: December 2, 1994  
 */  
#include <iostream.h> 10  
#include <E/trans.h>  
#include "congen.h"  
#include "context.h"  
#include "data_manager.h"  
#include "goal.h"  
#include "goallist.h"  
#include "plan.h"  
#include "artifact.h"  
#include "geometry.h" 20  
#include "decision.h"  
#include "SpecFrame.h"  
#include "classobj.h"  
  
#include "ClassManager.h"  
#include "InstanceManager.h"  
#include "AppManager.h"  
#include "ui.h"  
extern Congen * TheCongen;  
extern Data_manager* TheDataManager; 30  
extern ClassManager *class_manager_ptr;  
extern AppManager* app_manager_ptr;  
extern InstanceManager* instance_manager_ptr;  
  
#ifndef PUBLIC  
#define PUBLIC 2  
#endif  
  
#ifndef PERSISTENT  
#define PERSISTENT 1 40  
#endif
```

---

## E.4 CABIN script

---

```
void make_demo()
{
  int i;
  TheCongen->set_application("Cabin_Design");

  //Setting up root goal

  TheDataManager->create_ArfGoal("create_cabin",NULL);          50
  Goal* gl;
  gl= TheDataManager->lookup_Goal("create_cabin");
  gl->set_choice("Cabin");
  gl->set_effects_rulebase("cabin_eff.rul");
  TheCongen->set_rtgoal(gl);

  cout << "Finished creating rootgoal \n";

  //DDeclaring the goals                                     60

  TheDataManager->create_ModGoal("set_columns","Cabin",
                                "numcolumns",NULL);
  TheDataManager->create_ModGoal("set_girders","Cabin",
                                "numgirders",NULL);
  TheDataManager->create_ModGoal("set_supporting_beams","Cabin",
                                "numbeams",NULL);
  TheDataManager->create_ModGoal("set_purlins","Cabin",
                                "numpurlins",NULL);
  TheDataManager->create_ModGoal("set_trusses","Cabin",
                                "numtruss",NULL);          70
  TheDataManager->create_ModGoal("set_north_wall","Cabin",
                                "northwalltype",NULL);
  TheDataManager->create_ModGoal("set_south_wall","Cabin",
                                "southwalltype",NULL);
  TheDataManager->create_ModGoal("set_east_wall","Cabin",
                                "eastwalltype",NULL);
  TheDataManager->create_ModGoal("set_west_wall","Cabin",
                                "westwalltype",NULL);
  TheDataManager->create_ModGoal("set_foundation","Cabin",
                                "foundationtype",NULL);    80
  TheDataManager->create_AbsGoal("set_beams",NULL);
  TheDataManager->create_AbsGoal("set_walls",NULL);
  TheDataManager->create_AbsGoal("set_trussys",NULL);
  TheDataManager->create_AbsGoal("set_beam_column_grid",NULL);

  cout << "Finished declaring all the goals in the application \n";

  //EEntering plans and expansion goals
  E_CommitTransaction();
  E_BeginTransaction();                                     90

  TheDataManager->create_Plan("cabin_design_plan");
  Plan* pl = TheDataManager->lookup_Plan("cabin_design_plan");
  pl->set_parent("Cabin",0);
  pl->add_goal("set_beam_column_grid");
```

## E.4 CABIN script

---

```
pl->add_goal("set_trussys");
pl->add_goal("set_walls");
pl->add_goal("set_foundation");
cout << "create plan cabin_design_plan in the application \n";
                                                                    100

gl = TheDataManager->lookup_Goal("set_columns");
gl->set_choice("0");
gl->set_choice("4");
gl->set_choice("6");
gl->set_parent("set_beam_column_grid_plan");
gl->set_effects_rulebase("set_columns_eff.rul");

gl = TheDataManager->lookup_Goal("set_beams");
gl->set_choice("set_beams_plan");
gl->set_parent("set_beam_column_grid_plan");
                                                                    110
cout << "create goals set columns & beams in the application \n";

gl = TheDataManager->lookup_Goal("set_trussys");
gl->set_choice("set_trussys_plan");
gl->set_parent("cabin_design_plan");

gl = TheDataManager->lookup_Goal("set_walls");
gl->set_choice("set_walls_plan");
gl->set_parent("cabin_design_plan");
                                                                    120

gl = TheDataManager->lookup_Goal("set_foundation");
gl->set_choice("Mat");
gl->set_choice("Strip");
gl->set_choice("Spread");
gl->set_rulebase("set_foundation.rul");
gl->set_parent("cabin_design_plan");
gl->set_effects_rulebase("set_foundation_eff.rul");
cout << "create goals set trusses, walls and foundation in the application \n";

TheDataManager->create_Plan("set_beams_plan");
                                                                    130
pl = TheDataManager->lookup_Plan("set_beams_plan");
pl->set_parent("set_beams",1);
pl->add_goal("set_girders");
pl->add_goal("set_supporting_beams");

TheDataManager->create_Plan("set_beam_column_grid_plan");
pl = TheDataManager->lookup_Plan("set_beam_column_grid_plan");
pl->set_parent("set_beam_column_grid",1);
pl->add_goal("set_columns");
pl->add_goal("set__beams");
                                                                    140

TheDataManager->create_Plan("set_trussys_plan");
pl = TheDataManager->lookup_Plan("set_trussys_plan");
pl->set_parent("set_trussys",1);
pl->add_goal("set_purlins");
pl->add_goal("set_trusses");

TheDataManager->create_Plan("set_walls_plan");
pl = TheDataManager->lookup_Plan("set_walls_plan");
```

## E.4 CABIN script

---

```
pl->set_parent("set_walls",1);
pl->add_goal("set_north_wall");
pl->add_goal("set_south_wall");
pl->add_goal("set_east_wall");
pl->add_goal("set_west_wall");

cout << "Finished creating all the plans and expansion goals in the application \n";

//EEntering the goals
E_CommitTransaction();
E_BeginTransaction();

gl = TheDataManager->lookup_Goal("set_girders");
gl->set_choice("0");
gl->set_choice("2");
gl->set_rulebase("set_girders.rul");
gl->set_parent("set_beams_plan");
gl->set_effects_rulebase("set_girders_eff.rul");

gl = TheDataManager->lookup_Goal("set_supporting_beams");
gl->set_choice("0");
gl->set_choice("1");
gl->set_choice("2");
gl->set_choice("3");
gl->set_choice("6");
gl->set_choice("7");
gl->set_rulebase("set_supporting_beams.rul");
gl->set_parent("set_beams_plan");
gl->set_effects_rulebase("set_supporting_beams_eff.rul");

gl = TheDataManager->lookup_Goal("set_purlins");
gl->set_choice("0");
gl->set_choice("2");
gl->set_choice("4");
gl->set_rulebase("set_purlins.rul");
gl->set_parent("set_trusssys_plan");
gl->set_effects_rulebase("set_purlins_eff.rul");

gl = TheDataManager->lookup_Goal("set_trusses");
gl->set_choice("2");
gl->set_choice("3");
gl->set_rulebase("set_trusses.rul");
gl->set_parent("set_trusssys_plan");
gl->set_effects_rulebase("set_trusses_eff.rul");

gl = TheDataManager->lookup_Goal("set_north_wall");
gl->set_choice("Shear");
gl->set_choice("Brick");
gl->set_choice("Opening");
gl->set_rulebase("set_north_wall.rul");
gl->set_parent("set_walls_plan");
gl->set_effects_rulebase("set_north_wall_eff.rul");

gl = TheDataManager->lookup_Goal("set_south_wall");
```



## E.4 CABIN script

---

```
gl->set_choice("Shear");
gl->set_choice("Brick");
gl->set_choice("Opening");
gl->set_rulebase("set_south_wall.rul");
gl->set_parent("set_walls_plan");
gl->set_effects_rulebase("set_south_wall_eff.rul");
                                                                    210

gl = TheDataManager->lookup_Goal("set_east_wall");
gl->set_choice("Shear");
gl->set_choice("Brick");
gl->set_choice("Opening");
gl->set_rulebase("set_east_wall.rul");
gl->set_parent("set_walls_plan");
gl->set_effects_rulebase("set_east_wall_eff.rul");

gl = TheDataManager->lookup_Goal("set_west_wall");
gl->set_choice("Shear");
gl->set_choice("Brick");
gl->set_choice("Opening");
gl->set_rulebase("set_west_wall.rul");
gl->set_parent("set_walls_plan");
gl->set_effects_rulebase("set_west_wall_eff.rul");

cout << "Finished creating goals in the application \n";

//DDeclaring the rulefiles
E_CommitTransaction();
E_BeginTransaction();
                                                                    230

TheCongen->add_rulefile("cabin_eff.rul");
TheCongen->add_rulefile("set_columns_eff.rul");
TheCongen->add_rulefile("set_east_wall.rul");
TheCongen->add_rulefile("set_east_wall_eff.rul");
TheCongen->add_rulefile("set_foundation.rul");
TheCongen->add_rulefile("set_foundation_eff.rul");
TheCongen->add_rulefile("set_girders.rul");
TheCongen->add_rulefile("set_girders_eff.rul");
TheCongen->add_rulefile("set_north_wall.rul");
TheCongen->add_rulefile("set_north_wall_eff.rul");
TheCongen->add_rulefile("set_purlins.rul");
TheCongen->add_rulefile("set_purlins_eff.rul");
TheCongen->add_rulefile("set_south_wall.rul");
TheCongen->add_rulefile("set_south_wall_eff.rul");
TheCongen->add_rulefile("set_supporting_beams.rul");
TheCongen->add_rulefile("set_supporting_beams_eff.rul");
TheCongen->add_rulefile("set_trusses.rul");
TheCongen->add_rulefile("set_trusses_eff.rul");
TheCongen->add_rulefile("set_west_wall.rul");
TheCongen->add_rulefile("set_west_wall_eff.rul");
                                                                    240
                                                                    250

cout << "Finished adding rulefiles in the application \n";

//CCreating the classes
E_CommitTransaction();
```

## E.4 CABIN script

---

```
E_BeginTransaction();

class_manager_ptr->create_class("Cabin");
app_manager_ptr->add_class("Cabin_Design","Cabin");
class_manager_ptr->add_attributes_f("Cabin","Cabin.attr",PUBLIC);

cout << "Class: Cabin finished parsing \n" ;

class_manager_ptr->create_class("Cabin_part");
app_manager_ptr->add_class("Cabin_Design","Cabin_part");

class_manager_ptr->create_class("Site");
app_manager_ptr->add_class("Cabin_Design","Site");
class_manager_ptr->add_attributes_f("Site","Site.attr", PUBLIC);

cout << "Class: Site finished parsing \n" ;

class_manager_ptr->create_class("Struct_mbr");
app_manager_ptr->add_class("Cabin_Design","Struct_mbr");
class_manager_ptr->add_attributes_f("Struct_mbr","Struct_mbr.attr",PUBLIC);

cout << "Class: Struct_mbr finished parsing \n" ;

class_manager_ptr->create_class("Joint");
app_manager_ptr->add_class("Cabin_Design","Joint");
class_manager_ptr->add_attributes_f("Joint","Joint.attr",PUBLIC);

cout << "Class: Joint finished parsing \n" ;

class_manager_ptr->create_class("Linear_mbr");
app_manager_ptr->add_class("Cabin_Design","Linear_mbr");
class_manager_ptr->add_attributes_f("Linear_mbr","Linear_mbr.attr",PUBLIC);

cout << "Class: Linear_mbr finished parsing \n" ;

class_manager_ptr->create_class("Area_mbr");
app_manager_ptr->add_class("Cabin_Design","Area_mbr");
class_manager_ptr->add_attributes_f("Area_mbr","Area_mbr.attr",PUBLIC);

cout << "Class: Area_mbr finished parsing \n" ;

E_CommitTransaction();
E_BeginTransaction();

class_manager_ptr->create_class("Beam");
app_manager_ptr->add_class("Cabin_Design","Beam");
class_manager_ptr->add_attributes_f("Beam","Beam.attr",PUBLIC);

cout << "Class: Beam finished parsing \n" ;

class_manager_ptr->create_class("Column");
app_manager_ptr->add_class("Cabin_Design","Column");
class_manager_ptr->add_attributes_f("Column","Column.attr",PUBLIC);
```

## E.4 CABIN script

---

```
cout << "Class: Column finished parsing \n" ;

class_manager_ptr->create_class("Pier");
app_manager_ptr->add_class("Cabin_Design","Pier");
class_manager_ptr->add_attributes_f("Pier","Pier.attr",PUBLIC);

cout << "Class: Pier finished parsing \n" ;

class_manager_ptr->create_class("Wall");
app_manager_ptr->add_class("Cabin_Design","Wall");
class_manager_ptr->add_attributes_f("Wall","Wall.attr",PUBLIC);

cout << "Class: Wall finished parsing \n" ;

E_CommitTransaction();
E_BeginTransaction();

class_manager_ptr->create_class("Slab");
app_manager_ptr->add_class("Cabin_Design","Slab");
class_manager_ptr->add_attributes_f("Slab","Slab.attr",PUBLIC);

cout << "Class: Slab finished parsing \n" ;

class_manager_ptr->create_class("Ribbed_slab");
app_manager_ptr->add_class("Cabin_Design","Ribbed_slab");
class_manager_ptr->add_attributes_f("Ribbed_slab","Ribbed_slab.attr",PUBLIC);

cout << "Class: Ribbed_slab finished parsing \n" ;

class_manager_ptr->create_class("Waffle_slab");
app_manager_ptr->add_class("Cabin_Design","Waffle_slab");
class_manager_ptr->add_attributes_f("Waffle_slab","Waffle_slab.attr",PUBLIC);

cout << "Class: Waffle_slab finished parsing \n" ;

class_manager_ptr->create_class("Wall_opening");
app_manager_ptr->add_class("Cabin_Design","Wall_opening");
class_manager_ptr->add_attributes_f("Wall_opening","Wall_opening.attr",PUBLIC);

cout << "Class: Wall_opening finished parsing \n" ;

E_CommitTransaction();
E_BeginTransaction();

class_manager_ptr->create_class("Truss_member");
app_manager_ptr->add_class("Cabin_Design","Truss_member");
class_manager_ptr->add_attributes_f("Truss_member","Truss_member.attr",PUBLIC);

cout << "Class: Truss_member finished parsing \n" ;

class_manager_ptr->create_class("Truss_system");
app_manager_ptr->add_class("Cabin_Design","Truss_system");
class_manager_ptr->add_attributes_f("Truss_system","Truss_system.attr",PUBLIC);
```

## E.4 CABIN script

---

```
cout << "Class: Truss_system finished parsing \n" ;

class_manager_ptr->create_class("Mat_found");
app_manager_ptr->add_class("Cabin_Design","Mat_found");
class_manager_ptr->add_attributes_f("Mat_found","Mat_found.attr",PUBLIC); 370

cout << "Class: Mat_found finished parsing \n" ;

class_manager_ptr->create_class("Spread_found");
app_manager_ptr->add_class("Cabin_Design","Spread_found");
class_manager_ptr->add_attributes_f("Spread_found","Spread_found.attr",PUBLIC);

cout << "Class: Spread_found finished parsing \n" ;

class_manager_ptr->create_class("Strip_footing"); 380
app_manager_ptr->add_class("Cabin_Design","Strip_footing");
class_manager_ptr->add_attributes_f("Strip_footing","Strip_footing.attr",PUBLIC);

cout <<"Finished creating all the classes \n";

// base class relationships
E_CommitTransaction();
E_BeginTransaction();

class_manager_ptr->add_base("Planar_sys", "Struct_sys", PUBLIC); 390
class_manager_ptr->add_base("Boundary_sys", "Struct_sys", PUBLIC);
class_manager_ptr->add_base("Joint", "Struct_mbr", PUBLIC);
class_manager_ptr->add_base("Linear_mbr", "Struct_mbr", PUBLIC);
class_manager_ptr->add_base("Area_mbr", "Struct_mbr", PUBLIC);
class_manager_ptr->add_base("Beam", "Linear_mbr", PUBLIC);
class_manager_ptr->add_base("Column", "Linear_mbr", PUBLIC);
class_manager_ptr->add_base("Pier", "Linear_mbr", PUBLIC);
class_manager_ptr->add_base("Truss_member", "Linear_mbr", PUBLIC);
class_manager_ptr->add_base("Slab", "Area_mbr", PUBLIC);
class_manager_ptr->add_base("Ribbed_slab", "Slab", PUBLIC); 400
class_manager_ptr->add_base("Waffle_slab", "Slab", PUBLIC);
class_manager_ptr->add_base("Wall", "Area_mbr", PUBLIC);
class_manager_ptr->add_base("Wall_opening", "Area_mbr", PUBLIC);
class_manager_ptr->add_base("Strip_footing", "Area_mbr", PUBLIC);
class_manager_ptr->add_base("Mat_found", "Area_mbr", PUBLIC);
class_manager_ptr->add_base("Spread_found", "Area_mbr", PUBLIC);
class_manager_ptr->add_base("Truss_system", "Area_mbr", PUBLIC);

class_manager_ptr-> add_child("Struct_sys","Planar_sys");
class_manager_ptr-> add_child("Struct_sys","Boundary_sys"); 410
class_manager_ptr-> add_child("Struct_mbr","Joint");
class_manager_ptr-> add_child("Struct_mbr","Linear_mbr");
class_manager_ptr-> add_child("Struct_mbr","Area_mbr");
class_manager_ptr-> add_child("Linear_mbr","Beam");
class_manager_ptr-> add_child("Linear_mbr","Column");
class_manager_ptr-> add_child("Linear_mbr","Pier");
class_manager_ptr-> add_child("Linear_mbr","Truss_member");
class_manager_ptr-> add_child("Area_mbr","Truss_member");
class_manager_ptr-> add_child("Area_mbr","Slab");
```

## E.4 CABIN script

---

```
class_manager_ptr-> add_child("Slab","Ribbed_slab");           420
class_manager_ptr-> add_child("Slab","Waffle_slab");
class_manager_ptr-> add_child("Area_mbr","Wall");
class_manager_ptr-> add_child("Area_mbr","Wall_opening");
class_manager_ptr-> add_child("Area_mbr","Strip_footing");
class_manager_ptr-> add_child("Area_mbr","Mat_found");
class_manager_ptr-> add_child("Area_mbr","Spread_found");

cout << "Finished adding the base class relationships \n";

E_CommitTransaction();                                       430
E_BeginTransaction();
}
```

---