Testing A Federation Architecture in Collaborative Design Process

by

Jen-Diann Chiou

S.B., Civil Engineering
National Taiwan University, 1992

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

May 1996

Author ....
Department of Civil and Environmental Engineering

Certified by ....
Professor Robert D. Logcher
Department of Civil and Environmental Engineering
Thesis Supervisor

Accepted by ....
Joseph M. Sussman
Chairman, Departmental Committee on Graduate Studies
Department of Civil and Environmental Engineering

Testing a Federation Architecture in Collaborative Design Process

by

Jen-Diann Chiou

Submitted to the Department of Civil and Environmental Engineering on May 23, 1996 in partial fulfillment of the requirements for the degree of Master of Science in Civil and Environmental Engineering

Abstract

A structural design agent program is presented which is based on a testbed information infrastructure for supporting multi-institutional collaborative facility engineering, called a Federation Architecture. Equipped with the ability to communicate with other design agents, the structural design agent is able to carry out the design process in a collaborative fashion. The Structural Engineering Vocabulary, a data model based on object-oriented methodology, is described in details. Integration of several software modules, including a graphical user interface, a knowledge-based inference engine, and an object-oriented database management system, present major implementation issues for this project. Experience gained during the process of developing the structural agent is delineated. Finally, two examples, one for cabin and the other for firestation design, are given to demonstrate the capability of the structural agent and the overall collaborative framework. The last chapter describes our experience with and requirements for a useful federation architecture. Recommendations for improvements to the federation architecure are described in details.

Thesis Supervisor: Robert D. Logcher
Title: Professor of Civil and Environmental Engineering

Acknowledgements

This work would not have been possible without the help of many individuals and institutions.

First, I am indebted to my thesis advisor, Professor Robert D. Logcher, for his tremendous support of my work, for spending many long days discussing and reviewing my research and for his meticulous review of my thesis. He is the model of excellence both academically and in everyday life. He demonstrates, through everything he does, big and small, the importance of "the big picture" in accomplishing tasks, taking initiative, acting promptly, working hard, and attending to detail. For me, Professor Logcher embodies the paradigm of teaching and research.

I appreciate Professors Ming-Teh Wang and San-Cheng Chang, for their illumination of computer aided engineering and encouragement.

Professor Paramasivam Jayachandran provided most timely guidance on structural engineering issues during my research. I also would like to thank Professor Ulrich Flemming, Drs. Mike Case, Taha Khedro, Dr.-to-be James Snyder, and Mr. Daniel Malmer, for their help during this research project. It is a wonderful experience working with these brilliant people. I gratefully acknowledge the support of Construction Engineering Research Laboratory of Corps of Engineers.

I would like to express my thanks to Professors Jerome Connor, John Williams, Joseph Sussman, and Feniosky Pena-Mora, for their invaluable guidance, advice and support throughout my studies here.

I deeply enjoyed the camaraderie of fellow IESL students, Drs. Ruaidhri O'Connor, Nabha Rege, Kevin Amaratunga, and Mr. Dennis Shelden. Mrs. Joan McCusker always gave me warm greetings and best wishes just when I need them the most.

I would like to express my thanks to many friends during my stay at MIT who helped me throughout the first two years: Dr. Jeng-Feng Lee, a postdoctoral research fellow in the MIT Research Laboratory of Electronics, deserves much gratitude for his help and patience. Drs-to-be Ming-Chih Weng, Hsien-Jen Tien; Mr's. Te-San Liao, Yu-Hsuan Su, Wen-Chung Chang, and Shun-Ming Lee; Ms's. Faustina Tsai and Chu-Jun Huang.

I deeply appreciate the thoughtfulness and encouragement given by Ms. Ching-Wen Hsu and her family during the past years.

This thesis is dedicated to my parents, to whom I owe the most, for their unconditional support and sacrifice over all these years. I also feel very fortunate to be the only boy in my family because my two sisters are always there to help me through.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Collaborative Engineering Design

Collaborative engineering design [18] is the systematic, intelligent generation and evaluation of specifications for artifacts whose form and function achieve stated objectives and satisfy specified constraints through sporadic work of engineers dispersed geographically and working in distinct disciplines. The aid of computing and communication facilities is essential to this process. The ultimate goal of collaborative design system developers is to support interactions between teams of multiple engineers as they work on product development activities. The problems arising in collaborative design are as diverse as the technologies and tools used for solving them. Some problems can be solved with simple procedural models, others may require more sophisticated mathematical modeling; while others require heuristic-based reasoning approaches. The ability to effectively integrate and utilize heterogeneous design information - generated and consumed by multiple engineering tasks - is vital for industrial improvement.

There are several characteristics of the collaborative design process:

- The integration of design and construction involves professionals from many different disciplines, including architects, structural engineers, electrical and mechanical engineers, developers, and interior designers. Each discipline represents a specialization that is often not well understood by the others. Variations in background, education, and experience lead to discrepancies in terminology, attitude, style, and organization of information.
- Each discipline has its own perspective of the building. For example, a structural

engineer is interested in load distribution in the building and its translation to size and location of beams and columns, whereas an architect is interested in the character and expressiveness of the building as a whole, as well as in the function and appearance of the spaces and enclosures. Attempting to maintain all views of the building in a single representation, either on drawings or in a database, introduces problems of redundancy and consistency of information.

- Currently, communication occurs verbally or through the media of blueprints and written documentation. This tradition has resulted in limiting the data transferred from one discipline to another to a description of the solution, typically including only enough information for the contractor to construct the building. Intermediate decisions and justification for decisions made during design or construction planning are not communicated. Often the main medium is the drawings that the architects and engineers generate.
- Each discipline uses computer programs to assist in their tasks. For example, architects use AutoCAD or Microstation systems to produce drawings and structural engineers use finite element programs to analyze the walls, framing and slabs of the building. Not only do the computer systems vary from discipline to discipline, but the degree of automation varies from project to project. One group may make extensive use of computer-based tools, including database management systems, graphic exchange standards, CAD systems, etc., whereas another group may make limited use of computers for very specific tasks. Typically, because of the need to pass information from one group to another, the preparation of input data and interpretation of output is done manually.

- Each organization involved in design and/or construction make use of hardware and software for its own purposes, and those tools may not be compatible with other organizations. Even though there are efforts underway to use standard representations of design drawings, such as DXF or IGES, there are practical problems of translation to other CAD systems and legal concerns when errors in the data occurs. The lack of an automatic means of transferring data from one computer program to another makes it difficult to use existing software within an integrated environment.

In order to resolve these limitations, the ACL project was proposed to develop a testbed for the exploration of integration and communication issues in the building design, and the federation architecture was chosen as the information infrastructure for both communication and information exchange management. The systems used in this testbed include an architectural design system called SEED, a structural engineering environment developed at MIT, an energy analysis environment provided by BLAST and ACE. The owner (client) and project manager perspective is supported by ACE. In this section, we discuss each of these systems.

The goal of this project is to determine the tasks and level of effort required to connect the design systems in a collaborative design environment using a federation architecture and then to exercise the environment in an exploratory fashion. Several questions are being researched. How should the vocabulary (ontology) for different systems, which use different internal data and knowledge representations, be expressed so that communication and translation can occur? What should the characteristics of an application programmatic interface be so that systems can communicate with a facilitator, which can distribute data among participants? What capabilities are required of each agent to be a participating member of the federation? What does this mean in terms of software implementation?

How should the facilitator approach the task of translating between system representations and how much manual work is required. Is translation ultimately an n^2 problem or are there intermediate standards?

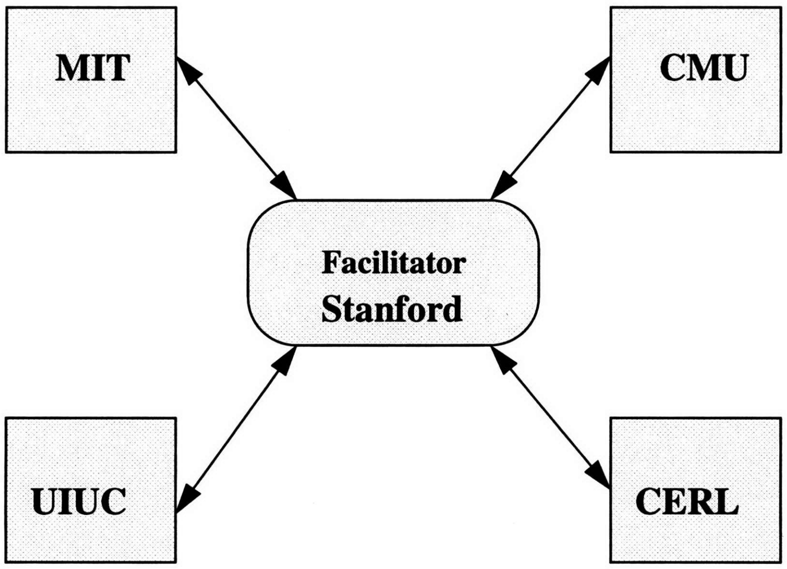


Figure 1.1: Relationships between the Agents

SEED (Software Environment to support the Early phases in building Design)

SEED[22] is an architectural design environment that generates a complete architectural program based on a rough project specification (e.g., building type, size, budget, site description and other context-specific information). The architectural program consists of the basic functional units needed for a building with the specified characteristics and requirements (shape, performance) that a configuration of these units must satisfy. A major component of the SEED environment is SEED-Layout, which provides support for the early phases in building design. SEED-Layout supports specifically the rapid generation of internal, computable design representations, including representations of conceptual alternatives and variants of such alternatives, so that the space of possibilities available for a given project can be more systematically explored, especially if feedback from other agents is received early before major commitments have been made. The effectiveness of the generative capabilities of SEED-Layout rely on an explicit problem specification, which can be edited interactively by designers and evolves as they gain more experience with a specific project. This general outlook makes SEED-Layout particularly suitable for collaborative design aiming at greater interactions between the teams involved at the early, crucial stage, in order to overcome the traditional, linear decision sequence that does not allow for iterations and the exploration of alternative designs and their trade-offs.

In addition to these generative capabilities, SEED-Layout can retrieve functional programs for recurring building types from a database, which can then be edited like any other problem specification. This gives SEED-Layout a programming capability that is minimally sufficient to support the first task listed above. Tasks involving the display, generation and evaluation of three-dimensional spatial and physical components are concentrated in another module, which takes over once SEED-Layout has completed a programming or layout task. SEED-Layout uses internally an object-oriented representation that is highly structured and rich in content. It relies specifically on two hierarchies: (i) an inheritance hierarchy that allows objects in one class to inherit attributes and behavior from a superclass, and (ii) a part-of or constituent hierarchy describing geometric assemblies at various levels of abstraction.

Agent Collaboration Environment (ACE)

The ACE is an implementation of a general purpose model of agent collaboration called the Discourse Model. Simply put, ACE treats assertions by agents as opinions, and provides functionality for agents to discuss those opinions. Historically, the roots of ACE's development lie in the area of blackboard systems, although the traditional convention that agents not be aware of each is not enforced. The ACE model has four readily identified parts. First, an agent model is defined that describes agents and their behavior. Second, a workspace contains agents and modeling artifacts, including frames (classes), semantic links (relation between classes), and constraints (enforceable relations between attributes of frames, through which values may propagate). Third, a network communication model enables the workspaces of individuals to share their models, resulting in a Virtual Workspace. Finally, a set of closely linked interfaces with CAD and databases links ACE with legacy software.

ACE/Building Loads Analysis and System Thermodynamic (BLAST)

The primary objective of ACE/BLAST system is to conduct an analysis of the thermal behavior of the building and recommend changes to improve comfort and reduce life cycle cost. ACE/BLAST needs a description of the preliminary design of the building geometry and other useful information for thermal simulation. BLAST is a comprehensive computer program used for predicting energy consumption and energy systems performance in buildings. It is currently used throughout the world as a stand-alone energy analysis tool. ACE facilitates an environment for the thermal engineering in providing an interactive facility in specifying and configuring the thermal zones, in selecting the surface types from the libraries by incorporating tools such as computer aided drafting systems, and in selecting appropriate fan systems and plant parameters for simulation. In addition, ACE provides checklists for the subagents of the thermal analysis during preparation of the BLAST input file for simulation.

The major processes, which constitute the task of energy analysis are: configuring the thermal zone, identifying the surfaces in the thermal zone, translating the building information into the BLAST objects and formulating scheduled loads in each thermal zone. Then, BLAST generates building descriptions for the simulation and adjusts building parameters for the optimal design. It selects surface and subsurface types from its libraries or creates the surface type using materials from the materials library. Some of libraries for surface types supported are walls, roofs, floors, doors and windows. Next, it translates the building geometry to the BLAST building objects. It finally prepares schedule of loads in each thermal zone using scheduled loads hints.

ACE/Owner/Project Manager

Owner and Project Manager agents have also been implemented in ACE. Their function is to establish the initial functional requirements for a building, situate the proposed building on a site, and pass the information to the architectural design system (SEED). They will also provide building visualization capabilities for a simulated human client through ACE's interface with commercial CAD software.

Perspective of Structural Engineering Design

To illustrate the importance of representation in structural design, and the diversity of representations that we actually use for artifacts and process representation, we present a brief discussion of the typical structural engineering problem in design and development.

A structural need is identified after reviewing the architectural requirements, whether it is for a cabin or a firestation. A structural concept is chosen first, perhaps a simple steel frame and steel roof truss for the cabin, something considerably more complex for the firestation, and we move to preliminary design. In this stage we usually restrict our efforts to preliminary sizing of the principal structural members, the object being to see whether the type of structural system that we have selected is practically feasible. We then move on to scoping out the structure by estimating the types and sizes of the remaining members (e.g. in the cabin design, purlins for the roof truss and floor joists as needed). Then we zero in on the final, detailed design in which we calculate actual dimensions and placements for all members and their connections. In the final step we check to insure that our design meets all the specification requirements, including both applicable building codes as well as design codes such as that of the American Institute of Steel Construction (AISC) or American Concrete Institute (ACI), which lay out performance specifications for members and connections.

Let us now examine the kinds of design knowledge deployed in completing such as structural design. Among the kinds of knowledge we apply are: classical mechanics (e.g., Newton's laws); structural mechanics (e.g., model of columns and beams); geometry of structures (e.g., relating the geometry of members and assemblages of members to the orientation of the loads that they are expected to carry); behavioral models (e.g., modeling the stiffness of a complete frame); algorithmic models of structures (e.g., finite element method (FEM) computer codes); structural design codes (e.g., AISC code); heuristic and experimental knowledge, both derived from practice and encoded in specifications and meta-knowledge about how and where to invoke the other kinds of knowledge. Much of this knowledge is multilayered. For example, our understanding of the behavior of structural systems is realized at three distinct levels: spatial layout (e.g., where to place columns to achieve clear floor spans), functional (e.g., how to support different kinds of loads), and behavioral (e.g., estimating the lateral stiffness of frames). Approximate methods of analysis plays an important role here.

And how do we represent these different kinds of structural design knowledge? In fact, we use several kinds of representations of the knowledge itself, including: mathematical models for classical and structural mechanics (e.g., partial differential equations, variational principles, etc.); case-specific analyses (e.g., buckling of slender columns); phenomenological, approximate analysis (e.g., the beam-like response of tall buildings); numerical programs (e.g., FEM codes); graphics and computer-aided design and drafting (CADD) packages; rules in design codes (e.g., the AISC code); and heuristic knowledge about structural behavior, analysis techniques, and so forth. Such qualitative knowledge is often subjective and is frequently expressed in rules. Thus, we already employ several different representations or "languages" of knowledge, including verbal statements, sketches and pictures, mathematical models, numerically based algorithms, and the heuristics and rules of design codes. When we use these different languages now, we manage to choose the right one at the right time, but in computational terms we should recognize that it would be desirable to link these different representations or "languages" so that we could model our design process in a seamless fashion. We should also recognize that we often cast the same knowledge in different languages, depending on the immediate problem at hand. For example, a statement (typical of that found in building codes) that the deflection of a floor in a residential building should not exceed its length (in feet) divided by 360 is actually a restatement of equilibrium for a bent beam. If we can externalize this kind of design knowledge into rules embedded in a rule-based expert system, it would be very beneficial for the maintenance and development of the knowledge base.

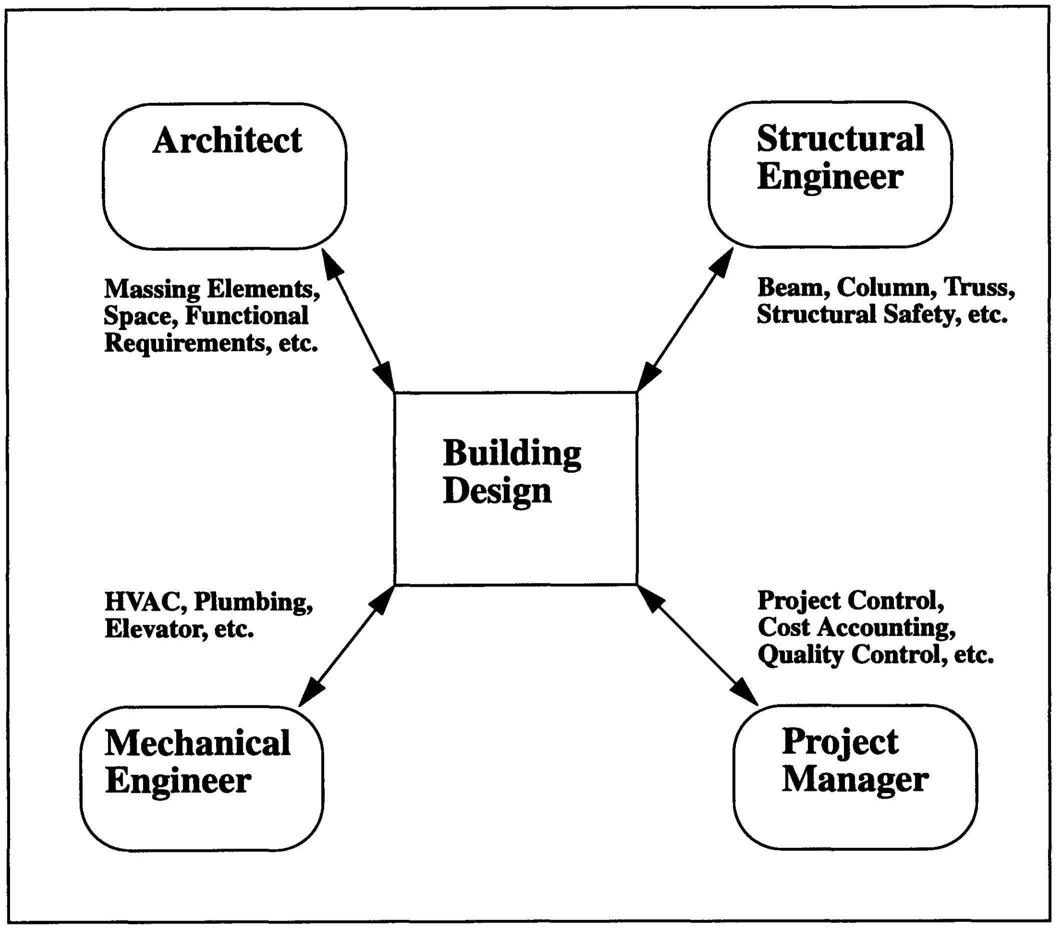

Architect
Structural Engineer
Massing Elements, Space, Functional Requirements, etc.
Beam, Column, Truss, Structural Safety, etc.
Building Design
HVAC, Plumbing, Elevator, etc.
Project Control, Cost Accounting, Quality Control, etc.
Mechanical Engineer
Project Manager

Figure 1.2: Different Engineering Perspectives

1.2 Related Research

1.2.1 CMU Integrated Building Design Environment (IBDE)

IBDE[14] is a testbed for the exploration of integration and communication issues in the building industry. It vertically integrates the various design and planning tasks in the delivery of a constructed facility - from the initial architectural programming through structural and foundation system synthesis and design, on to the planning and scheduling of construction activities. Its major objective is purposely designed to be modular and served as a testbed for the empirical evaluation and calibration of integrated design support environments. There are two significant limitations to emphasize. First, IBDE is not a prototype of a possible commercial building design system. IBDE was conceived as a purely experimental, empirical testbed for the exploration of a host of issues in integration and communication. Secondly, IBDE is not intended to serve as a normative, prescriptive model of how building design "ought to be done." Rather, in the empirical spirit that characterizes the entire project, it is intended to provide a means for arriving at generalizations about the design process that are based on the experience and insights gained from experiments with the system.

1.2.2 MIT Distributed and Integrated Computing Environment (DICE)

MIT DICE [17] project can be envisioned as a network of computers and users, where the communication and coordination is archived, through a global database, by a control mechanism. The definition of an agent in this framework is a combination of a user and a computer. No specific role is assigned to the computer, thus a general approach that allows multiple computer tools is assumed. DICE supplies interface modules that map the representations used by the agents from and to the shared blackboard. In this framework, any of the knowledge modules can make changes or request information from the Blackboard; requests for information are logged with the objects representing the information, and changes to the Blackboard may initiate either of the two actions: finding the implications and notifying various knowledge modules, and entering into the negotiation process, if two or more knowledge modules suggests conflicting changes.

1.2.3 UIUC System Workbench for Integrating and Facilitating Teams (SWIFT)

The SWIFT[25] project is targeted at managing large and complex engineering tasks consisting of many inter-related subtasks whose solutions require a high degree of interaction among people with diverse expertise. It is intended that SWIFT will break the disciplinary, time and geographical boundaries between people who need to work together to achieve a complex task. SWIFT can operate in either single-user mode or with the functionalities provided by ObjectStore database management system. For its initial generation, the SWIFT system borrows this providing an additional proof-of-concept that this architecture is indeed independent of a specific language and database application.

1.3 Roadmap of the Document

The first chapter of this report initiates the discussion of the collaborative engineering design process. Related researches are also introduced and compared.

Chapter 2 is devoted to describing the federation architecture, which is the information infrastructure employed in this research.

Chapter 3 outlines the ideas behind the MIT structural design agent as well as the design scenario and process. Object-oriented concepts as well as the rationale and usage of Object Modeling Language are also described.

Chapter 4 gives the detailed information and experience gained during the entire development process, including the integration of the software modules involved.

Two examples, cabin and firestation design, are presented in Chapter 5 to demonstrate the capabilities of the agent.

Conclusion and future research issues are summarized in Chapter 6.

Appendix A lists the structural design rules in CLIPS. The communication API specification, provided by Stanford CIFE, is listed in Appendix B. The BNF grammars of OML and the associated Instance Manipulation Language are given in Appendix C and D, respectively.

Chapter 2

Federation Architecture

2.1 Introduction

Federation architecture [2] [3] is an information infrastructure that aims at integrating design software by allowing better software interoperability. In this approach, various types of software are viewed as agents that communicate their design information and knowledge using a standard Agent Communication Language (ACL). Agent interaction is assisted and coordinated through system programs called facilitators.

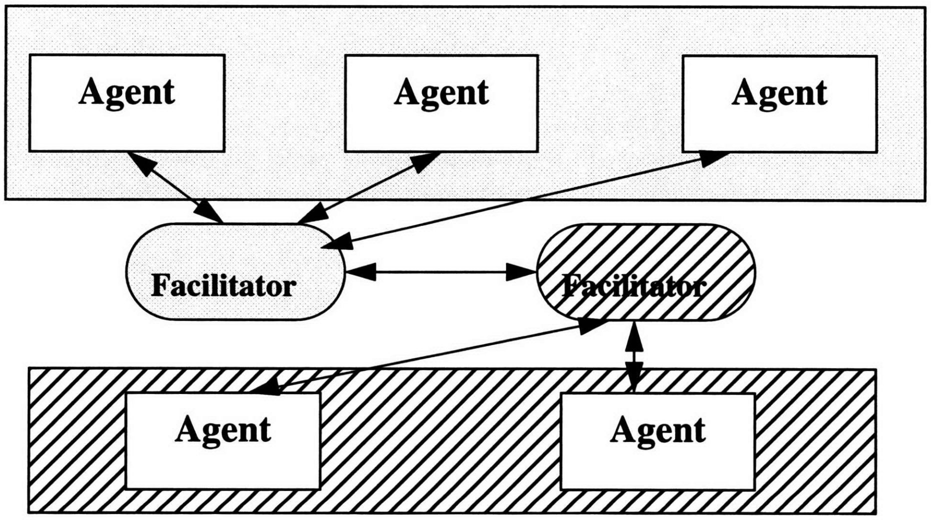


Figure 2.1: Federation Architecture

Agents and facilitators are linked together forming a federation architecture. In this architecture, agents communicate only with the facilitators, and the facilitators communicate with each other. In effect, the agents form a federation in which they surrender their communication autonomy to the facilitators, hence the name of the architecture. One of the primary advantages of the federation architecture is improved flexibility in integrating agents, which allows different agents to be ignorant about other existing agents in the architecture.

Communication in the federation architecture is undirected. Agents communicate their design information in the form of messages without specifying destinations or addresses for those messages. They also specify interests to designate their desires for information produced by other agents. It is the responsibility of the facilitators to determine the appropriate recipients of the messages based on these interests and forward the messages accordingly. In performing this responsibility, facilitators handle a number of important operations to facilitate communication and the exchange of design information and knowledge among agents. They translate message, schedule the executions of different agents, help to decompose the problems into sub-problems. and assist in relating different design information.

It is important to note that the agent-based approach to software interoperability is based on the idea of communication standards. While this idea has been the basis of many specific application areas (e.g., mail standard SMTP for electronic mail, standard formats like GIF for graphics), the novelty of this approach lies in the language used and in pushing the idea of communication standard to its limit to improve the interoperability of software.

2.2 Agent Communication Language

In the federation architecture, agents communicate in a language called Agent Communication Language (ACL). This language provides a unified format by which agents unambiguously exchange complex expressions describing various aspects of design. This language follows from the decisions of a series of meetings of project participants. Besides, Object Modeling Language (OML) [20] has been chosen as the standard object-oriented representation language for this project.

2.2.1 Message Characteristics

Each message in ACL consists of a sequence of expression enclosed in parenthesis. The first element in any sequence in the message type indicating the communication mode (e.g. declarative, interrogative, imperative). Different communication modes are expression by different Knowledge Query Manipulation Language (KQML) message types. These details have been encapsulated in the API developed by the Stanford team. By calling this API, the specific design agent is able to communicate with the facilitator to transmit and exchange the design information with other design agents.

2.2.2 Object Modeling Language

The second principal element in an ACL message consists of expressions in the Object Modeling Language (OML) developed by Carnegie Mellon University team. OML is an object-oriented language with high-level constructs and superior modeling semantics. The details of OML is delineated in Section 4.2.

2.3 Agents

An agent is a software program capable of communicating with other software using ACL. In principle, it is possible for an agent to interact directly with other agents. However, in a pure federation architecture, all communication takes place between an agent and its local facilitator.

There are two phases in the operation of an agent - its initialization phase and its normal operation. During the initialization phase, the agent declares it existence and asserts various aspects of its specification to its facilitator. The specification of an agent is a collection of commands created in OML that describe the agent's interests and its perspectives, and characterize its behavior. The interests of an agent are those messages that the agent desires to receive and to be informed about. The perspectives specify the formats in which the agent would like to receive certain information. The behavioral part of an agent's specification characterizes the agent's activity. For the purpose of agent-based software engineering, the most important part of this activity is the agent's reaction to the messages it receives.

Once the initialization phase is complete, the agent enters a normal operation. Within an agent, typically, there is a loop that detects different events (e.g. mouse clicks, key strokes) and handles them by calling appropriate event handlers. **Message** from the facilitator are treated as events of this sort. When a message is received by an agent, it is processed by its message handler which invokes the associated callback function. Depending on the agent's role, the agent may request further information or services or may generate information and send it to the facilitator.

2.4 Facilitator

In the federation architecture, a facilitator can be viewed as a system program that provides services for agent collaboration. Agents are free to send messages without specifying the destinations of these messages. It is the responsibility of the facilitator to determine the appropriate recipients for undirected messages and translate the messages in the language of the recipients and to forward the messages accordingly. This service is based on the agents' interests.

Another important service provided by the facilitator is translation, which is the transformation from one form to another. One of the important aspects of translation is vocabulary translation. The need for vocabulary translation arises because of differences between the abstractions inherent in the implementation of agents. Other types of translation, which involve deducing new design information (needed by some agents) based on other information, may be necessary to support agent collaboration.

A third important service is the scheduling of agent activities based on their behavior. Depending on the different agent's roles, scheduling can be viewed from different perspectives. In some cases, scheduling can merely be forwarding messages to different agents in a priority order. In other cases, scheduling may include time allocation of jobs on each processor. The facilitator is an agent that provides some services to other agents in an environment and performs certain functions.

A facilitator accomplishes the tasks of performing domain-independent automated reasoning for facilitating and coordinating the interactions of agents in an environment by receiving and distributing appropriate messages to agents according to their interests, scheduling agent activities and performing different types of vocabulary translations. Most of the functions performed by the facilitator are based on agents' specifications communicated by agents at the registration phase. Other types of vocabulary translation are performed to facilitate the execution of design tasks performed by autonomous design agents. This translation is done by capturing domain-specific knowledge about different facility design tasks and their interrelationships in the form of sets of KIF axioms constituting KIF theories. Provided with these theories, the facilitator performs reasoning to infer implicit design information received from information received from design agents and expresses it explicitly for the use of other agents. The facilitator also reasons about the relationships among different level of design abstractions and different design models describing the same physical entity. In this way, the facilitator is capable of identifying appropriate design changes and of forwarding them to appropriate agents based on some other changes.

The operation of the facilitator in such an environment is continuous. At each point of its operation, the facilitator checks for agent registration and for messages with already registered agents. When the facilitator detects that an agent is requesting registration, it executes its registration protocol for accepting such a registration. Upon registration, an agent will typically send it specifications (including interests, its perspectives, and its behaviors) to the facilitator so that it becomes identifiable in the environment. Once the agent is registered with the facilitator, it can send messages and receive messages about the evolving design according to its specifications.

Every message received by the facilitator is processed in an automated fashion and a number of appropriate messages are forwarded to various registered agents. In processing a message, the facilitator first identifies the message type and then performs a perspective translation according to the agent's perspective of the information contained within the message. Next, the facilitator reasons about this information to infer other appropriate information. Then, the facilitator attempts to find agents interested in the information and message types based on the agents' interests. Once all interested agents are identified, the facilitator prioritizes the order in which messages will be forwarded to the interested agents' behaviors. Finally, in the order determined, the facilitator performs a perspective translation for every agent and forwards appropriate messages to it.

2.5 API Specification and Usage

The proposed API provides agents with two sets of methods that would enable them to communicate with the facilitator in the form of ACL messages. The API hides the details of ACL from programmers by providing an object oriented program interface. Using this API, programmers deal with two classes that are used to send and receive messages from the facilitator.

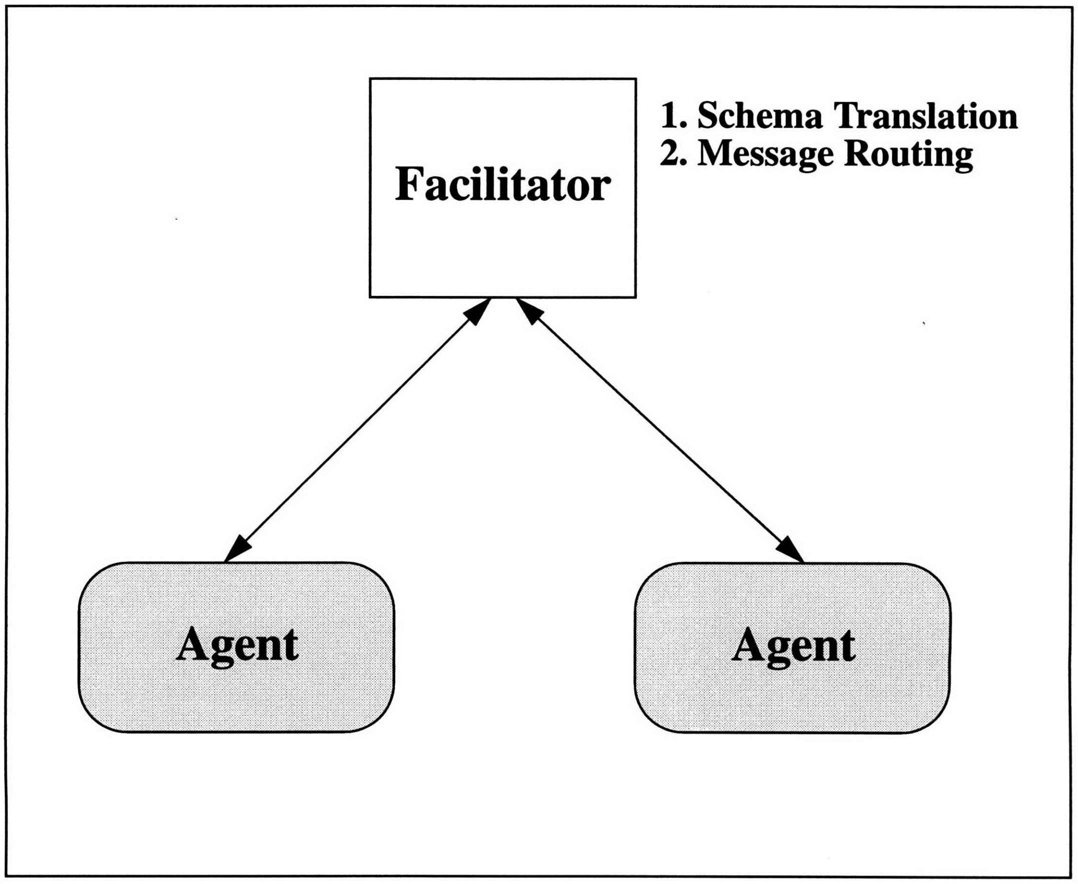


Figure 2.2: Facilitator

The first class, **Facilitator**, provides a set of methods that can be used to deal with all the connections, disconnections, message sending, and message receiving. The second class, **Message**, provides a set of methods that can be used to interrogate the message type, its class name, its object name, its slot name, its value, and its relation.

Agents communicate with the **Facilitator** by first creating a **Facilitator** object. Then, agents must call connect as the first operation on this object. Following this operation, an agent typically would communicate its interests by invoking the appropriate methods associated with the **Facilitator** class. Agents would then be free to send different types of messages to the **Facilitator** by invoking methods on the **Facilitator** object. Agents would also receive messages from the **Facilitator** according to the interests they communicated to the facilitator in the form of **Message** objects. Agents can interrogate these message objects to identify their types and contents to take the appropriate actions.

The API has restrictions on the different data types and object name convention. It is important to note that the case of different names (e.g., class name, slot name, etc.) is not maintained because of the underlying implementation of the facilitator. For this reason, all names obtained by interrogating a message object will be upper case. In addition, the use of colon ':' in any name is not permissible for the above mentioned reason. However, colon ':' can certainly be used in a string value as discussed below.

The API supports different data types including integer, floating point, string, 2D point, 3D point, interval, polygon, and matrix. Agents obtain slot values as a character string by interrogating the message object. The character string will conform to the following grammar.

```
<value>:
   <simple-value-list>
  | <structured-value-list>

<structured-value>:
   `{' <simple-value-list> `}'
  | `[' <simple-value-list> `]'
  | `{' <structured-value-list> `}'
ring will conform to the following grammar.
| `[' <structured-value-list> `]'

<structured-value-list>:
   <structured-value>
  | <structured-value-list> <structured-value>
<simple-value-list>:
```

<simple-value> | <simple-value-list> <simple-value>
<simple-value>:
<string> | <integer> | <real> | NIL | <variable>

Examples:

Integer value: 1
Floating point value: 1.0
String value: "Column"
2D-Point: {0} {0}
3D-Point: {0} {0} {0}
Polygon: [{1} {2} {3}] [{4} {5} {6}] [{7} {8} {9}]
2x2 Matrix: {{1.0 2.0}} {3.0 4.0}}
Interval: {1 4}

The methods embedded in facilitator and message object are outlined as follows. Appendix C contains the full specification in details.

Facilitator Methods

bool **Facilitator**::connect()
bool **Facilitator**::disconnect()
bool **Facilitator**::forget()
bool **Facilitator**::set_name_space()
char** **Facilitator**::get_connected_agents()
char** **Facilitator**::get_all_agents()
bool **Facilitator**::message_waiting()
Message* **Facilitator**::get_message()
bool **Facilitator**::commit()
bool **Facilitator**::abort()
bool **Facilitator**::construct_instance()
bool **Facilitator**::destruct_instance()
bool **Facilitator**::add_value()
bool **Facilitator**::modify_value()
bool **Facilitator**::delete_value()
bool **Facilitator**::add_relation()
bool **Facilitator**::delete_relation()
bool **Facilitator**::int_all()
bool **Facilitator**::int_construct_instance()
bool **Facilitator**::int_destruct_instance()
bool **Facilitator**::int_add_value()

bool **Facilitator**::int_modify_value()
bool **Facilitator**::int_delete_value()
bool **Facilitator**::int_add_relation()
bool **Facilitator**::int_delete_relation()

Facilitator Enumerated Types

enum int_action { k_int_insert, k_int_remove };
bool **Facilitator**::register_callback()

Message Methods

int **Message**::type()
char* **Message**::msg_class()
char* **Message**::object()
char* **Message**::slot()
char* **Message**::value()
char* **Message**::child_class()
char* **Message**::child_object()

Example of Use

This is an example of an agent connecting to a facilitator, sending messages to it, and then receiving messages and processing them.

```
#include "ACL.h"
int main()
Facilitator* f = new Facilitator; // create the facilitator object
f->connect("acl-facilitator", "acl-facilitator", "hpdce.stanford.edu", 4010);
f->construct_instance("Column", "col1");
f->add_value("Column", "col1", "name", "\"column1\"");
f->add_value("Column", "col1", "x_coord", "0.0");
f->add_value("Column","col1","y_coord","0.0");
f->add_value("Column","col1", "z_coord","0.0");
f->commit();
   while ( f->message_waiting() ) {
      process_message(f->get_message());
   }
   f->disconnect();
   return 0;
}
```

Implications of this Design

Agents are responsible for doing any necessary mapping from names to classes, objects, or slots. It is up to the receiving Agent, for example, to determine that a class_name of “column”, object_name of “Col-1”, and a slot_name of “x_coord” refer to the x coordinate of column object Col-1.

2.6 Advantages and Disadvantages

The major advantage of the capabilities of API is its simplicity. It is quite easy for the programmer to capture the gist for its usage. On the other hand, there are a set of limitation that constrain the current API implementation and facilitator. The detailed discussion is given in the last chapter.

Chapter 3

Characteristics of Structural Design Agent

3.1 Introduction

In this chapter, the concepts behind the structural design agent are presented. These includes object-oriented programming, structural engineering vocabulary, and design scenario and process.

In our definition, a computer-aided structural design agent would organize, process, manage, and communicate with other agents the design information associated with complex design problems, manage the overall process, and make decisions whenever necessary. The major difficulty in creating such systems is that the engineering design process is not clearly understood and the complexity in communicating and negotiating with other agents. The problems are (1) the structure of the process, (2) the tasks to be performed, and (3) the information required to carry out these tasks are not well defined. In addition, We lack a robust and versatile data structure which can seamlessly represent complex design process as well as design artifact information. In order to address this difficulty, a conceptual framework is needed to help formalize the design process.

In terms of structural engineering, the framework needed for computer integrated design include (1) an organizational model for the design process, (2) a set of well-defined design tasks, and (3) a set of formalisms for representing and processing the knowledge required for these tasks. This chapter presents an object-oriented data model for structural engineering design. The scope is limited to routine structural design in which conventional structures are design using standard components (e.g., beam, column, truss, etc.) and clearly defined design procedures such as those outlined in various design code.

Design tasks ranging from the selection of structural concept through the completion of a detailed design sizing are considered. From our perspective, the representation is the key issue. It is not that problem solving and evaluation are less important; they are extremely important, but they too must be expressed and implemented at an appropriate level of abstraction. Thus, they are also inextricably bound up with concepts of representation.

Structural engineers involved in building design encounter a wide variety of building types, which vary from one another in their topology, their construction material and in their load resisting systems. They start their work defining minimum information as input data so that any type of building can be designed. The computer model identifies, at a high level of abstraction, the objects and their interrelations involved in the design of any structure. As design proceeds, increased detail is generated. The objects and relations are derived directly from the modeling of a structural design, which is a necessary part of the design process whether it is done manually or with the aid of a computer.

We begin with an outline of basic concepts involved in object-oriented methodology, with emphasis on structural engineering, as well as the importance of information modeling. This is followed by the description of a proposed model for a generic structural engineering vocabulary.

3.2 Object-Oriented Methodology [26] [27]

From the beginning, the design of the structural agent has been centered on object-oriented concepts. Object-Oriented programming (OOP) is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

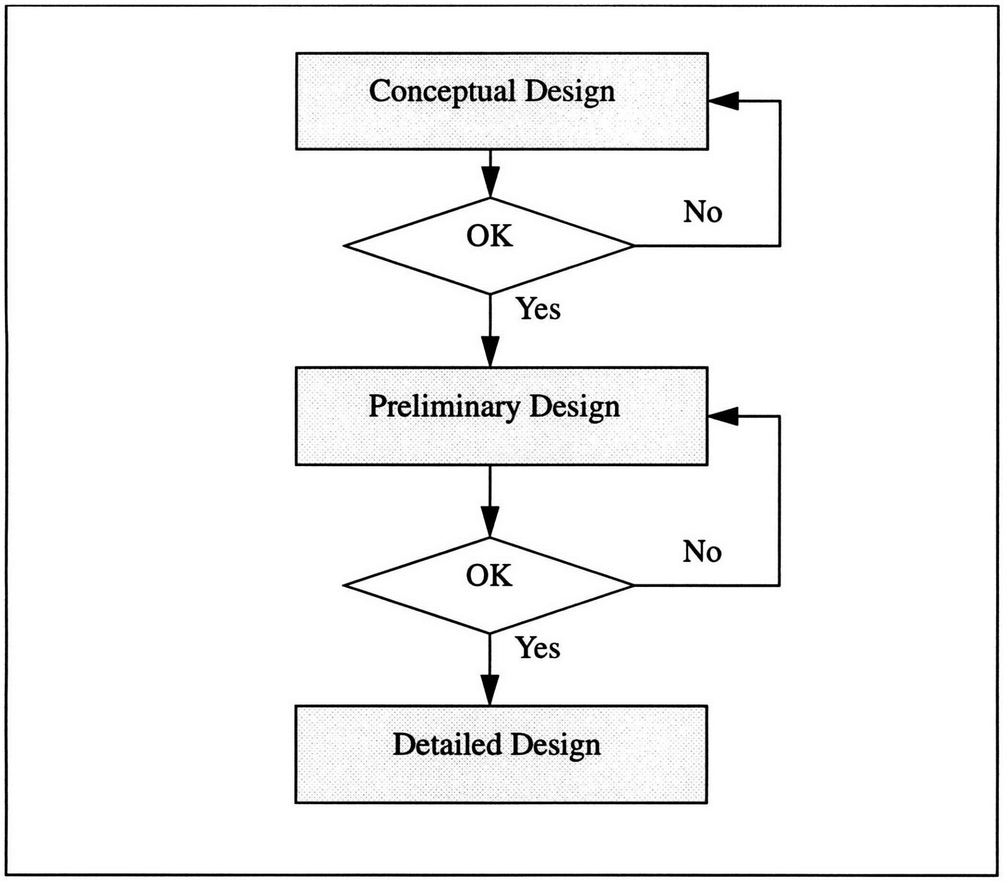


Figure 3.1: Structural Design Process

There are three important parts to this definitions: object-oriented programming (1) uses *object,* not algorithms, as its fundamental logical building blocks (the "part of" hierarchy); (2) each object is an *instance* of some *class*, (3) classes are related to one another via *inheritance relationships*, and (4) objects have behavior so that functionality can be directly associated with the class to which an object belong. A program may appear to be object-oriented; but if any of these elements is missing; it is not an object-oriented program. Specifically, programming without inheritance is distinctly not object-oriented.

Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design. There are two important parts to the above definition of OOD.

- leads to an object-oriented decomposition
- uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system, in addition to the static and dynamic aspects of the system.

The support for object-oriented decomposition is what makes object-oriented design quite different from structured design: the former uses class and object abstractions to logically structure systems, and the latter uses algorithmic abstractions. The object-oriented data model essentially supports the following two relationships between data types.

Aggregation relationship. Unlike the relational model, the object-oriented approach allows the value of an attribute to refer to another class of objects. This permits the establishment of a directed graph structure for the classes in the data model. This relationship allows the specification of complex objects by establishing "part-of" links between the classes in the data model.

Generalization relationship. This relationship connects a class and its direct and indirect subclasses. The generalization relationship establishes a class hierarchy that describes the "is-a" type connections between the classes in the data model. The class hierarchy establishes one mechanism for inheritance of attributes from superclasses.

The three major pillars of object-oriented technology are inheritance, polymorphism, and information hiding. With these three mechanisms, the software development and maintenance can be done with more robust framework and better integrity.

3.3 Object Modeling Language (OML)

In order to simulate the semantic-rich design objects, the Object Modeling Language [20] [21] is chosen as the standard representation and information modeling language for different design agents. OML is defined and utilized in the context of satisfying the needs of ACL project participants and taking advantage of rapidly growing object-oriented technology. In OML terminology, the computer tools that send, receive, and process data at the different geographical sites trying to accomplish design tasks are usually called *agents* in the collaborative design literature. These agents are implemented on machines with different architectures, written in different programming languages, and based on different conceptualizations of their respective domains, which include different assumptions about the way in which they interact with users. During collaboration, these users may interact with other users through their respective agents or directly through parallel communication mechanisms such as telephone or video. Major technical and conceptual issues that have to be resolved if this type of work setup are the following:

- The packaging and unpackaging of data reflecting heterogeneous machine architectures.
- The syntax and semantics of the language in which data are transmitted. This will require that the agents phrase and transmit their messages in terms of a shared vocabulary or with the use of translators, which in turn, reflect shared notions and concepts.
- A communication mechanism that allows new agents to be added with ease and provides selective (content-based) routing; that is, a message posted by one agent (for-example, about a recent design change) is only sent to agents that are interested in that type of change.

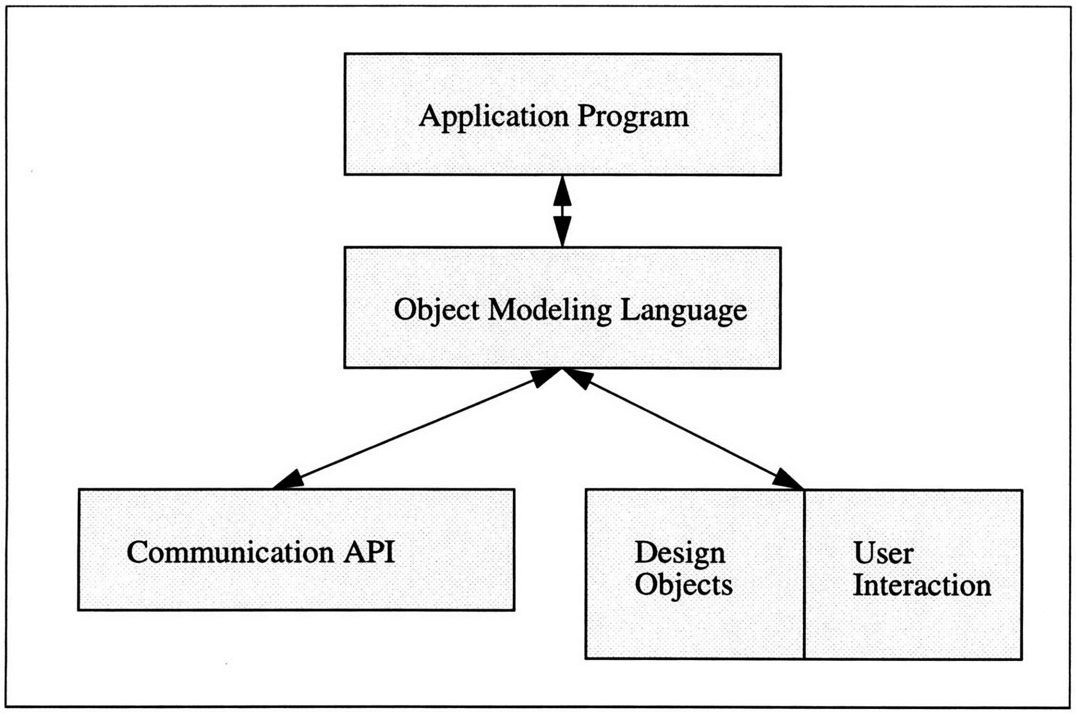


Figure 3.2: Relationship of OML, API, and Application Programs

• Coordination between transmissions, for example, a request for a piece of information or a design change must not only be received, but answered in a timely fashion by the appropriate agent, which may actually operate in a different time zone. Complexity is added to this exchange between agents if, at the same time, also the users interface with each other, with their own agent, and through these, with the other agents.

• The storage and retrieval of design information that are of interest to more than one agent, including the management of different design versions.

A basic assumption underlying the ACL project is that the individual agents maintain their own internal representations and generate from these the views most appropriate to the tasks they execute. Most of the agents participating in the ACL project use an object-centered representation of design information that is rich in content and highly structured;

that is, relations between objects are of prime importance. These representations are consistent with the vast majority of design representations that have been developed throughout computer science, the engineering disciplines, and architecture.

Given such a distributed representation, the individual agents are responsible for capturing design updates or changes made by other agents in their own internal representation and for maintaining consistency with the other agents' representations. This configuration reduces drastically the transaction and view problems that arise when agents share a common work space or global design representation.

However, it also raises several issues, some of which are addressed in this chapter. It is first of all clear that each description of an object that is communicated between agents must contain the class to which the object belongs: an agent simply has to know if an object is a room, a column, etc. Furthermore, an agent has to know the superclasses to which an object belongs. For example, if an agent want to design a connection between two structural members and has received information about objects that belong to classes like 'column' and 'beam', it has to know that all of these belong to the superclass 'linear member' and can thus be connected together with only one joint. Furthermore, the class/superclass relations must be shared consistently between agents. If we share this information, we may as well share more general inheritance relations because this allows us to inherit attributes, which greatly reduced the amount of data that have to be passed along in object descriptions.

The information received by any agent must be interpreted by this agent so that it is semantically consistent with the interpretation of the sending agent. Specifically, relationships between objects that are not implied by inheritance must be communicated between agents.

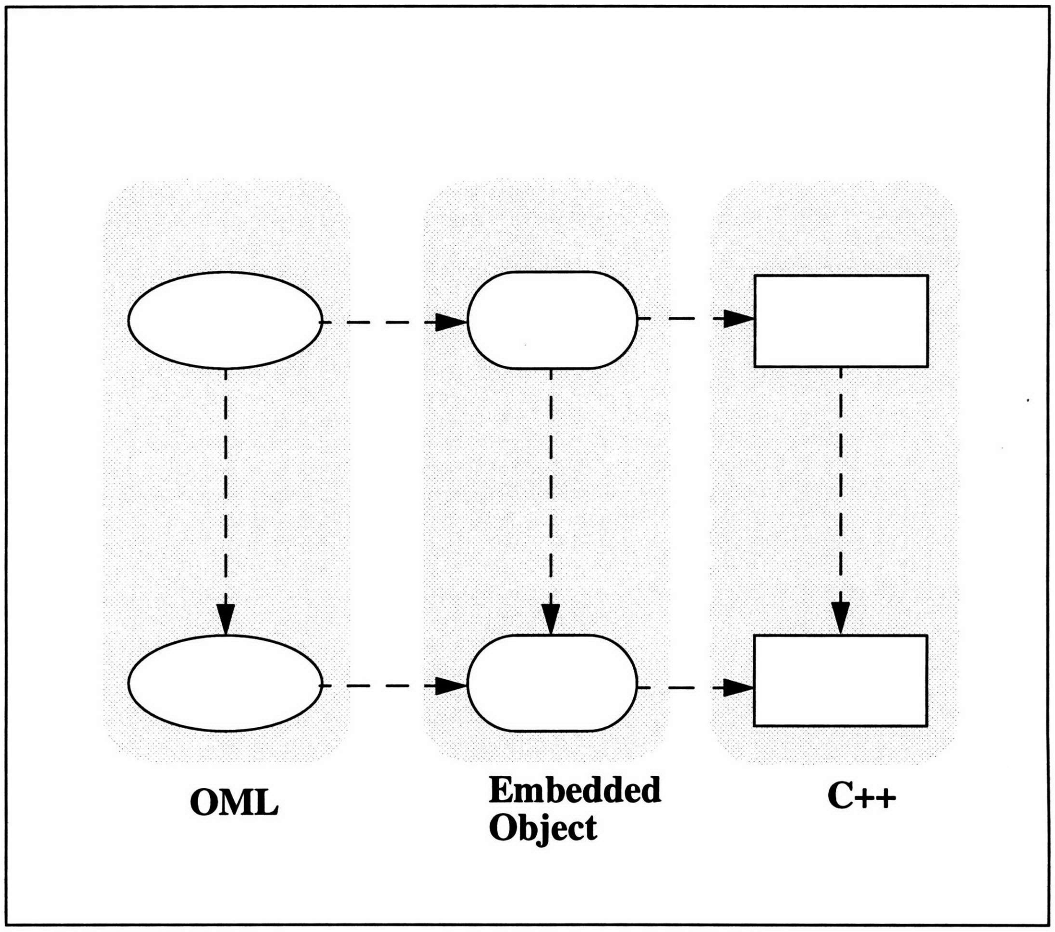


Figure 3.3: Binding Object Relationships [20]

For example, if an object *a* is a part of an object *b* and the geometric attributes of *a* are relative to a local origin established by *b*, the part of relationship cannot be recognizedby an agent that did not create the objects and their relationships. A remote agent is then unable to locate the object correctly in its own coordinate space; that is, this part-of relationship has to be communicated explicitly.

This may be advisable even for relationships that can, in principle, be rediscovered at remote sites by reasoning with communicated attribute values (like geometric coordinates) because it may reduce the computational loads on the agent at the remote site and, more importantly, be an important indicator of the intent of the agent who created the relationship. How else would we be able to distinguish between intended and accidental relationships? For example, more rooms are usually adjacent to each other on a floor than required for functional reasons, and an agent trying to rearrange some of these rooms may need to know which of the existing adjacencies can be ignored.

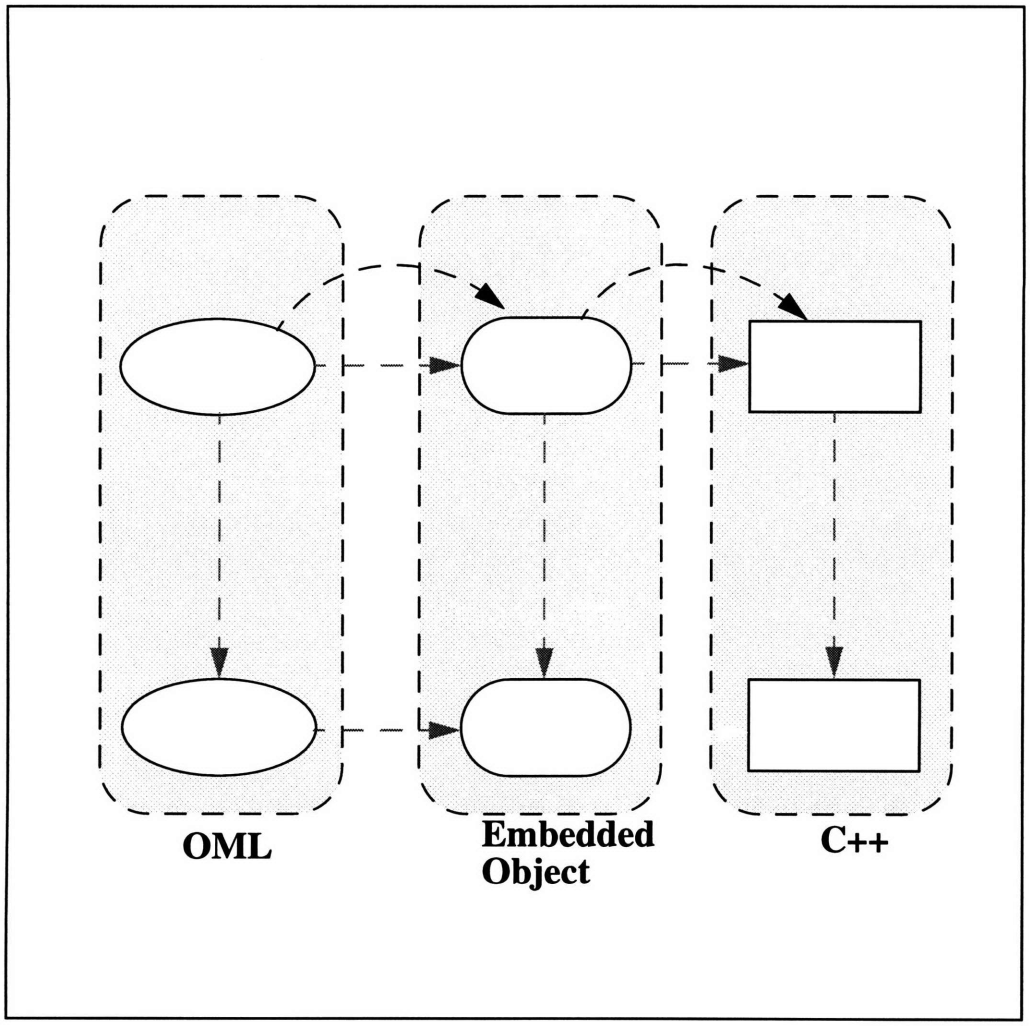


Figure 3.4: OML Event Origination [20]

Furthermore, an understanding of the concepts and relationships needed at any specific site is evolving and never stable: the attributes of an object may change, or new objects or relationships may be introduced. This means that the translators between object representations are also evolving, which creates the need to maintain semantic consistency between these evolving translators across agents at all times. This suggests that the maintenance of the translators should not be left to the individual sites and manual procedures, but be governed by a shared methodology that automates as much of the process as possible, including the automatic generation of parts of the translators using algorithms that are known to produce correct results.

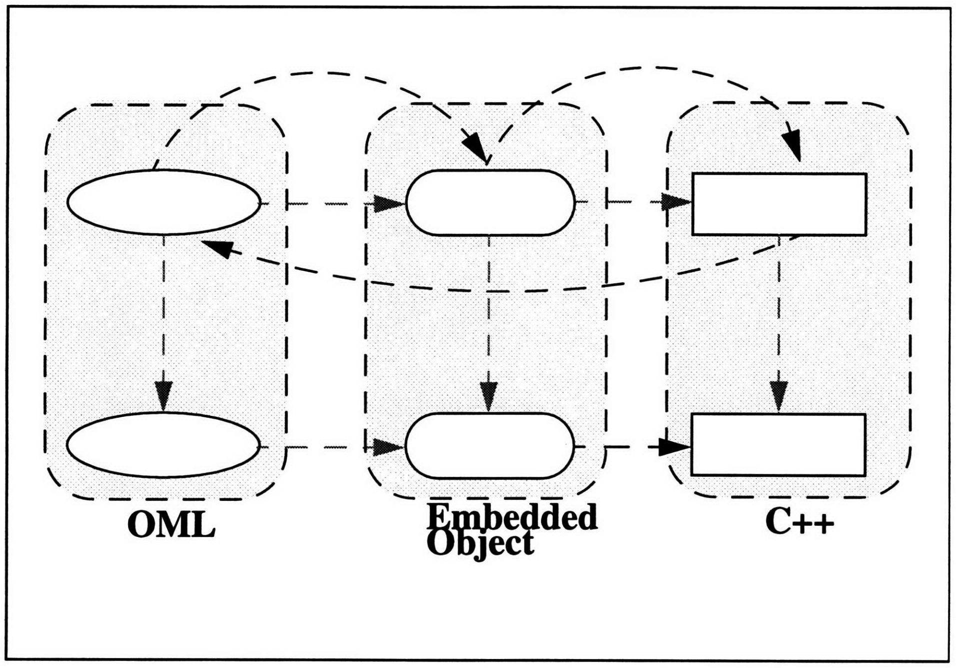


Figure 3.5: OML Event Generated via an Application [20]

This section addresses these issues and proposes a methodology that resolves them. We mention in passing some other problems that the ACL team has to face as a result of the selected communication approach.

- If a design elaboration or change made by one agent conflicts with the goals pursued or requirements maintained by another agent, these agents must negotiate about how to resolve the conflict.

• It is not clear at any time which parts of the distributed representations constitute the current version of the design or any other version; likewise, it is not immediately clear when a design is finished. Manipulation of change and status is solely left to each agent.

The OML is proposed under this circumstance, as a method for communication of software objects and their relationships between agents and the syntax and semantics of the communication language. OML provides a reasonably precise definition of an object model specification language and its intended semantics. The language syntax is specified in an BNF-style, which facilitates the development of translators for the language. For example, specifications written in this language can be translated to object database classes, C++ classes or to KIF sentences (first order logic). The grammar specification is presented along with examples of how it is intended to be used.

The ACL project team members were determined to use the schema translation method because they assumed from the start that a common vocabulary cannot be generated for this project and is in any case not desirable. The team members decided furthermore to use an augmented Federation Architecture for this project, where a facilitator provides schema translation. These object models are based on a shared syntax and semantics of schema definition, but each model needs to represent only what input it needs *(interests)* and what outputs it can produce *(capabilities)*. In addition, the object model provides the mechanisms needed to bind objects in the model to a programming language such as C++ or Lisp. An important advantage of this architecture is that it can be formally specified in a computable model and therefore formally verified.

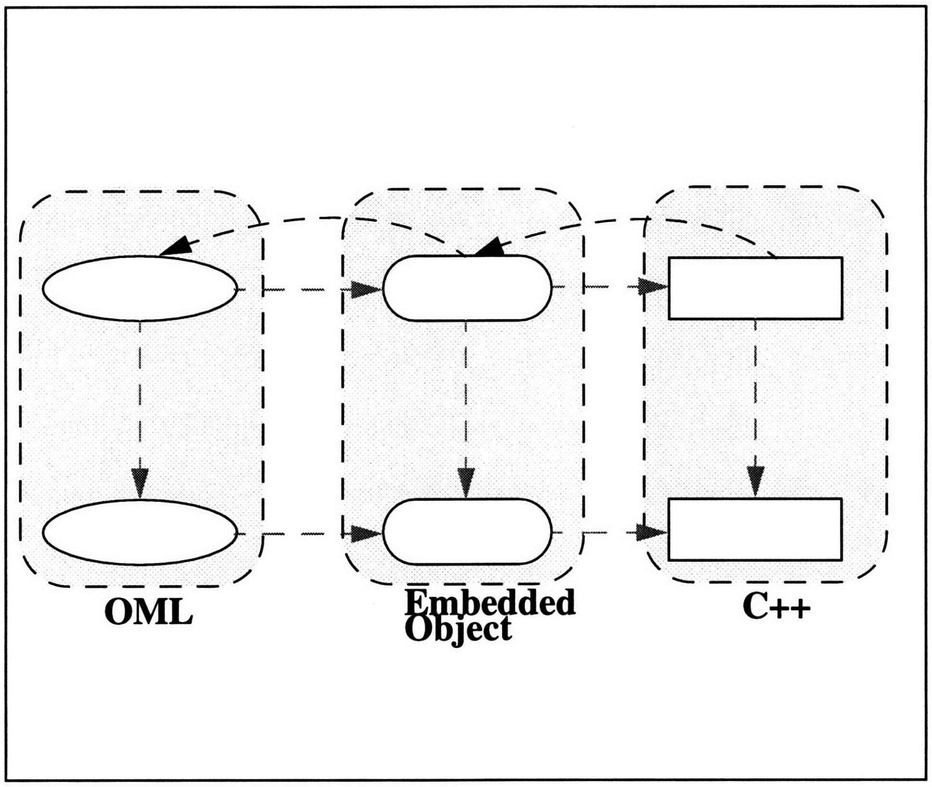


Figure 3.6: Application Event Propagated to OML [20]

The object model contains both classes and instances. However, a language binding can be developed that would not store instances in the model: only classes are stored, while changes to instances are mapped directly to the bound language.

3.4 Structural Engineering Vocabulary [1] [12]

In the past, structural engineers used computers mainly for only analysis portion of the detailed design process. Member sizing and conformance checking were done by hand, and previous experience and design heuristics were relied upon heavily. As computers became more prevalent, software become available which helped to reduce the drudgery of code conformance checking and member sizing. Today there are software packages available which provide analysis, member sizing and conformance checking for particular types of structures, e.g., steel or reinforced concrete frames. In addition, there are graphical routines for pre- and post-processing the data, thereby allowing engineers to verify topology and loadings, and to view the structural response, such as moment and shear diagrams and deflected shapes.

Looking at the entire structural design process, we see data generated in increasing detail during a process of hierarchical detailing. A rough conceptual design leads to the identification of major components. When the configuration is accepted, these components are further detailed, leading to components of components, and so on. This process would normally lead to an explosion of detail if the designer performed a complete enumeration of objects. Designer, however, recognizes the economic need for repetition. Thus they select typical components, and design these components for use within other components multiple times. Then when those components are further detailed, only a single copy of each is further disaggregated.

The Structural Engineering Vocabulary is proposed to satisfy these requirements. The class **Struct_system** is the base class for all structural components. Each structural component represents a physical object which can be designed and has associated with it a local geometric description. This object maintains links to where the system is used, to a material object, to a loading case object, and to its components. Its components may be the same class or subclasses. Since the **Struct_system** class is recursively nested in both generalization and aggregation relationships, it can contain several **Struct_system** instances (e.g., beam, column, truss) to form the hierarchy. For example, a bent has one truss, two columns and two footings which are all **Struct_system** instances in this context.

We can define and simulate behaviors of the **Struct_system** object through a set of methods. For instance, if would like to know the stiffness of a column member, we can define a methods as follows:

double Column::Compute_Stiffness();

which returns the stiffness of the current column that the system is designing. Through the information hiding and encapsulation, the designer can easily understand the state of the structural component without looking up tables or computation.

More often than not, the structural engineer prefers to put a particular physical component in two or several distinct locations if the loading conditions are about the same. This data model is very suitable for supporting this function. The rationale behind this design is to avoid the enumeration and duplicative detailing. Take the cabin design as an example, two of the four bents used are exterior bents, while the other two are interior ones. In this case, the engineer can simply design an interior bent and an external one, according to different loading criteria, and put each in two different locations. The major advantage of this approach is no data redundancy or duplication so that the data management and consistency are more robust than with an enumeration approach. As the number of design objects grow, our approach proves to be a more efficient solution.

Furthermore, each object which can be designed has attached to it with a set of design rules written in the CLIPS productions. During the design process, when the designer trigger or notify the agent to perform the design tasks, the program will call an inference engine and use each object's rules to automatically select components for an object and walk down the hierarchy invoking the associated rules embedded within the component objects to perform more detailed design.

If the designer feel that it is necessary to perform the redesign (e.g., the architect sends some modifications or the structural engineer is requested to conform to specific kind of standard), he/she may modify, with the aid of object structure browser, the slot value of an arbitrary object and the agent will perform the redesign of those objects below the current node in the aggregation hierarchy. For example, if the designer changes certain values in the Bent object, the agent will mark those objects contained (e.g., column, truss, and footing,) are required to be redesigned.

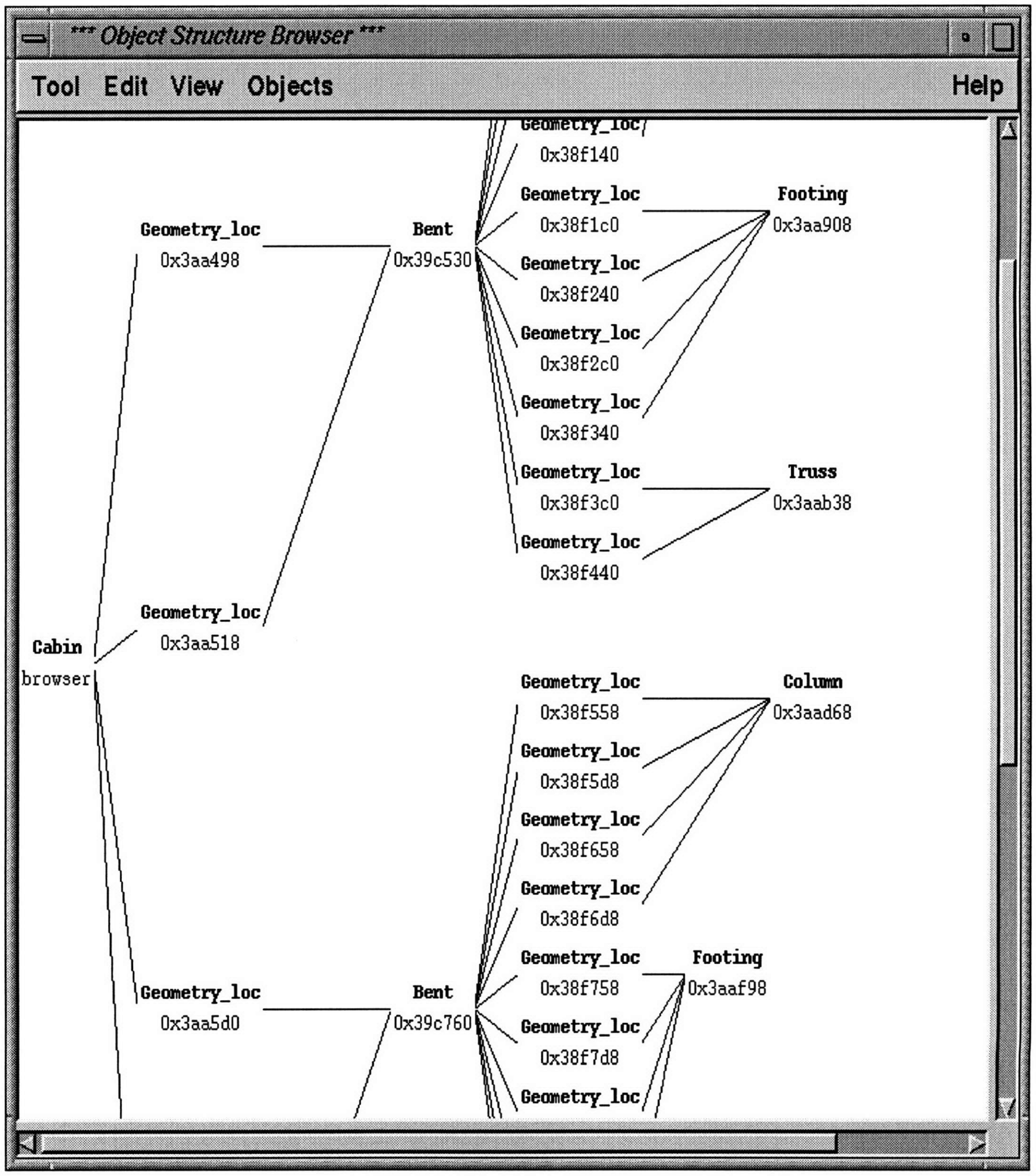


Figure 3.7: Object Relationships

The hierarchical relationships between design objects and their geometric information is displayed in the object structure browser (see Figure 3.7). According to the loading criteria, there are two kinds of bent designs in this case: one is an interior bent, and an exterior one. We associate each bent with two geometry objects, which means that we use the same design in two different places to take advantage of the symmetric distribution of loads. The same rules apply to the components contained in the bent object. The entire framework provides a uniform representation of structural elements. Practical design examples are presented in Chapter 5.

Structural Engineering Vocabulary in OML

```
// Domains
    DECLARE DOMAIN double number(10,4) SINGLE;
    DECLARE DOMAIN String char(256) SINGLE;
    DECLARE DOMAIN int number(10) SINGLE;
    DECLARE DOMAIN mm number(10,4) SINGLE;
    DECLARE DOMAIN mm_square number(10,4) SINGLE;
    DECLARE DOMAIN StringList char(256) MANY;
    DECLARE DOMAIN mpa number(10,4) SINGLE;
    DECLARE DOMAIN kilograms number(10,4) SINGLE;
    DECLARE DOMAIN newton_mm number(10,4) SINGLE;
    DECLARE DOMAIN point-2d number(10,5) MANY;
    DECLARE DOMAIN point-3d number(10,5) MANY;

    DECLARE DOMAIN dimension number(10,5) MANY;
    DECLARE DOMAIN direction number(10,5) MANY;
    DECLARE DOMAIN restraint number(10,5) MANY;
    ---------------------------------------------------------------------------
    // Relationships
    DECLARE RELATIONTYPE has_a LINK;
    DECLARE RELATIONTYPE hasmany VECTOR;
    DECLARE SUBDOMAIN deg double RANGE 0.0 360.0;
    DECLARE SUBDOMAIN material String ONEOF concrete steel timber;
    DECLARE SUBDOMAIN load_type String ONEOF point linear area;
    DECLARE SUBDOMAIN slab_type String ONEOF waffle ribbed flat;
    DECLARE SUBDOMAIN connect_type String ONEOF fixed hinge roller;
DECLARE SUBDOMAIN wall_type String ONEOF brick rc;
------------------------------------------------------------------------------
// Geometric definition
------------------------------------------------------------------------------
DECLARE FORWARD CLASS Struct_system;
DECLARE CLASS Geometry_loc I/O SUBCLASS OF Object (
    VALUE name String, //name
    VALUE origin_x double,
    VALUE origin_y double,
    VALUE origin_z double,
    VALUE orient_x double,
    VALUE orient_y double,
    VALUE orient_z double,
    RELATION parent has_a Struct_system, //point to parent structure
    RELATION design has_a Struct_system//use a particular design
);
------------------------------------------------------------------------------
// Design Description Classes
------------------------------------------------------------------------------
DECLARE CLASS Material I/O SUBCLASS OF Object (
    VALUE name String, // steel concrete timber
    VALUE elastic_modulus double,
    VALUE poisson_ratio double,
    VALUE weight_density double,
    VALUE mass_density double,
    VALUE alpha double // coefficient of thermal expansion
);
------------------------------------------------------------------------------
DECLARE CLASS Loading_condition INPUT SUBCLASS OF Object(
    VALUE name String, //LC1, LC2,...
    RELATION loads hasmany Load, //combination of various loads
    VALUE number_of_loads int //number of loads
);
------------------------------------------------------------------------------
DECLARE CLASS Load INPUT SUBCLASS OF Object(
    VALUE name String, //snow, live, dead, wind, earthquake
    VALUE dimension dimension,
    VALUE origin_x double, // relative to parent
    VALUE origin_y double, // relative to parent
    VALUE origin_z double, // relative to parent
    VALUE direction_x double,
    VALUE direction_y double,
    VALUE direction_z double,
    VALUE magnitude double,
    VALUE unit-measure load_type //point, line, area
);
--------------------------------------------------------------------------------
DECLARE CLASS Struct_system I/O SUBCLASS OF Object (
    VALUE name String,
    RELATION structural_system hasmany Struct_system, // multiple struct_sys
    RELATION components hasmany Geometry_loc, // multiple components
    RELATION parent has_a Struct_system, // point to parent structure
    RELATION material hasmany material,
    RELATION loading hasmany Loading_condition,
    RELATION uses hasmany Geometry_loc,
    VALUE number_of_loading_condition int,
    VALUE number_of_structural_system int,
    VALUE number_of_components int, // number of components
    VALUE dimen_x double,
    VALUE dimen_y double,
    VALUE dimen_z double,
    VALUE self_weight kilograms // self-weight of current structure
);
--------------------------------------------------------------------------------
DECLARE CLASS Joint PRIVATE SUBCLASS OF Struct_system(
    VALUE name String,
    VALUE restraint restraint // 6-tuple array
);
--------------------------------------------------------------------------------
DECLARE CLASS Linear_mbr PRIVATE SUBCLASS OF Struct_system (
    VALUE cross_section_area mm_square
);
--------------------------------------------------------------------------------
DECLARE CLASS Area_mbr PRIVATE SUBCLASS OF Struct_system (
    VALUE cross_section_area mm_square
);
--------------------------------------------------------------------------------
DECLARE CLASS Beam OUTPUT SUBCLASS OF Linear_mbr (
    PRIVATE VALUE b_effective_depth mm,
    PRIVATE VALUE b_moment newton_mm,
    VALUE b_reinforcement_area mm_square,
    VALUE b_section_type String, // AISC section types.
);
--------------------------------------------------------------------------------
DECLARE CLASS Column OUTPUT SUBCLASS OF Linear_mbr (
    PRIVATE VALUE c_effective_length mm,
    PRIVATE VALUE c_moment newton_mm,
    VALUE c_reinforcement_area mm_square,
    VALUE c_steel_area mm_square,
    VALUE c_concrete_area mm_square,
    VALUE c_section_type String, // AISC section types.
);
--------------------------------------------------------------------------------
DECLARE CLASS Rect_Column OUTPUT SUBCLASS OF Column (
);
--------------------------------------------------------------------------------
DECLARE CLASS Cir_Column OUTPUT SUBCLASS OF Column (
    VALUE c_radius mm
);
--------------------------------------------------------------------------------
DECLARE CLASS Slab OUTPUT SUBCLASS OF Area_mbr (
    PRIVATE VALUE s_effective_xspan mm,
    PRIVATE VALUE s_effective_yspan mm,
    PRIVATE VALUE s_effective_depth mm,
    PRIVATE VALUE s_bending_moment mpa,
    VALUE s_slabtype slab_type,
    VALUE s_reinforcement_area mm_square
);
--------------------------------------------------------------------------------
DECLARE CLASS Waffle_slab OUTPUT SUBCLASS OF Slab
    VALUE rib_width mm,
    VALUE rib_depth mm,
    VALUE rib_spacing mm,
    VALUE topping mm
);
--------------------------------------------------------------------------------
DECLARE CLASS Ribbed_slab OUTPUT SUBCLASS OF Slab (
    VALUE rib_width mm,
    VALUE rib_depth mm,
    VALUE topping mm
```

```
);
--------------------------------------------------------------------------------
DECLARE CLASS Wall OUTPUT SUBCLASS OF Area_mbr (
   VALUE w_reinforcement_area mm_square,
   PRIVATE VALUE w_axial_load mpa,
   PRIVATE VALUE w_bending_moment mpa,
   PRIVATE VALUE w_slenderness_ratio double
);
--------------------------------------------------------------------------------
DECLARE CLASS Wall_opening OUTPUT SUBCLASS OF Area_mbr(
   RELATION parent has_a Wall
);
--------------------------------------------------------------------------------
DECLARE CLASS Wall_Footing OUTPUT SUBCLASS OF Footing (
   RELATION wall has_a Wall
);
--------------------------------------------------------------------------------
DECLARE CLASS Mat_found OUTPUT SUBCLASS OF Footing (
   RELATION piers hasmany Column,
   VALUE number_of_piers int
);
--------------------------------------------------------------------------------
DECLARE CLASS Spread_found OUTPUT SUBCLASS OF Footing (
   RELATION pier has_a Column
);
--------------------------------------------------------------------------------
DECLARE CLASS Footing OUTPUT SUBCLASS OF Area_mbr (
   VALUE f_reinforcement_area mm_square,
   VALUE f_type String, //mat, spread, wall...
   PRIVATE VALUE f_bearing_pressure mpa,
   PRIVATE VALUE f_axial_load mpa,
   PRIVATE VALUE f_moment_load newton_mm
);
--------------------------------------------------------------------------------
DECLARE CLASS Bent OUTPUT SUBCLASS OF Struct_system (
);
--------------------------------------------------------------------------------
DECLARE CLASS Cabin OUTPUT SUBCLASS OF Struct_system (
);
--------------------------------------------------------------------------------
```

```
DECLARE CLASS Truss OUTPUT SUBCLASS OF Struct_system (
    VALUE span mm,
    VALUE depth mm,
    VALUE span_depth_ratio double,
);
```

3.5 Knowledge-based Expert System for Structural Design

3.5.1 Background

In the past, the conceptual design [24] [68] of an initial configuration of structural systems has largely been done manually by the engineer, with little or no support from computer programs. The development of a configuration and the initial sizing of components has been based primarily on experience and available design tables. The computer was thought to become useful only when a design configuration had been developed to a sufficient level of detail to produce drawings and to perform a formal analysis.

The computertools commonly used focus on these two tasks: the production of drawings using CAD systems and the analysis using finite element and matrix analysis programs. These tools can facilitate detailed structural system design through the visualization of the geometry provided by CAD systems and of the behavior of the design provided by analysis programs. However, they provide no assistance to the engineer in producing an initial configuration and sizes of components.

The development of knowledge-based systems in engineering domains has provided an opportunity to provide computer-based support for conceptual and preliminary design. The representation and application of design experience as a knowledge base can be usedto propose alternative configurations and initial component sizes. Much of the experience that engineers have developed through a career of solving design problems could be

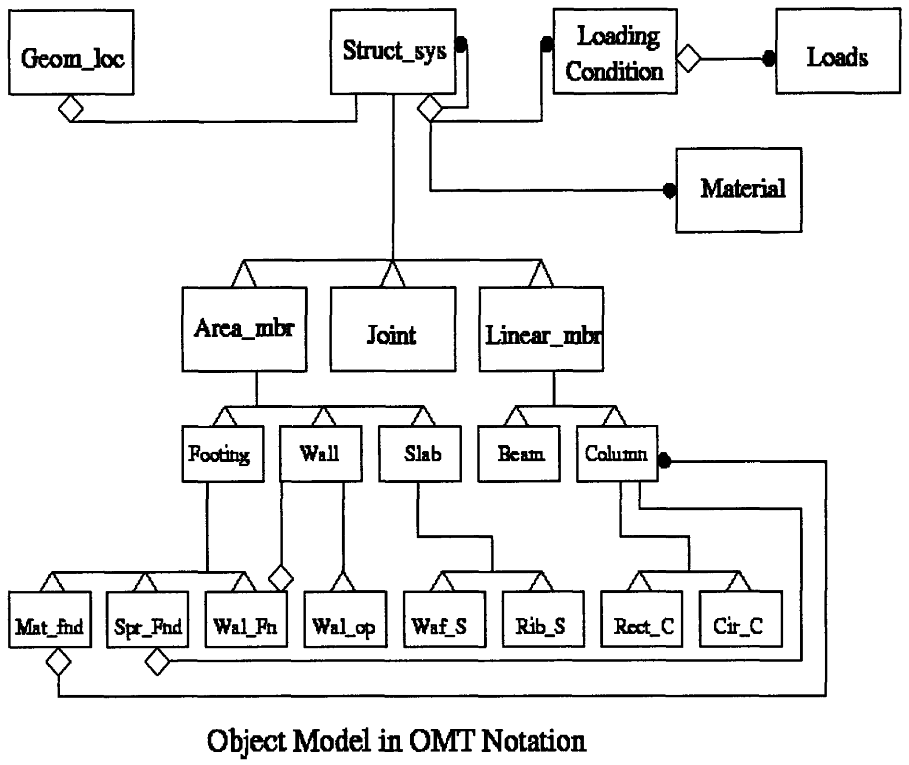


Figure 3.8: Object Model in OML

generalized and applied explicitly to new design problems. So far, the majority of knowledge-based systems for conceptual and preliminary structural design have been developed as research projects in universities.

One of the early knowledge-based systems for preliminary structural design of buildings is HI-RISE. In this system, alternative structural systems are configured by designing lateral and gravity load-resisting systems. Components are sized using approximations of load magnitudes and distribution, heuristics, or table lookup to provide dimensions and/or selection designations. There are several expert system which have been developed for preliminary structural design purpose. These various approaches have two common characteristics: (1) they are developed using a representation language such as Lisp or Prolog; and (2) they rely on generalized design experience. Although none of these system have been used in professional practice, their development has let to a better understanding of what kinds of design knowledge can be used for conceptual design and how such knowledge can be represented in a knowledge base.

3.5.2 Our Approach

The development of a preliminary design knowledge-base has frequently led to the development of generalized representation of design knowledge. Frame, semantic networks, production rules, predicate logic, and the recent emergence of object-oriented programming [37] are the most common solutions.

For our implementation, we chose the rule-based approach to organize the knowledge base. We must be clear, however, that such an implementation involves more than programming fundamental logic statements. In fact, we can use some of the structure of logic to express problem-solving knowledge in terms of rules.

Rules are statements to the effect that we should perform a specified action in a given situation. Typically written in sets of IF-THEN clauses, the left-hand (IF) sides of the rules define the situations that must hold before a rule can be applied, while the right-hand (THEN) sides of the rules define the actions to be taken. In design, rules are used to represent (1) design heuristics or rules of thumb and (2) design codes, such as the building codes. For example,

IF a structural element has one dimension much thinner than the other two

AND it is loaded in that direction

THEN it will behave as a plate in bending

In order to integrate the information of design process with each associated design artifact, we embed these design rules into the design object as their attributes. That is, we map the functional information to the representational form, see Figure 3.9.

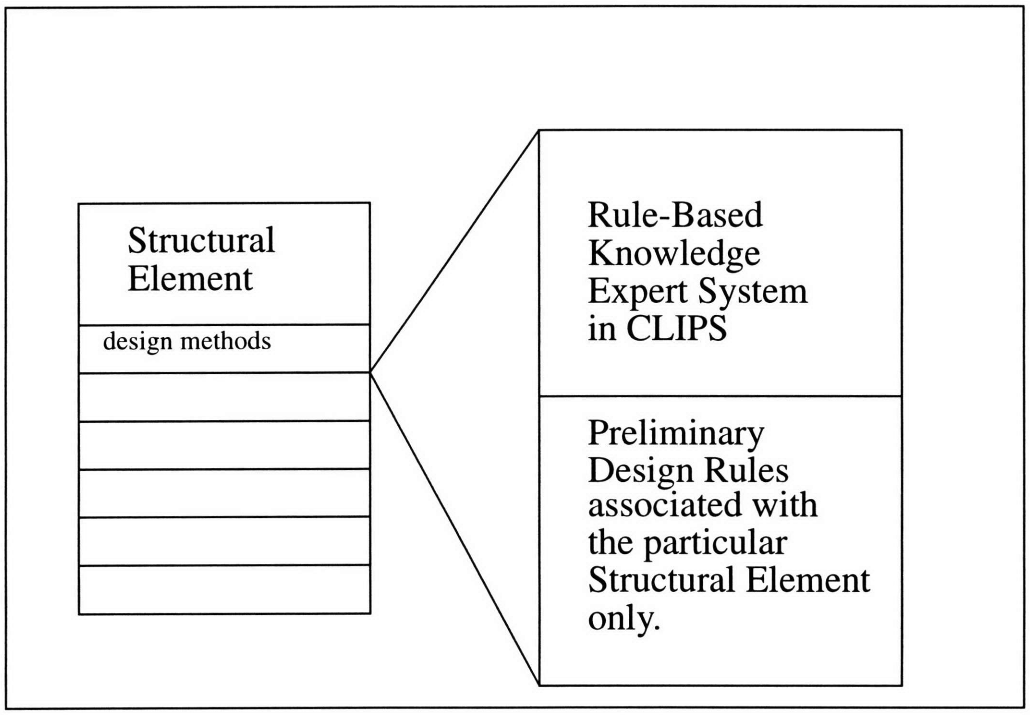

Structural Element
design methods
Rule-Based Knowledge Expert System in CLIPS
Preliminary Design Rules associated with the particular Structural Element only.

Figure 3.9: Objects and Rules

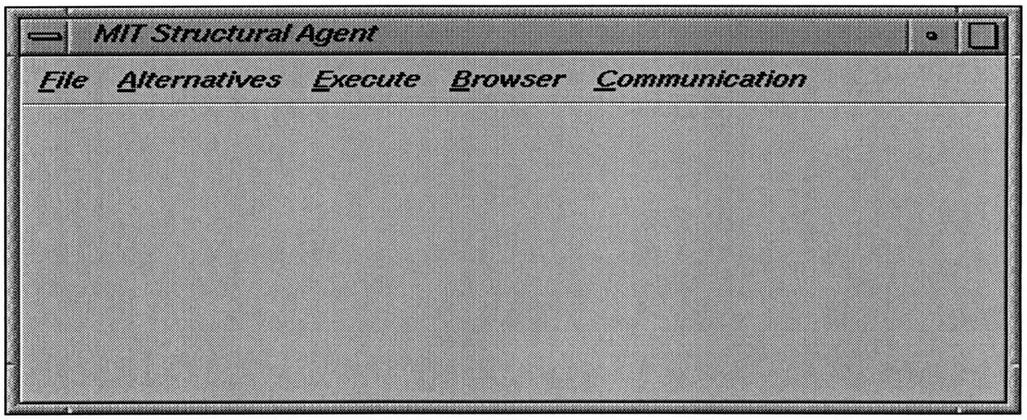


Figure 3.10: Structural Agent Console

3.6 Structural Design in the MIT Agent.

This section describes the role of the structural agent within the ACL project. In order to conform to CMU's nomenclature, we refer to the agent as the structural agent (SA), since it behaves in its interactive mode as a real-life structural engineer during the preliminary building design phase.

Theoretically, the design process can progress in a linear fashion if no participants have objections with the design decisions of others. Unfortunately, this scenario is rare orderly. Evolving program requirements, budget realities, designers' preconceptions of constraints that will be imposed by other agents, increased knowledge of site conditions (such as subsoil problems), public agency reviews, and many other factors make it necessary to go back and modify previous design decisions. Design moves forward, but rarely in the straight sequence implied by the standard description of design.

Moreover, design rarely ends with the completion of design development. Most architects and engineers agree that design choices occur in every step of the process. In other words, building design neither starts with conceptual design nor ends with the completion of the detailed design phase. Therefore, how agents effectively resolve conflicts and intelligently negotiate with one another is one of the central issues in this project.

There are important reasons for suggesting that planning for structural systems should be introduced at the very earliest stages of the design process[15] [67]. These reasons are derived from the need to construct buildings which support and enclose mechanical and other environmental service subsystems, provide support for horizontal and vertical movement of people and materials, as well as access for heating, ventilation, air-conditioning, power, water, and waste disposal. In addition, provision for acoustical and lighting needs is often influenced by structural design. All of these systems interface with a building's structure.

We also categorize the general functionalities of a structural agent into two parts: generation and communication. Interaction and information which flows between SA and other agents is based on a predefined scenario.

3.6.1 Functionality

The SA is to provide at least the functionalities described in the following sections. We divide the structural design into the following three tasks: conceptual, preliminary, and detailed design phases. Normally, iterations are performed on the last step until the response of the structure is deemed adequate and the assumed and computed component forces are in agreement.

3.6.2 The Collaborative Design Process

The scenario of a three-phased structural design within the MIT agent, as well as the collaborative design process with other agents, is outlined below.

Conceptual Design Phase

The purpose of the conceptual design phase is to determine the structural system and its overall layout. Ideally, this stage should be performed in concert with the spatial configuration design, so that concerns about structural performance can be meshed with considerations of the overall 3-D geometry, the organization of circulation, and the allocation of functions to the various levels of the building. Given the input from the owner agent (OA), the project manager agent (PMA), and the architectural agent (AA), the SA willgenerate all feasible solutions and broadcast the results to all interested parties for approval. The AA is expected to be most interested and his approval is sought first.

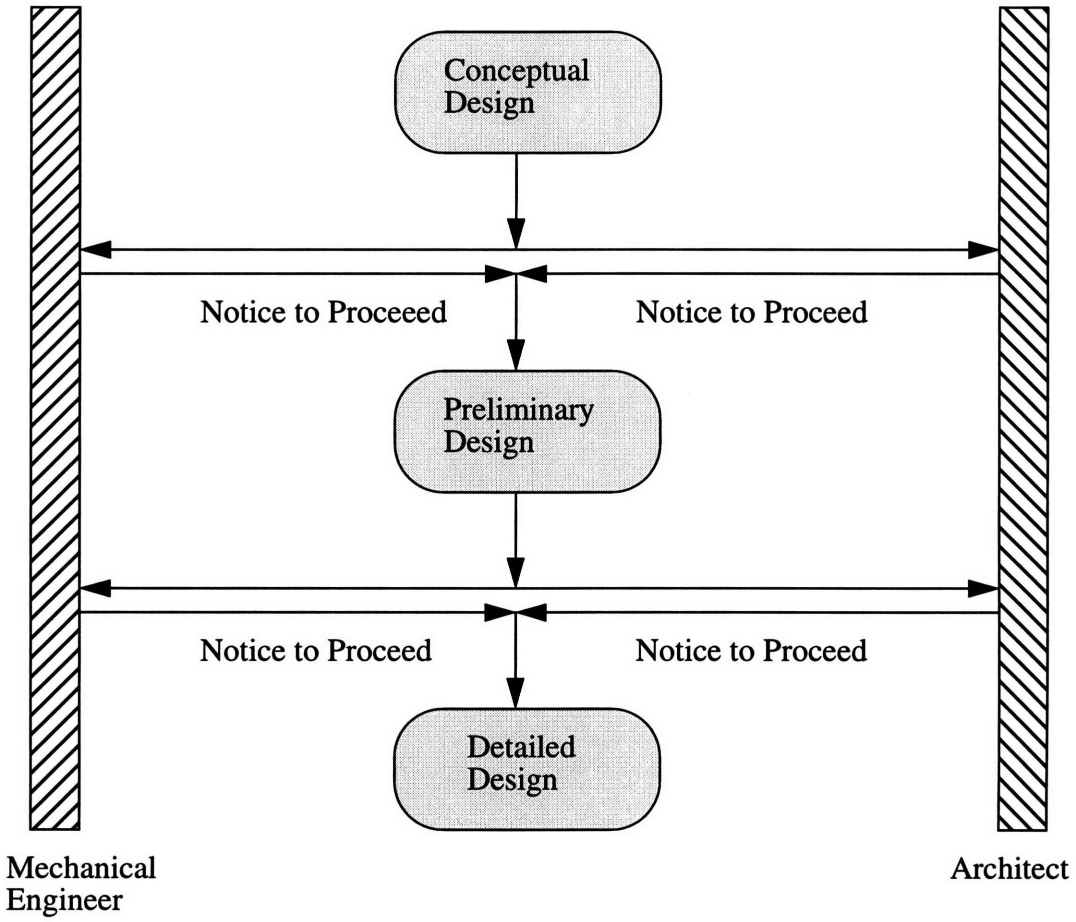


Figure 3.11: Structural Agent's design process

INPUT

The OA provides the project name and number and types of buildings. The PMA provides an early project duration and soil classification. The major input data used for configuration comes from the AA. Please refer to Architectural Agent Vocabulary (AAV) [20] for a detailed description of the following objects.

Basically, the structural agent is interested in the project schedule and budget, geotechnical conditions of the site, occupancy and usage of the building, locations of rooms, stories, walls, openings, floor depth, and geometric locations of massing elements, if available. The OML of structural agent's interests describing input object classes and their attributes is given as follows:

```
DECLARE DOMAIN double number(10,4) SINGLE;
DECLARE DOMAIN real number(10,3) SINGLE;
DECLARE DOMAIN expr char(1000) EXPR real;
DECLARE DOMAIN String char(256) SINGLE;
DECLARE DOMAIN int number(10) SINGLE;
DECLARE DOMAIN mm number(10,4) SINGLE; // length unit
DECLARE DOMAIN mm_square number(10,4) SINGLE; // area unit
DECLARE DOMAIN coordinate number(10,3) MANY;
DECLARE DOMAIN low_corner number(10,3) MANY;
DECLARE DOMAIN high_corner number(10,3) MANY;
DECLARE DOMAIN point TUPLE (x real, y real, z real);
DECLARE DOMAIN angle number(10,3) SINGLE;
DECLARE DOMAIN mpa number(10,4) SINGLE;
DECLARE DOMAIN kilograms number(10,4) SINGLE;
DECLARE DOMAIN newton_mm number(10,4) SINGLE;
DECLARE DOMAIN dollars number(10) SINGLE; //Relationships
DECLARE RELATIONTYPE has_a LINK;
DECLARE RELATIONTYPE hasmany VECTOR;
DECLARE CLASS CATEGORY MIT_Agent;
DECLARE SUBDOMAIN deg double RANGE 0.0 360.0;
DECLARE SUBDOMAIN material String ONEOF concrete steel timber;
DECLARE SUBDOMAIN load_type String ONEOF point linear area;
DECLARE SUBDOMAIN slab_type String ONEOF waffle ribbed flat;
DECLARE SUBDOMAIN connect_type String ONEOF fixed hinge roller;
DECLARE SUBDOMAIN wall_type String ONEOF brick rc;
DECLARE SUBDOMAIN occ_type String ONEOF residential industrial;
DECLARE SUBDOMAIN direction String ONEOF North South East West;
DECLARE FORWARD CLASS Site; DECLARE FORWARD CLASS Owner;

DECLARE CLASS Rectangle INPUT SUBCLASS OF Object (
    VALUE name String,
    VALUE low_corner_x double,
    VALUE low_corner_y double,
    VALUE low_corner_z double,
    VALUE high_corner_x double,
    VALUE high_corner_y double,
    VALUE high_corner_z double
```

```
);
DECLARE CLASS Triangular-Prism INPUT SUBCLASS OF Rectangle (
    VALUE ridge_height double,
    VALUE ridge_orient_x double,
    VALUE ridge_orient_y double,
    VALUE ridge_orient_z double,
};
DECLARE CLASS Project INPUT SUBCLASS OF Object (
    RELATION located_at has_a Site ,
    RELATION owned_by has_a Owner ,
    VALUE name String,
    VALUE budget dollars ,
    VALUE occ_type occ_type
);
DECLARE CLASS Owner INPUT SUBCLASS OF Object (
    VALUE name String ,
    VALUE title String ,
    VALUE address String ,
    VALUE city String ,
    VALUE area_code String ,
    VALUE telephone String ,
    VALUE fax String ,
    RELATION projects_owned has_a Project
);
DECLARE CLASS Site INPUT SUBCLASS OF Object (
    RELATION boundary has_a Rectangle ,
    RELATION has_project has_a Project
);
DECLARE CLASS Occupancy INPUT SUBCLASS OF Object (
    VALUE occ occ_type,
    VALUE load int
);
DECLARE CLASS Roof INPUT SUBCLASS of Object (
    VALUE roofing_type String,
    VALUE constr_type String
);
DECLARE CLASS Massing_element INPUT SUBCLASS OF Object (
    RELATION geometry has_a Rectangle,
    RELATION roof has_a Roof,
    VALUE name String,
    VALUE origin_x double,
    VALUE origin_y double,
    VALUE origin_z double,
    RELATION constituent hasmany Storey,
    VALUE height mm
);
DECLARE CLASS Building INPUT SUBCLASS OF Object (
    RELATION geometry has_a Rectangle ,
    RELATION roof has_many Roof,
    VALUE name String,
    VALUE origin_x double,
    VALUE origin_y double,
    VALUE origin_z double,
    VALUE default_centerwall_thickness mm,
    RELATION constituent hasmany Massing_element,
    VALUE height mm
);
DECLARE CLASS Storey INPUT SUBCLASS OF Object (
    RELATION geometry has_a Rectangle,
    RELATION part_of has_a Building,
    VALUE name String,
    VALUE origin_x double,
    VALUE origin_y double,
    VALUE origin_z double,
    RELATION parts hasmany Wall,
    RELATION adjacent_enclosure hasmany Wall,
    RELATION constituents hasmany Room,
    VALUE elevation mm,
    RELATION story_below has_a Storey,
    RELATION story_above has_a Storey,
    RELATION occupied has_a Occupancy
);
DECLARE CLASS Room INPUT SUBCLASS OF Object (
    RELATION part_of has_a Storey,
    RELATION geometry has_a Rectangle,
    VALUE name String,
    VALUE origin_x double,
    VALUE origin_y double,
    VALUE origin_z double,
    VALUE constr_type String,
    RELATION adjacent_enclosure hasmany Wall,
    VALUE elevation mm,
    RELATION occupied has_a Occupancy
);
DECLARE CLASS Wall INPUT SUBCLASS of Object (
    RELATION part_of has_a Storey,
    RELATION parts hasmany Window,
    RELATION geometry has_a Rectangle,
    VALUE name String,
    VALUE origin_x double,
    VALUE origin_y double,
    VALUE origin_z double,
    VALUE constr_type String,
    VALUE type String,
    VALUE open_perc double
);
DECLARE CLASS Opening INPUT SUBCLASS of Object (
    VALUE origin_x double,
    VALUE origin_y double,
    VALUE origin_z double
);
DECLARE CLASS Door INPUT SUBCLASS of Opening (
    RELATION geometry has_a Rectangle,
    VALUE name String,
    RELATION part_of has_a Wall
);
DECLARE CLASS Window INPUT SUBCLASS of Opening (
    RELATION geometry has_a Rectangle,
    VALUE name String,
    RELATION part_of has_a Wall
);
-------------------------------------------------------------------------------
```

OUTPUT

The output in this phase is comprised of a recommended structural system and an alternate system. Each alternative is classified in terms of basic structural materials, column-beam-girder grid, truss type, location of shear walls, openings, and foundation type. In this simple example, the structural agent produces two structural systems.

Based on the input data from other agents, the structural agent employs the following design rules to generate the conceptual design:

Sample Rules for Conceptual Design

- **(rule** (if rc_availability is yes and material is rc)
  (then structural_system is bearing_wall with certainty 70.0
  structural_system is frame with certainty 70.0))

- **(rule** (if soil is tight-sand)
  (then footing is spread_footing with certainty 70.0
  structural_system is frame with certainty 80.0))

- **(rule** (if structural_system is frame and material is rc)
  (then economic_span_depth_ratio is 20 with certainty 80.0)
  (then spacing_height_range is 16 to 40 with certainty 80.0))

These rules are used to create objects. The OML, as the following listing, describes what object classes and attributes will be sent to other agents.

Sample output in OML

ADD FRAME Material;
SET RC TO $CURRENT;
ADD FRAME Struct_system;
SET Cabin TO $CURRENT;
ADD VALUE Cabin name {Cabin};
ADD RELATION Cabin material RC;
ADD FRAME Struct_system;
SET Frame_Structure TO $CURRENT;
ADD VALUE Frame_Structure name {Frame_Structure};
ADD VALUE Frame_Structure dimen_x {7200.000000};
ADD VALUE Frame_Structure dimen_y {3600.000000};
ADD VALUE Frame_Structure dimen_z {3300.000000};
ADD FRAME Struct_system;
SET Bearing_Wall_Structure TO $CURRENT;
ADD VALUE Bearing_Wall_Structure name {Bearing_Wall_Structure};
ADD VALUE Bearing_Wall_Structure dimen_x {7200.000000};
ADD VALUE Bearing_Wall_Structure dimen_y {3600.000000};
ADD VALUE Bearing_Wall_Structure dimen_z {3300.000000};

```
ADD RELATION Frame_Structure material RC;
ADD RELATION Bearing_Wall_Structure material RC;
ADD FRAME Geometry_loc;
SET Frame_loc1 TO $CURRENT;
ADD VALUE Frame_loc1 name {Frame_loc1};
ADD VALUE Frame_loc1 origin_x {0.000000};
ADD VALUE Frame_loc1 origin_y {0.000000};
ADD VALUE Frame_loc1 origin_z {0.000000};
ADD VALUE Frame_loc1 orient_x {1.000000};
ADD VALUE Frame_loc1 orient_y {1.000000};
ADD VALUE Frame_loc1 orient_z {1.000000};
ADD RELATION Frame_Structure components Frame_loc1;
ADD RELATION Frame_loc1 design Frame_Structure;
ADD FRAME Geometry_loc;
SET Wall_loc1 TO $CURRENT;
ADD VALUE Wall_loc1 name {Wall_loc1};
ADD VALUE Wall_loc1 origin_x {0.000000};
ADD VALUE Wall_loc1 origin_y {0.000000};
ADD VALUE Wall_loc1 origin_z {0.000000};
ADD VALUE Wall_loc1 orient_x {1.000000};
ADD VALUE Wall_loc1 orient_y {1.000000};
ADD VALUE Wall_loc1 orient_z {1.000000};
ADD RELATION Bearing_Wall_Structure components Wall_loc1;
ADD RELATION Wall_loc1 design Bearing_Wall_Structure;
ADD RELATION Cabin structural_system Frame_Structure;
ADD RELATION Cabin structural_system Bearing_Wall_Structure;
.......
```

Preliminary Design Phase

The major task in this phase is to compute the approximate size of structural components. The SA will send the results to the AA for approval.

INPUT

A Notice to Proceed from the facilitator instructs the agent to advance to the next phase.

OUTPUT

The location of the lateral and gravity subsystems and the approximate size of the structural components are the output of this design phase. For example, preliminary sizings for slabs, beams, and columns. preliminary sizings for truss members. preliminary sizings for footings, shear walls, and so on.

Based on the output of the previous stage, the structural agent utilizes the following rules to generate the preliminary sizing and locations of major structural components:

Sample Rules for Preliminary Design

- (**rule** (if structural_system is column)
  (then unbraced_length is from 10 to 30 with certainty 80.0))

- (**rule** (if structural_system is column and material is rc)
  (then economic_span_depth_ratio is 20 with certainty 80.0)
  (then spacing_height_range is 18 to 25 with certainty 80.0))

- (**rule** (if structural_system is truss and material is steel and type is triangular)
  (then span is 30 to 150 with certainty 100.0)
  (then spacing is 12 to 20 with certainty 100.0))

- (**rule** (if structural_system is truss and material is steel)
  (then economic_span_depth_ratio is 10 to 12 with certainty 80.0)
  (then spacing_height_range is 20 to 25 with certainty 80.0))

- (**rule** (if structural_system is footing and material is concrete and type is column)
  (then thickness is wall_thickness with certainty 80.0)
  (then minimum_width is 3 with certainty 80.0)
  (then width is 2.0 * wall_thickness with certainty 80.0))

These rules are used to generate objects for preliminary design. The OML, as the following listing, describes what object classes and attributes will be sent to other agents for approval.

Sample output in OML

```
ADD FRAME Material;
SET RC TO $CURRENT;
ADD FRAME Struct_system;
SET Cabin TO $CURRENT;
ADD RELATION Cabin material RC;
ADD FRAME Bent;
SET Bent1 TO $CURRENT;
ADD VALUE Bent1 dimen_x {3600.000000};
ADD VALUE Bent1 dimen_y {2400.000000};
ADD VALUE Bent1 dimen_z {20.000000};
ADD FRAME Bent;
SET Bent2 TO $CURRENT;
ADD VALUE Bent2 dimen_x {3600.000000};
ADD VALUE Bent2 dimen_y {2400.000000};
ADD VALUE Bent2 dimen_z {20.000000};
ADD FRAME Geometry_loc;
SET Bent1_loc TO $CURRENT;
ADD VALUE Bent1_loc origin_x {0.000000};
ADD VALUE Bent1_loc origin_y {0.000000};
ADD VALUE Bent1_loc origin_z {0.000000};
ADD VALUE Bent1_loc orient_x {0.000000};
ADD VALUE Bent1_loc orient_y {0.000000};
ADD VALUE Bent1_loc orient_z {0.000000};
ADD RELATION Bent1 components Bent1_loc;
ADD RELATION Bent1_loc design Bent1;
ADD FRAME Geometry_loc;
SET Bent4_loc TO $CURRENT;
ADD VALUE Bent4_loc origin_x {7200.000000};
ADD VALUE Bent4_loc origin_y {0.000000};
ADD VALUE Bent4_loc origin_z {0.000000};
ADD VALUE Bent4_loc orient_x {0.000000};
ADD VALUE Bent4_loc orient_y {1.000000};
ADD VALUE Bent4_loc orient_z {1.000000};
ADD RELATION Bent1 components Bent4_loc;
ADD RELATION Bent4_loc design Bent1;
ADD FRAME Geometry_loc;
SET Bent2_loc TO $CURRENT;
ADD VALUE Bent2_loc origin_x {2400.000000};
ADD VALUE Bent2_loc origin_y {0.000000};
ADD VALUE Bent2_loc origin_z {0.000000};
ADD VALUE Bent2_loc orient_x {0.000000};
ADD VALUE Bent2_loc orient_y {1.000000};
ADD VALUE Bent2_loc orient_z {1.000000};
ADD RELATION Bent2 components Bent2_loc;
ADD RELATION Bent2_loc design Bent2;
.........
```

Detailed Design Phase

This phase provides a detailed design of all structural components and verifies satisfaction of design codes. Finite element structural analysis programs are employed to check deflection and drifts, as well as the force resultants acting on the components. This analysis will be updated if component characteristics change during this phase. After completing the negotiations with the AA, we proceed to the Phase III for detailed structural component design. We can utilize reliable structural analysis programs such as SAP90, GTSTRUDL, and ETABS to verify and optimize the design.

INPUT

A Notice to Proceed from the facilitator tells the agent move to the next phase.

OUTPUT

The final sizing of the structural components is presented in this phase.

3.6.3 Scenario

The following scenario describes interactions between the SA and other agents.

- Receive architectural plan from the Architecture Agent (AA).
- Generate data model for structural design.
- Produce several structural system alternatives.

- Send two alternatives to the AA for approval.
- Receive the AA's response for structural system.
- Perform preliminary design.
- Send the preliminary component sizes to the AA.
- Receive notification for change (location or shape).
- Evaluate the feasibility of the change and the necessity of redesign.
- Negotiate with the AA.
- Send the updated change or redesign to the AA.
- Perform detailed design.
- Send the result of detailed design to the AA after negotiations are complete.
- Receive OK from AA.

Criteria for Redesign

The SA should be able to determine if a redesign is needed after she receives the AA's request for a modification. For example, if the AA changes the boundary of a building or the spacing of columns, then the SA should restart the structural design from Phase I. On the other hand, if the AA requests a change in the depth of the slab for compatibility with the HVAC system, then the SA need only restart the design from Phase II.

Structural Requirements

The SA is supposed to conform to these requirements and constraints.

Functional Requirements

- spatial function

- load carrying function
- serviceability requirement
- deflection
- vibration

Documentation requirements

The following documentations are required during the period of structural design development [78]. The structural agent must be able to provide these documents at the other agents' request.

- basic structural system and dimensions
- final structural design criteria
- foundation design criteria
- preliminary sizing of major structural components
- critical coordination clearances
- outline specifications or material lists

Chapter 4

Implementation

4.1 Introduction

The MIT agent plays the role of structural design agent in this collaborative design process. There are several software tools being employed and integrated to serve for this particular purpose. The ObjectStore database management system [28] is utilized as the foundation layer for information exchange between different software tools as well as the data repository. The agent console is implemented in Tcl/Tk [79] which maintains the communication channel with facilitator. A forward chaining rule-based language called CLIPS [8] is employed and embedded within the system as an inference engine. ET++ [80] , a full-featured C++ class library, is used as the building blocks of graphical user interface, including a customized object structure browser which is particularly suitable for engineering design.

In addition, the agent is also equipped with the capabilities of communicating with facilitator via a set of communication API, high-level object-oriented semantic modeling, and object version management. All of these features are specially devised for assisting the collaborative engineering design process. In order to take advantage of the communication API and OML, a set of translation routines are written to hide the details of directly calling the API and avoid the possibility of hand coding.

The structure of the software system is shown in Figure 4.1. The system runs on a Sun Sparcstation 10 with SunOS 4.1.3. C++ [10] [11] is the major programming language and, of course, ANSI/C compilers are also needed to build some of the libraries. GNU gdb debugger is very helpful in finding programming errors. In terms of interactive computing, the overall performance of this system is satisfactory.

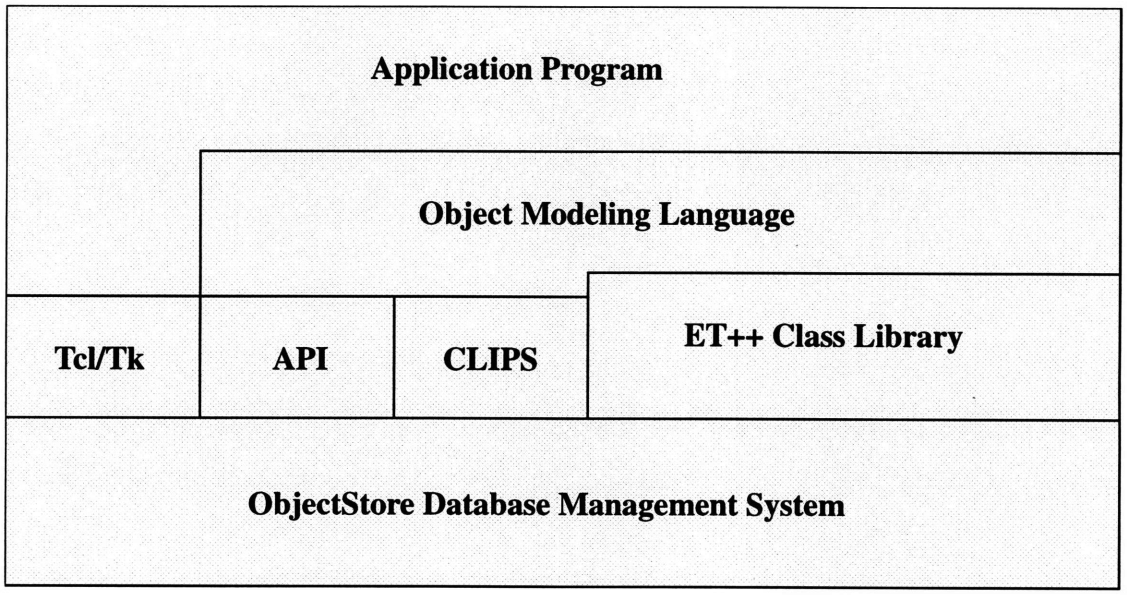


Figure 4.1: Layered Architecture of Structural Agent

A tremendous amount of efforts have been devoted to the implementation of this structural design agent. The critical issue is how to find a way to integrate these heterogeneous and independently developed software tools to work together in a seamless fashion. Some decisions are made based on practical concerns, therefore, they may not be the most elegant implementation strategies.

4.2 Object Modeling Language

The first task of implementing the design agent is to bind OML with C++. We have to define the structural engineering vocabulary in OML at first, then run the language binding translator to translate the OML code to C/C++. Besides, the user of OML also have to provide a set of maps to show the relationships of the OML construct and its C++ counterpart. After successfully compiling the code generated by the translator, the user can generate the executable program via combining the existing data structures, mapping functions, and object model library.

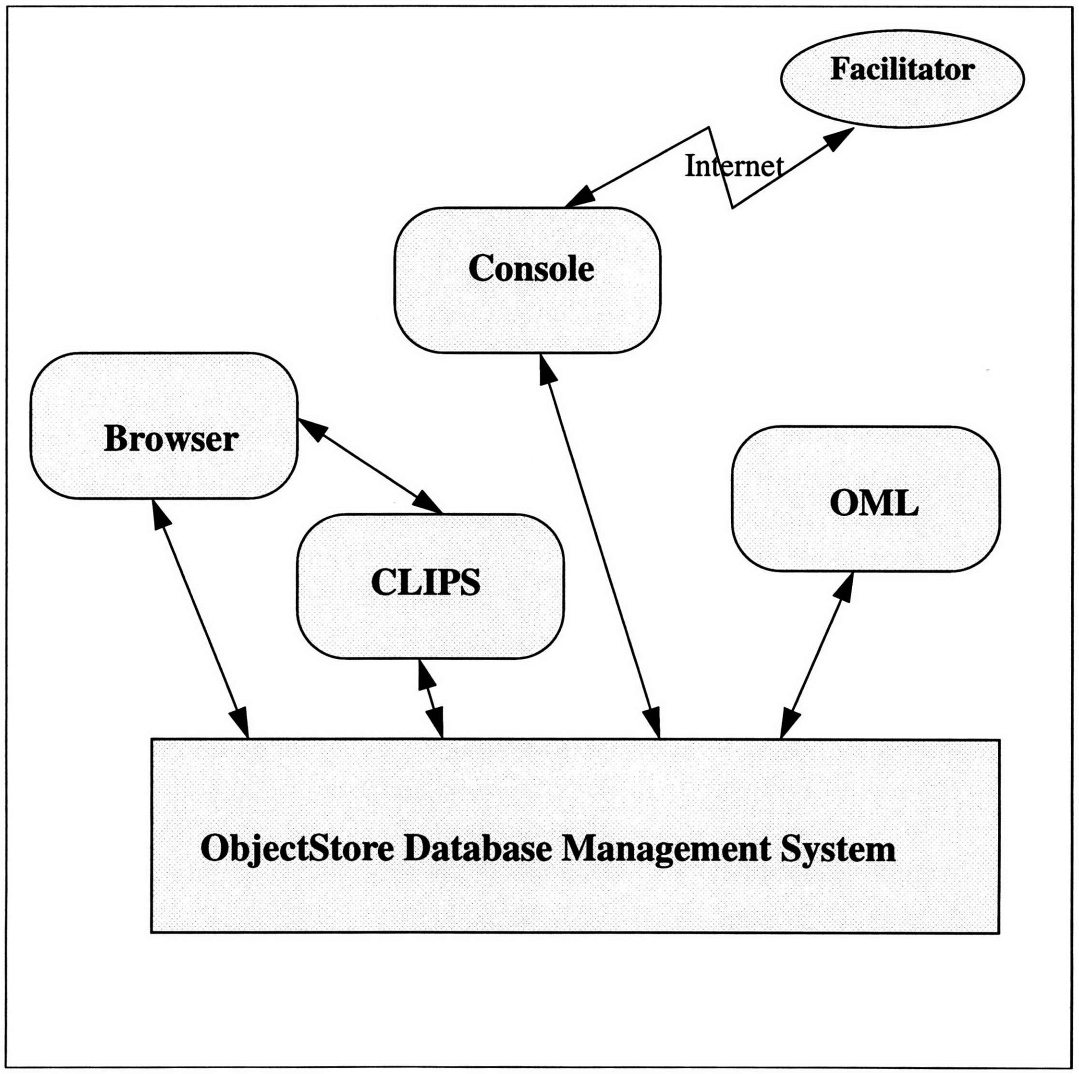


Figure 4.2: Information Exchange of Structural Agent

Experience with OML

The ideas behind OML are to provide a high level object-oriented semantic modeling language. Problems such as language binding, message routing, and user event dispatching are resolved in this conceptual framework. With this powerful tool, the designer can fully concentrate on development of domain knowledge and easily manipulate the design objects. Due to the lack of standards in C++, the name mangling problem arises while the programmer would like to link two libraries, which are compiled under different compilers, to generate

Most of the OML code provided by CMU team is in C. It is pretty straightforward to call C function from C++, and vice versa. As a result, except the interface code, most of the OML code is compiled with standard ANSI/C compiler and then linked with the existing data structures which are essentially pure C++ code.

4.3 Tcl/Tk [79]

Tcl is an application-independent command language. It exists as a C library package that can be used in many different programs. The Tcl library provides a parser for a simple but fully programmable command language. The library also implements a collection of built-in commands that provides general-purpose programming constructs such as variables, lists, expressions, conditionals, looping, and procedures. Individual application programs extend the basic Tcl language with application-specific commands. The Tcl library also provides a set of utility routines to simplify the implementation of tool-specific commands.

Tcl is unusual because it presents two different interfaces : a textual interface to users who issues Tcl commands, and a procedural interface to the applications in which it is embedded. Each of these interfaces must be simple, powerful and efficient. There were four major features which make Tcl distinct from other programming languages:

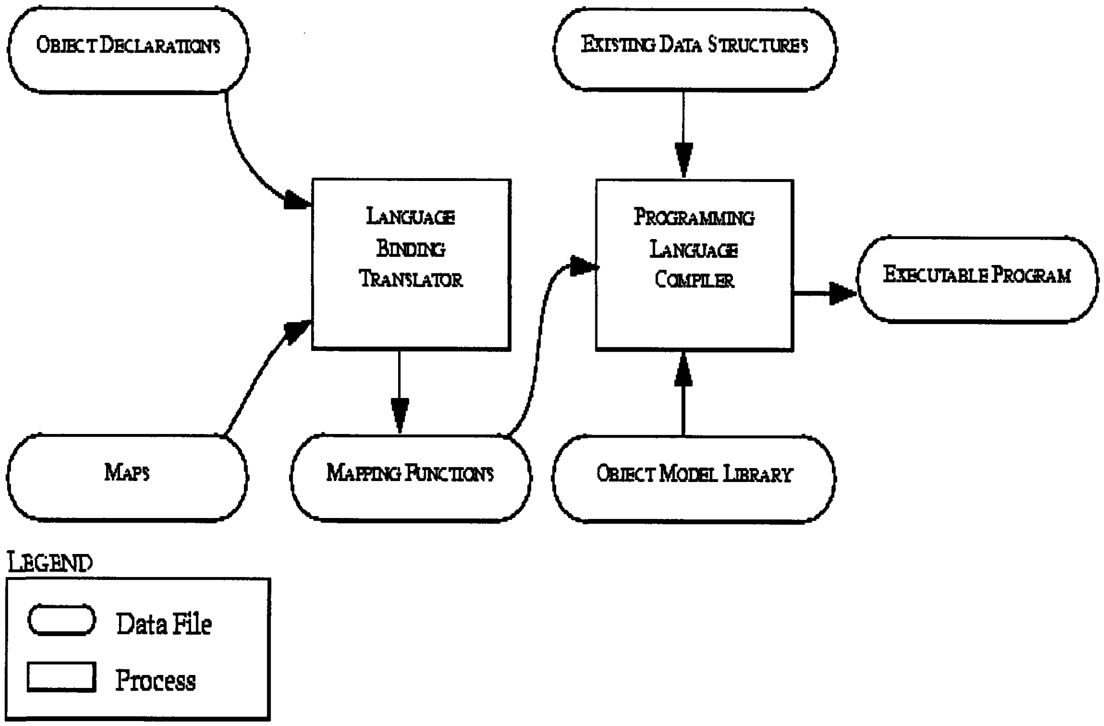


FIGURE 1.Basic Translation Process

Figure 4.3: Basic Translation Process

- **The language is for commands.**

  Almost all Tcl “programs” will be short, many only line long. Most programs will be typed in, executed once or perhaps a few times, and then discarded. This suggests that the language should have a simple syntax so that it is easy to type commands. Most existing programming languages have complex syntax; the syntax is helpful when writing long programs but would be clumsy if used for a command language.

- **The language should be programmable.**

  It should contain general programming constructs such as variables, procedures, conditionals, and loops, so that user can extends the built-in command set by writing

Tcl procedures. Extensibility also argues for a simple syntax: this makes it easier for Tcl programs to generate other Tcl programs.

- **The language must permit a simple and efficient interpreter.**

  For the Tcl library to be included in many small programs, particularly on machines without shared-library facilities, the interpreter must not occupy much memory. The mechanism for interpreting Tcl commands must be fast enough to be usable for events that occur hundreds of times a second, such as mouse motion.

- **The language must permit a simple interface to C applications.**

  It must be easy for C applications to invoke the interpreter and easy for them to extend the built-in commands with application-specific commands.

Although the built-in Tcl commands could conceivably be used as a stand-alone programing system. Tcl is really intended to be embedded in application programs. An application using Tcl extends the built-in commands with a few additional commands related to that particular application.

To use Tcl, an application creates an object called an *interpreter,* using the following library procedure:

*Tcl_Interp * Tcl_CreateInterp();*

An interpreter consists of a set of commands, a set of variable bindings, and a command execution state. It is the basic unit manipulated by most of the Tcl library procedures. Simple applications will use only a single interpreter, while more complex applications may use multiple interpreters for different purposes. Once an application has created an interpreter, it call the Tcl_createCommand procedure to extend the interpreter with application-specific commands:

*typedef int (*Tcl_CmdProc)(ClientData clientData, Tcl_Interp *interp, int argc, char *argv[]);*

*Tcl_CreateCommand(Tcl_Interp *interp, char *name, Tcl_CmdProc proc, ClientData clientData);*

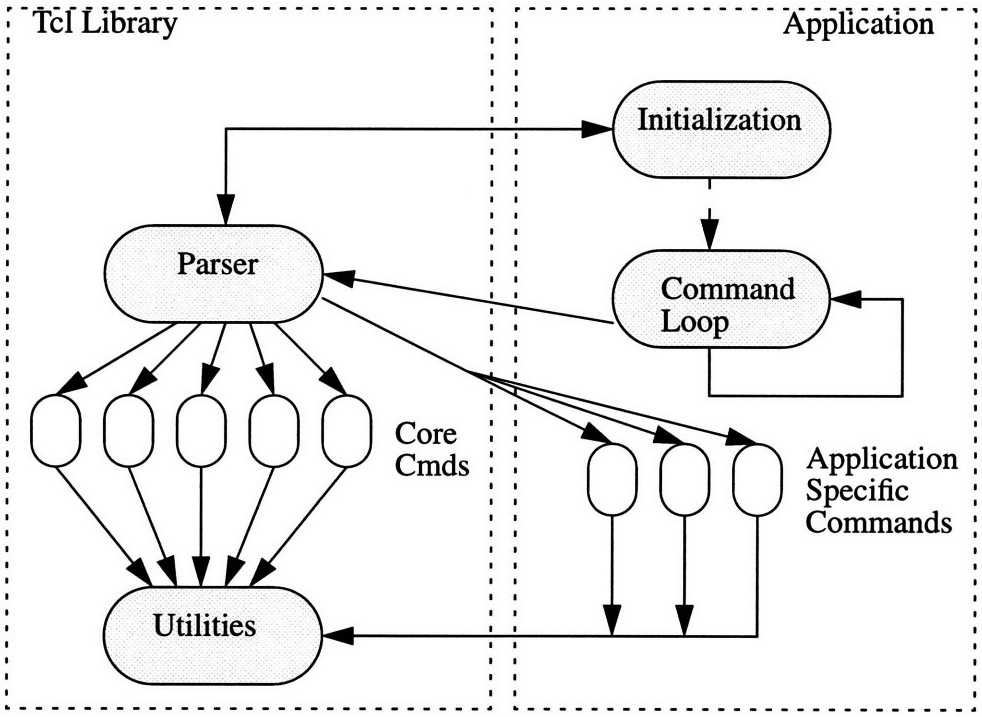


Figure 4.4: Tcl Internal Structure

Each call to Tcl_CreateCommand associates a particular command name (name) with a procedure that implements that command (proc) and an arbitrary single-word value to pass to that procedure (clientData).

After creating application-specific commands, the application enters a main loop that collects commands and passes them to the Tcl_Eval procedure for execution:

*int Tcl_Eval(Tcl_Interp *interp, char *cmd);*

The Tcl_Eval procedure parses its cmd argument into fields, looks up the command name in the table of those associated with the interpreter, and invokes the command procedure associated with that command. All command procedures, whether built-in or application specific, are called in the same way, as described in the typedef for Tcl_CmdProc above.

For example, in order to integrate Tcl with ACL API, we can built a menu command called **Connect** which establishes the communication channel from the design agent to the facilitator. We can predefine the procedure and pass the command name to the interpreter. The Tcl interpreter will invoke the associated function to execute the task.

Experience with Tcl/Tk

Tcl is an excellent glue language. A Tcl application can include many different library packages, each of which provides an interesting set of Tcl commands. With Tk, the user can construct sophisticated graphical user interface with a few lines of code. The scripting feature of Tcl makes it a genuinely rapid development tool. The major drawback of Tcl/Tk is also its loose coupling and flat structure. It is apparently not suitable to adopt Tcl to develop a browser for highly structured and closely related object relationships in structural engineering vocabulary.

4.4 Object-oriented Database Management System [36] [56] [57]

One of the principal challenges in information management in engineering design is the integration of diverse, special-purpose application modules. As compared with traditional relational database system, object-oriented database is particularly suitable for engineering design and modeling because it has superior semantic expressiveness and clearer mapping of conceptual images. Besides, object-oriented problem representation and encapsulation can help solve the complex design problem in a more organized and coordinated manner. Every behavior of an object is managed within the object itself and its relations with other objects are carefully controlled and coordinated.

4.4.1 Features of ObjectStore Database Management System

The typical characteristics of object-oriented database management systems are outlined below:

- **Object Model** - The basic modeling primitive is object and objects can be categorized into types. The behavior of objects is defined by a set of operations that can be executed on an object of the type.
- **Object Definition Language** - It is a specification language used to define the interfaces to object types that conform to the Object Model.
- **Object Query Language -** It allows the user to query denotable object starting from their names, which act as entry points into a database.
- **Programming Bindings** - It often binds with the object-oriented programming language such as C++ or Smalltalk.

The exclusive features provided by ObjectStore are described as follow :

Distributed architecture

When considered in aggregate, dispersely located computers in either standalone or networked configurations are by far the most powerful computing resource in an organization. The challenge of scaling up in a distributed computing environment is to harness the power of these computers by making it possible to do more of the processing on the workstation rather than a centralized mainframe. ObjectStore's architecture represents a key innovation in this area.

Programming Support [38]

Object databases are also extremely efficient for interactive applications which use complex structures and are highly compatible with many object-oriented development tools as follows:

- **Versatile Data Structures**

  In addition to the data definition and manipulation facilities provided by the host language C++, ObjectStore provides support for accessing persistent data inside transactions, a library of collection types, bidirectional relationships, an optimizing query facility, and a version management facility to support collaborative work. Tools supporting database schema design, database browsing, and application compilation and debugging, are also provided.

  Within the ObjecStore, the manipulation of data looks just like an ordinary C++ program, even though the objects are persistent. ObjectStore automatically sets read and write locks, and automatically keeps track of what has been modified, helping to protect the integrity of the database against the possibility of crash. Access to persistent data is guaranteed to be transaction-consistent (i.e., all-or-none update semantics), and recoverable in the event of system failure.

  In ObjectStore, the relationships can be thought of as a pair of inverse pointers, so that if one object points to another, the second object has an inverse pointer back to the first. Relationships maintain the integrity of these pointers. One-to-one, one-to-many, and many-to-many relationships are all supported.

- **Version Management, Configurations, and Alternatives [35] [45] [46]**

  Objectstore provides facilities for multiple users to share data in a cooperative fashion. With these facilities, a user can check out a version of an object or group of objects, make changes (perhaps entailing a long series of individual update transactions), and then check changes back in to the main development project so that they are visible to other members of the cooperating team. In the interim, other users can continue to use the previous versions, and therefore are not impeded by concurrency conflicts on their shared data, regardless of the duration of the sessions involved. If other users want to make concurrent parallel changes, they can check out alternative versions of the same object or group of objects, and work on their versions in private. Again, the result is that there is not concurrency conflicts, even though the users are operating on the same object. Alternative versions can later be merged back together to reconcile differences resulting from this parallel development.

Users can control exactly which versions to use, for each object or group of objects of interest, by setting up private workspaces that specify the desired version. This might be the most recent version, or a particular previous version, or even a version on an alternative branch. Users can also use workspace to selectively share their work in progress. Workspaces can inherit from other workspaces, so that one designer could specify that his or her workspace should be default inherit from the team's global workspace or could then add individual new versions as changes are made, overriding the default. The versioning feature is very useful for assisting the designer to keep track of the design change as well as alternatives.

4.5 Features of ET++ Graphical User Interface Application Framework

Constructing the graphical user interface [42] often requires considerable effort because they must not only provide the functionality of conventional program, but also have to show the data as well as manipulation concepts in a pictorial way. Handling commands from input devices in order to build an event-driven application complicates the programmer's task even more.

A promising solution is that of an object-oriented application framework [9] that defines much of an applications's standard user interface, behavior, and operating environments so that the programmer can concentrate on implementing the application specific parts. An application framework allows reusing the abstract design of an entire application, modeling each major component with an abstract class. In a graphical application, for example, these components are documents, windows, commands and the application itself. While the framework approach is useful for the development of any software, it is especially attractive if a standard user interface should be encouraged, for it is possible to completely define the components that implement this standard and to provide these reusable components as building blocks to other developers. This is an advantage over the toolbox approach where user interface are explained textually rather than being wired into the software.

ET++ is a homogeneous object-oriented class library integrating user interface building elements, basic data structures, and support for object input/output with high level application framework components. The main goals in designing ET++ have been the desire to substantially ease the building of highly interactive applications with consistent user interfaces following well-known desktop metaphor, and to combine all ET++ classes into a seamless system structure.

The Model-View-Controller (MVC) paradigm [42] is an approach to modularize the structure of a user interface. MVC strictly separates interactive behavior of an application from the underlying data structure (model). The interactive behavior is split into rendering a data structure (view) and reacting on user input (controller). The MVC paradigm was used in ET++ primarily where different implementation of a data structure (model) should have the same interactive behavior. Most of the advanced application framework for user interface design employ this approach.

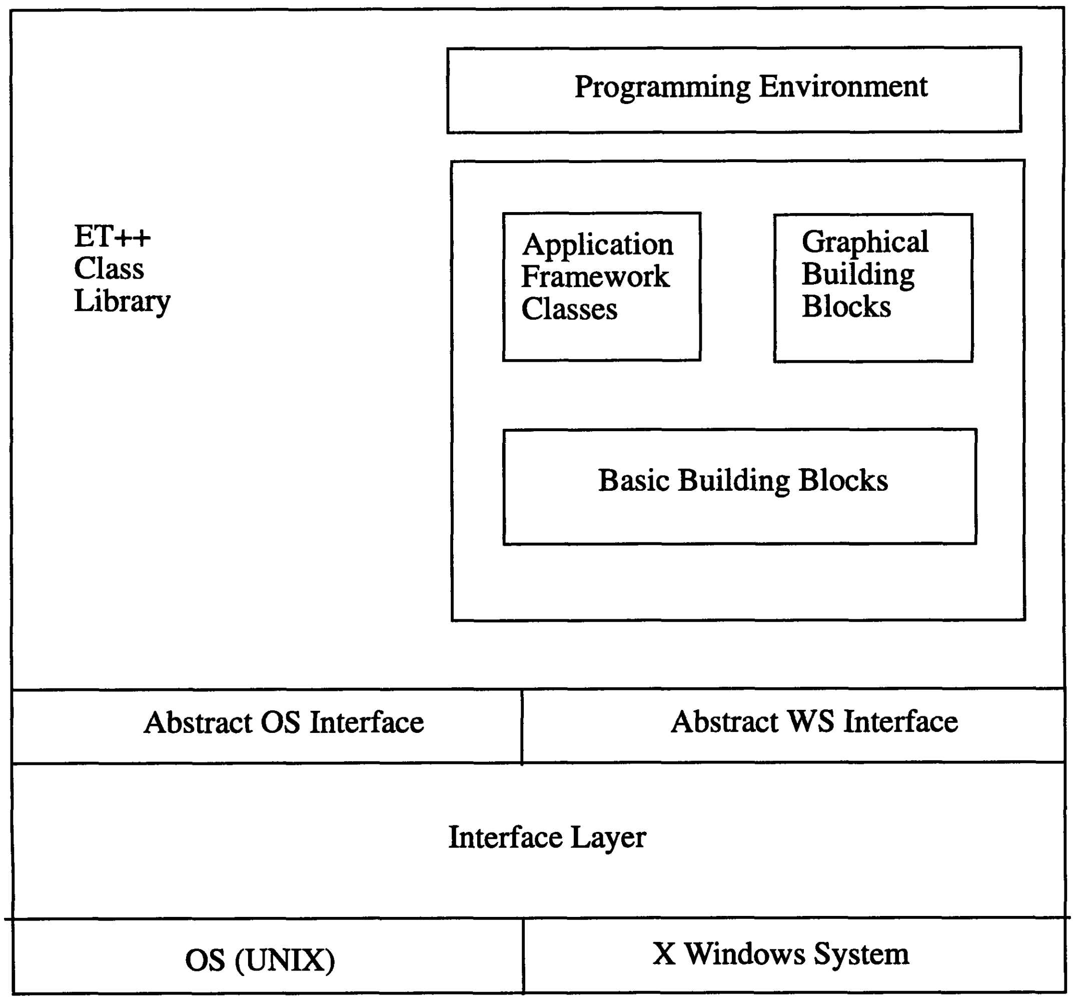


Figure 4.5: ET++ Architecture

The backbone of ET++ architecture is a class hierarchy with about 234 classes and a small device dependent layer mainly mapping an abstract window and operating system interface to an underlying real system. The Basic Building Blocks contain the most important abstract classes of the ET++ class hierarchy: the class Object, the root of the overall class hierarchy, implements the common behavior for all ET++ classes; the class **VObject** (visual object) implements the common behavior for all graphical classes, In addition, the

Basic Building Block define general useful basic data structures, like arrays, lists, sets, etc. which are used heavily for the implementation of ET++ itself.

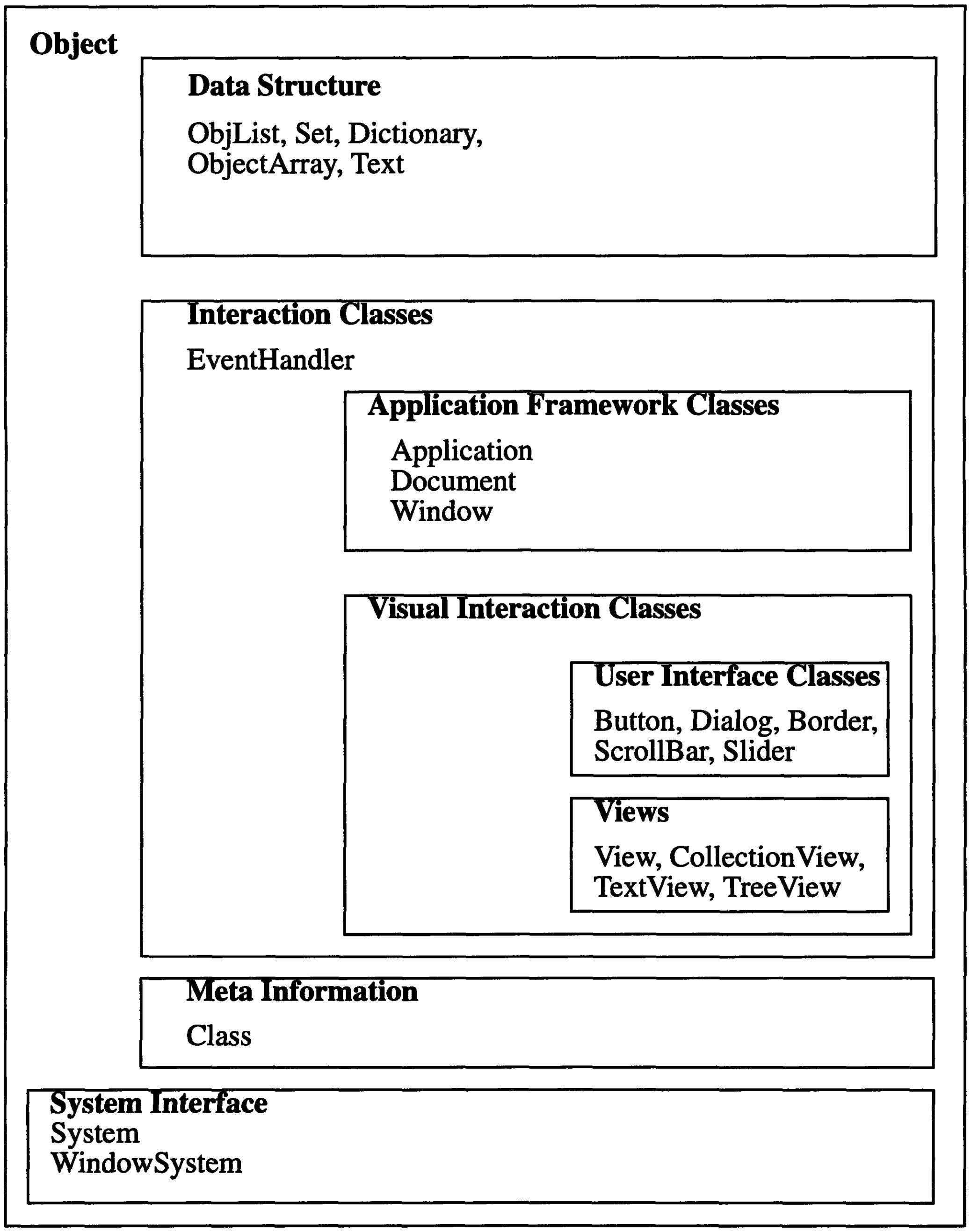

Object
Data Structure
ObjList, Set, Dictionary, ObjectArray, Text
Interaction Classes
EventHandler
Application Framework Classes
Application
Document
Window
Visual Interaction Classes
User Interface Classes
Button, Dialog, Border, ScrollBar, Slider
Views
View, CollectionView, TextView, TreeView
Meta Information
Class
System Interface
System
WindowSystem

Figure 4.6: ET++ Class Hierarchy

Application classes are high level abstract classes that factor out the common control structure of application running in a graphic environment. They define the abstract model of a typical ET++ application and together form a generic ET++ application.

The Graphical Building Blocks contain all graphical and interactive components found in almost every user interface toolbox, such as menus, dialogs, scrollbars. In addition, it defines the framework to easily build new components from existing ones.

The System Interface Layer provides its own hierarchy of abstract classes for operating system services, window management, input handling, and drawing on various devices. Subclasses exists for implementing the system interface layer's functionality for various window systems and the UNIX operating system.

The Programming Environment part of ET++ system is a set of small set of ET++ applications that are automatically included in every ET++ application. They implement tools for inspecting the running application and browsing through its source code to help the developer understand the program.

One of the problems of C++ is that its run-time system does not provide any information about the class structure and instance variables of an object. Consequently, an additional mechanism has to be introduced to gather this information in order to support an IsKindOf method and for the object input/output facility. ET++ uses the approach of associating with each class a special object describing its structure. These descriptors are instances of the class Class which is itself a subclass of Object. In analogy to Smalltalk, they are called metaclasses. Metaclasses store the following information about a class:

- the associated superclass
- the name of the class
- the size of an instance in bytes

• the types of its instance variables, and

• a source code reference to the definition and implementation part of the class

The most important principle embodied in an application framework is to express an abstract design for a particular kind of application with a collection of abstract classes. These classes define the application's natural components and how they interact. This interaction is the main difference between an application framework and a collection of abstract but not strongly related classes. The former allows the factoring out of much of the control flow into the class library, while the latter requires that the developer know exactly when to call which method.

An abstract ET++ application consists of a single instance of the class Application which controls the application as a whole and manages any number of Documents. A Document is another abstract class that encapsulates the data structure or the model of an application and knows how to open and close documents and how to save them on disk. The implementation of opening and closing document is a very good example of factoring out the common control flow: consider, for example, an end user who intends to open new file without saving the old modified file to disk. The abstract class Document manages all necessary dialogs to ask the user if and where to save the old file and what file to open next. In addition, interactive applications have the notion of a current selection on which some operation triggered by the user will be performed. The class **View**, another subclass of **VObject**, represents an abstract and possibly arbitrarily large drawing surface. Its main purpose is to factor out all control flow necessary to manage rendering and printing as well as maintaining a current selection. A Document can have any number of **Views**, all showing the same model in various representations. This closely corresponds to the MVC model.

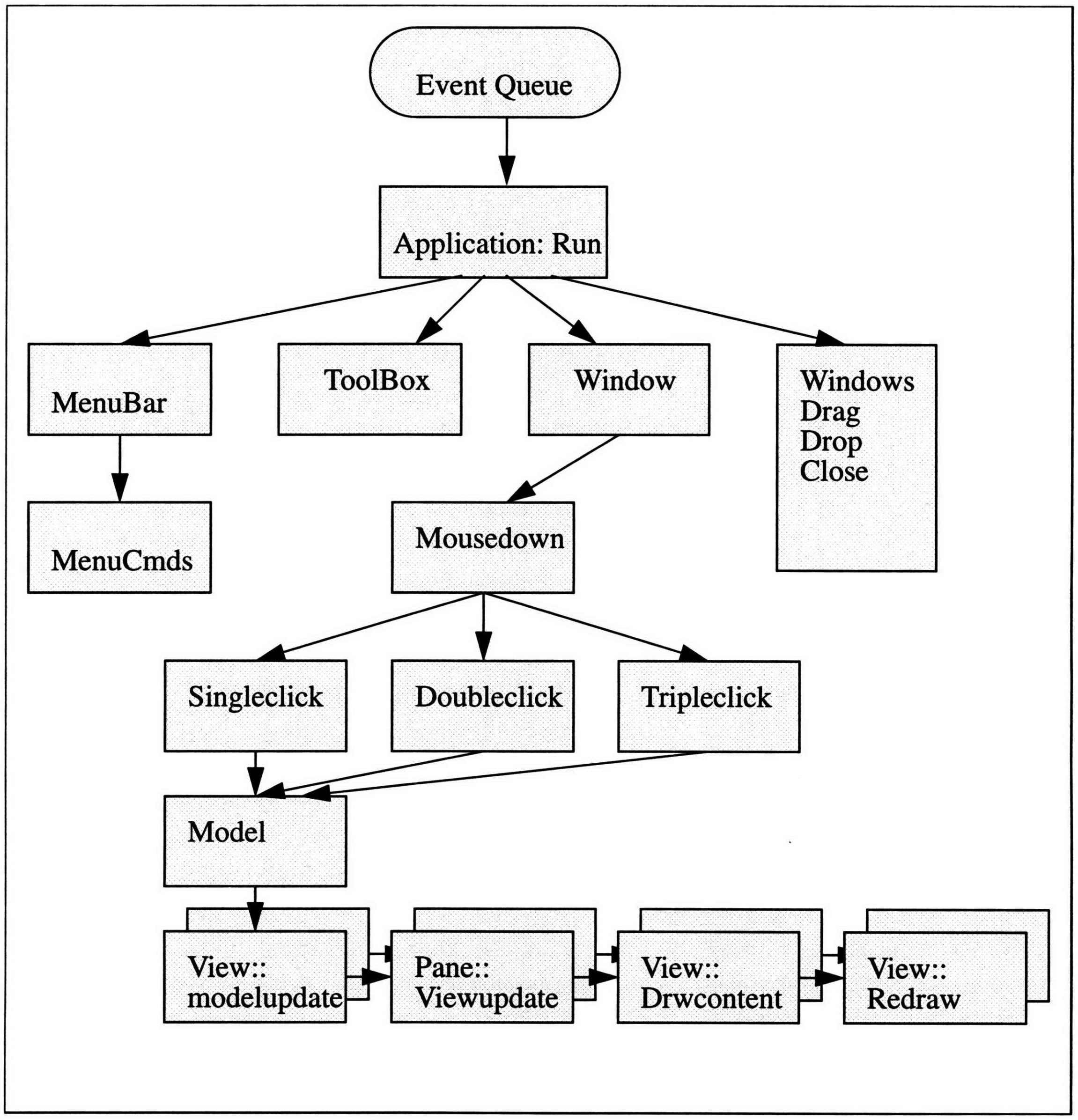


Figure 4.7: ET++ Event Looping Path

First experience with ET++ have shown that providing a class library with rich functionality considerably increases the learning time for programmers to fully exploit this functionality. For this reason the design and implementation of a programming environment including browsers for ET++ was initiated. Browsers and browser-like tools make inheritance more accessible and easy to use. ET++ programming environment includes a source code browser to access class definitions and their structure as well as an inspector to view object structure at run-time. Every ET++ application has those tools automatically built it; they execute in the application process. The browser or inspector can be invoked either at startup time or at any time while the application is running by pressing a special key combination.

Object Structure Browser

The Object Structure Browser visualizes runtime relationship between objects. It shows an object's part-of hierarchy. For example, Figure 4.8 shows the part-of hierarchy of the cabin design. A node represents an object and the thin lines connect the object with it parts. The object structure browser can be invoked from the inspector on the inspected object. Double clicking a node in the object structure browser shows the corresponding object in the inspector.

With the object structure browser, the designer is able to visualize the relationships of design components, check the result, and perform the redesign, if necessary. As a matter of fact, it is quite sophisticated to construct such a browser. First of all, we have to keep track of the meta-object information so that the browser knows the relationships between objects at run-time. As compared with the class hierarchy browser, it is a dynamic process, rather than static.

Experience with ET++/C++

The most interesting application of ET++ is certainly the ET++ programming environment whose user interface illustrates the high reusability of the ET++ classes. All of the inspector's user interface components, the source code browser, and the hierarchy viewer had already been part of the ET++ class library or were implemented as very simple subclasses thereof. An object-oriented application framework transfers a lot of control flow from client code into class library. This characteristic allows the addition of new functionality without any modifications in existing applications and with little programming effort.

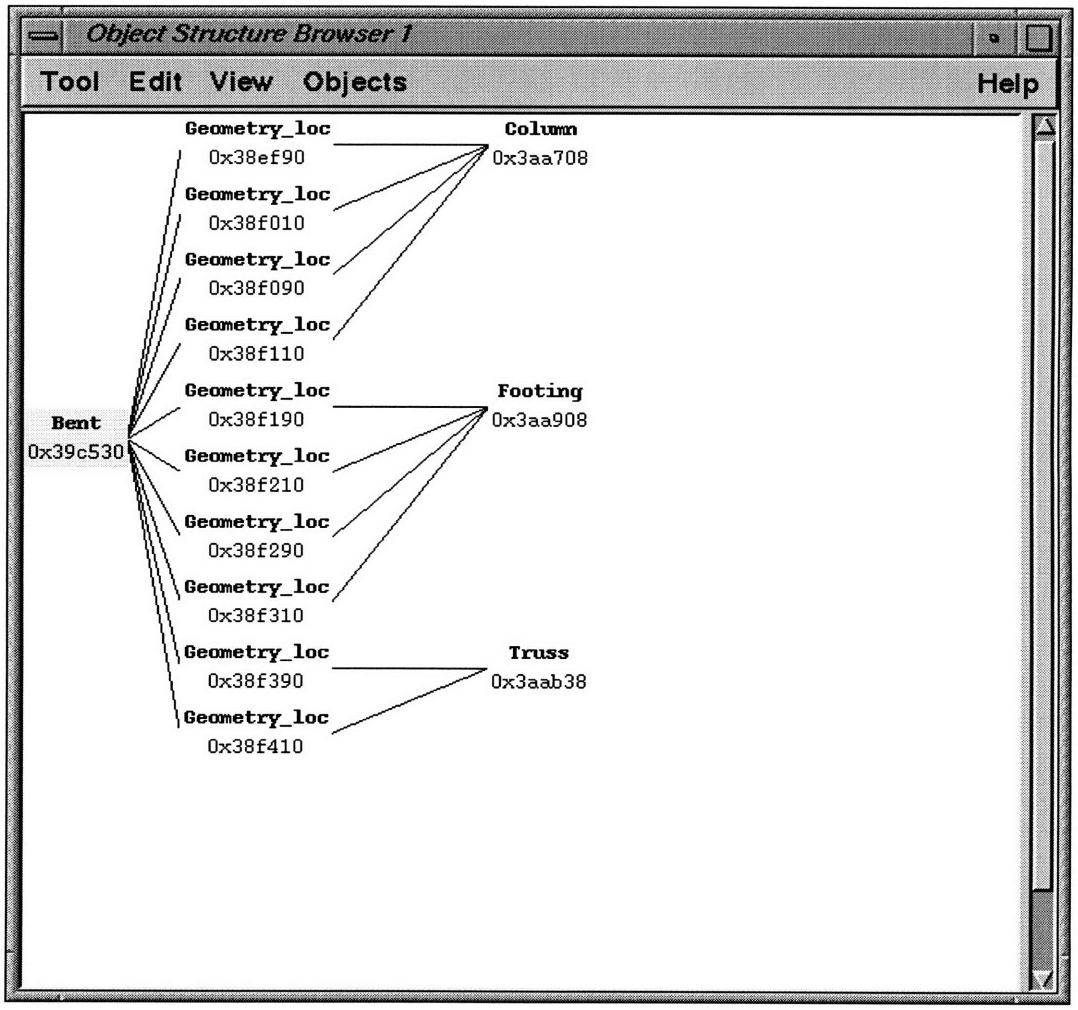


Figure 4.8: Object Structure Browser

Working experience with C++ has shown that having information about the classes and their structure available at run-time is not just convenient but even required. The approach of building additional descriptor objects manually or with the help of some complicated preprocessor macros is not very elegant. All of the information collected in such a metaclass object is in fact a subset of the C++ translator's compile time information 9e.g., its symbol table). As a logical consequence it would be very easy to automatically generate the necessary structures containing the meta-information. The difficult part is to agree upon what information is really needed. But without a consensus every library builder will most likely invent some new tricks and programming conventions making the exchange of classes between libraries much more difficult.

Integration of ET++ and ObjectStore

In order to demonstrate the relationships between objects, we utilize the ET++ class library for the construction of graphical user interface. The critical issue is to integrate the ET++ class library with ObjectStore so that the persistent design objects can be visualized on the screen. Besides, for the purpose of redesign and modification, the designer is also allowed to modify the slot value of the object whenever necessary. The integration of these two systems should take these issues into consideration.

Using ET++ and ObjectStore together is interesting because the combination of these two systems offers advantages none of them can provide on its own. The integration of ET++ and ObjectStore allows to easily to visualize the objects easily.

The goal of this work was to create a seamless integration of ET++ and ObjectStore. This meant to make as few changes as possible to the implementation of ET++ and especially to the protocol of ET++. With these tools,

- new classes can be derived from the root class Object whose instances can be allocated either in transient or persistent memory;
- the basic functionality of ET++ (runtime type information, change propagation, object cloning, input/output of objects, inspector) is also provided for persistent

instances.

- the already existing collection classes of ET++ plus new ones with the same semantics but an improved implementation are available for persistent usage.
- persistent classes work in read-only transaction as far as it is reasonable.

This is sufficient for writing new multi-user applications that store their data in an ObjectStore database.

4.5.1 Persistence in ObjectStore

Persistence in Objectstore is orthogonal to the type system because each instance of a class can reside in a different memory space, be it in the main memory or in one of several databases. Whether an object is persistent or transient is determined at creation time. In contrast, persistence is often inherited from a persistent root class. With ObjectStore, the main difference in the interface is the introduction of a second new operator that allocates the new object in a specified database.

Some points have to be considered when data structures for persistent usage are designed because they could prevent the class from being used in persistent memory:

- At the end of transaction, no pointers must point from a persistent object to a transient object because these pointers can not be reattached to an object when the persistent object is retrieved again. Some strategies on how to avoid them is described below.
- Because persistence is orthogonal to the type system in ObjectStore, it introduces a new dimension that has to be handled. Without persistence, all objects are created in the same memory space and a give pointer always points into the single memory space. With persistence, the programmer has to determine for each newly created

object where it is to be allocated. A pointer can be used safely in a persistent object unless it is made sure that it points to an address in the same memory space. If this rule is violated, the program is likely to crash. For complex objects - objects consisting of a root object and several subobjects - the general strategy should be to allocate all subobjects in the same database segment as the root object.

- Objects can not have a partially transient and a partially persistent state. There are no routines that are called when an object is retrieved from or stored in the database. So objects cannot be converted on the fly between the two states. All objects that have been allocated in the persistent memory must always be ready for persistent storage. For example, they must not contain any pointers to transient addresses.
- Every access to persistent data takes place within a transaction. So, before retrieving a root object in the database, a transaction must be started, and to make changes persistent the transaction must be committed/ Because the address of the persistent objects may change from transaction to transaction, the objects must be released at the end of the transaction and retrieved again at the beginning of the next transaction.
- It is desirable to perform read operations within read-only transactions to increase the potential throughput. With the pessimistic lock management of ObjectStore, no two transaction can modify the same object at the same time. It is not always easy to make sure that an object will not be changed during a transaction. Often, a temporary object is needed and a pointer to it is stored in the persistent object. Several strategies and techniques how to avoid such pointers are given below.
- ObjectStore uses the unix signal mechanism. In particular, it replaces the handler of

the "segmentation violation" signal. Therefore, it is rather difficult to trace down bugs that crash the application with the message "segmentation fault".

4.5.2 Implementation

To successfully use persistent ET++ objects, you have to do at least the following:

• Use **PersistentMetaDef** and **PersistentMetaImpl** macros.

• Use the extend **new** operator to allocate persistent objects in the desired database segment.

• Introduce transactions to your application.

• Add the new types to the database schema

4.5.3 Run-time Information about classes

C++ is not able to provide information about the dynamic type of an object or about its instance variables. ET++ requires this information for the object input/output facility and the programming environment. In order to preserve enough information, the programmer has to use the macros **MetaDef, NewMetaImpl,** etc., as is described in the ET++ documentation.

If the persistent objects of a class are to be created, a set of new macros must be used. The macro **PersistentMetaDef** replaces **MetaDef,** and **PersistentMetaImpl** and **PersistentMetaImpl0** replace the macros **NewMetaImpl** and **NewMetaImpl0.** For abstract classes and classes without persistent instances, the old macros can still be used.

If the new macros are used to provide the run-time information, we say the class is persistent although "persistently usable" would be a more precise item. A persistent class can still have transient instances, and static members are always transient. However, it must be declared in the schema file and it is entered into the schema of the database.

There is no drawback or performance penalty in declaring a class to be persistent. The objects are still the same size and still behave the same. A few new methods have however been added which support persistent allocation, persistent copying, etc.

The macro **PersistentMetaDef** is defined as follows,

```
#define _PersistentMetaImpl(name)
    Object *name::NewObj(os_segment *seg) {
        return new(seg, name::DBTypeSpec())
        name((_Object_dummy*)0, (_Object_dummy*)0); }
        os_typespec *name::GetDBTypeSpec() { return name::DBTypeSpec(); } \
```

4.5.4 Persistent Object Creation

A transient object is created as before:

```
coll = new OrdCollection(number_of_elements);
```

For persistent object creation, an extended **new** operator is provided. It takes two additional arguments: a database segment where the object is to be allocated and an ObjectStore type specification.

```
void * Object::operator new (size_t sz, segment *seg, os_typespec *type);
```

The argument are the same as for the general new operator provided by ObjectStore. Further information on these two arguments can be found in the ObjectStore manuals. Arrays of ET++ object are not supported. A small but useful extension to the ObjectStore semantics has been made: if either **seg** or **type** are null, the new object is created in transient memory and treated like a proper ET++ object i.e., entered in the object statistics etc. In fact, the extended **new** operator calls the original ET++ **new** operator.

If a new object of type **OrdCollection**is to be created in the same database segment as the this-object, it is done as following:

coll = new (DBSegment(), OrdCollection::DBTypeSpec()) OrdCollection(numnber_of_elements);

Because the new object is often allocated in the same database segment as an existing one, a short-cut exists:

coll = DBNEW(OrdCollection)(number_of_elements);

It only works in member functions because it relies on the **this** variable to determine the database segment where the object resides and created the new object at the same place. To use the macro **DBNEW,** the file **OS-Types.h** must be included. Beside being much shorter and less error-prone, it works for transient and persistent objects, If the **this**-object is in transient memory, the new object will also be allocated there.

4.5.5 Object Deletion

Persistent objects are deleted like transient objects:

delete coll;

The delete operator transparently handles the transient and persistent case, so the programmer does not have to care about persistence when deleting an object.

4.5.6 Changes to the class Object

Persistence introduces a new dimension. Each function that creates a new object, has to explicitly specify where it is to be allocated. Therefore, the semantics of several functions had to be extended and their implementation changed. Furthermore, to comfortably and efficiently handle persistent objects, additional information is sometimes needed. This is provided by some new member functions.

4.5.7 Inquiries about persistence

Although all functions uniformly work on transient and persistent objects, it is sometimes necessary to distinguish between transient and persistent objects:

bool Object::IsPersistent();

IsPersistent returns **TRUE** if the object has been allocated in persistent memory, **FALSE** otherwise.

To exactly find out in which database or database segment an object was created, **DBase** and **DBSegment** are used:

*segment * Object::DBSegment();*

*database *Object::DBase();*

If the objects reside in transient memory, null is returned, which is a legal parameter to all functions that expect a database segment. The use of **os_segment::of** and **os_database::of** is discouraged since they produce unexpected results if applied to transient objects.

4.5.8 ObjectStore schema information

For ObjectStore to work, it need the complete schema information of all persistent objects. Since C++ cannot provide it, it must be provided by the programmer when an object is created. The type of an object is specified as an argument of type **os_typespec**, which is an ObjectStore data structure in transient memory.

To simplify the handling of the type information, two new member functions have been added:

*static os_typespec *Object::DBTypeSpec();*

*virtual os_typespec *Object::GetDBTypeSpec();*

Both functions return an object of type **os_typespec**. The first one is a static function and is mainly used for the creation of objects when the dynamic type is known but the object does not yet exist. The second one is a virtual method. It is applied to already existing objects when the dynamic type is not known at compile time. The returned objects will be delete automatically. These routine only work for class for which **PersistentMetaDef** and **PersistentMetaImpl** has been used.

4.5.9 Change Propagation

Change propagation works for both transient and persistent objects although the relationships are only temporary. They are temporary because they are always stored in transient memory, and they must be redefined in each transaction. This implementation is sufficient to fully support an MVC structure. In a typical application, the model is persistent while the viewers are transient objects.

4.6 Integration of CLIPS and ObjectStore

CLIPS (C Language Interface Programming System) is an rule-based programming environment for building knowledge-based expert systems. This language is famous for its high portability between different platforms and flexibility to add new application-dependent functions.

4.6.1 Object-oriented CLIPS

The latest version of CLIPS has incorporated the object-oriented concepts into the language itself. The motivation of this incorporation are as follows:

- Rules alone are insufficient when a deep understanding of the underlying principles is required.
- Combination of rules and OOP constructs provides a versatile and natural method for modeling states of a problem.

There are several feature in combining rules and objects. Object patterns are just like template fact patterns, except:

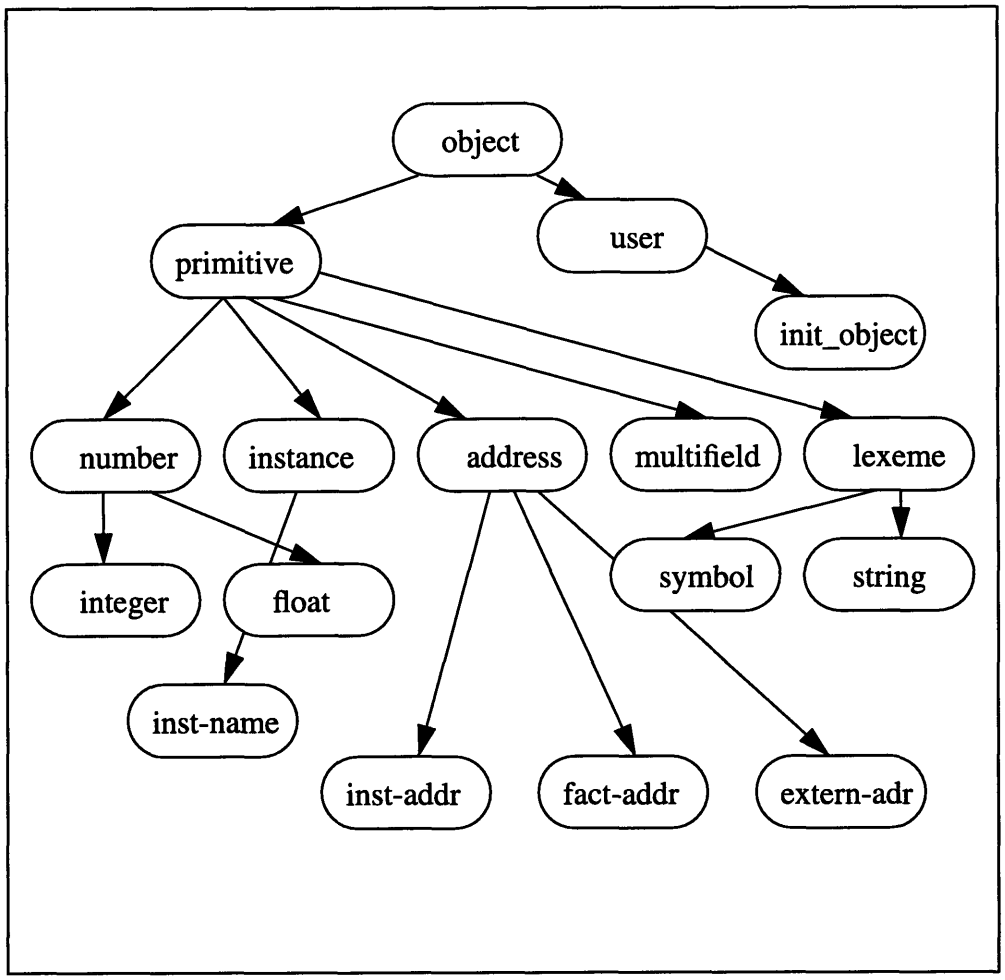


Figure 4.9: Predefined CLIPS class hierarchy

- Rules can match only on classes of objects which were defined (loaded) before the rule. A class and any slots being matched against must be "reactive."
- CLIPS uses as much static information (slot existence, slot constraints, etc.) about classes as possible to "preprocess" the pattern, i.e., restrict it to as few classes of objects as possible a priori.
- Object patterns make use of inheritance, i.e., a pattern matching on a superclass can match instances of a subclass unless specifically excluded in is-a constraint.

• Unlike facts, slot changes are done in place. This is the largest performance benefit of using objects with rules because object slot changes do not affect patterns which do not explicitly match on the slot.

• Special restrictions are allowed (similar to slot patterns) for the class and name of an object (is-a and name respectively).

4.6.2 User-defined Functions

The prorgrammer can define application-specific functions and embed them into CLIPS easily. The DefineFunction2 function is used as follows:

DefineFunction2 Parameters:

• The name of the function in CLIPS.

• The return value type of the function (always use the most specific return type).

• A pointer to the function.

• The name of the corresponding C function (used by the constructs-to-c command).

• The argument restrictions used by the constraint checker (minimum number of arguments, maximum number of arguments, default argument type, and specific argument types).

4.6.3 Database-related Functions

A set of database-related functions have been defined to connect the CLIPS inference engine with ObjectStore database management system.

• extern int dbopen(); // open the database

• extern int dbclose(); // close the database

• extern int dbmethod(); // invoke an method of an object

• extern int dbquery(); // send a query string to database

- extern int dbretrieve(); // retrieve an object from database
- extern int dbsave(); // save the database
- extern int dbmake_instance(); //create an instance on database

Experience with CLIPS

- For the purposes of memory management and garbage collection, symbols, strings, and instance names are stored in a symbol table to prevent duplication and to keep track of the number of references to each data value.
- Storing a permanent pointer to the string is dangerous because the string may later be garbage collected invalidating the original pointer to the string.
- You can always change the type, value, begin, and end values of a "data object."
- Don't intersperse calls to functions like Reset and Run with calls that access string arguments since these types of functions can cause garbage collection to occur.
- Don't forget to allocate memory for "data objects" when necessary.

Many programs unintentionally rely on the order of rule execution for correct results.

```
(defrule rule-1
    ?f <- (a) =>
    (retract ?f))
(defrule rule-2
    (a) =>
    (assert (b)))
```

In the preceding example, if rule-1 fires first, rule-2 will never be executed.

- In a robust rule-based system, the execution order of rules of equal salience should not matter. The ease of maintenance of a rule-based system is highly dependent upon this assumption.

- Use the random conflict resolution strategy to vary the placement of rules of similar salience on the agenda.
- The execution order of rules using the random conflict resolution strategy is reproducible either by restarting CLIPS or using the seed function before a run.

Modules

- Use modules for execution control. Modules allow you to explicitly indicate control relationships between rules.
- Use modules to limit the visibility of constructs and facts/instances that are not pertinent to other modules.

Always Use Constraints

- The use of ordered facts should be avoided. Always use deftemplates and defclasses so that type and value constraints can be applied to the slots.
- Constraints help to detect typos, illegal slot values, illegal arguments to functions, and unmatchable patterns.
- Dynamic constraint checking is generally only useful for testing purposes since there is no recovery mechanism.

4.6.4 Where CLIPS Deviates from Pure OOP

- Not everything in CLIPS is an object, i.e., only the primitive types (symbol, integer, etc.) and instances of user-defined classes are objects. Other data items, such as constructs (facts, rules, classes, etc.) must be manipulated in the traditional non-OOP manner (i.e., through specialized functions, e.g., ppdefrule).

- Instance-set queries and rules can read object attribute values directly (i.e., without message-passing) thus compromising encapsulation to some degree to benefit performance.
- Message-passing must still be used to manipulate bound instances on the RHS of rules or the actions of instance-set queries.
- Encapsulation is provided by message-handlers. Instances must be manipulated with messages. Only message-handlers may directly read (with the two exceptions already noted) and write the slots of an instance. CLIPS can create default message-handlers for the basic reading and writing of slots with the create-accessor slot facet
- An instance is created with the special function, make-instance, which sends the new object the init message and places slot overrides with put- messages. The default system init message-handler writes class default values for slots if no override was provided.
- All reading/writing of slots and deletion of instances must be done with messages.

The structural design rules written in CLIPS is listed in Appendix A.

Chapter 5

Examples

5.1 Cabin Design Example

This simple example demonstrates the design and decision-making process of the structural agent. Figure 5.1 shows the cabin plan. First of all, the structural agent needs to set up the environment before receiving any information needed to initiate the design process from the facilitator, including create the database, setting up the workspace, connecting to the facilitator, and register the agent's interests, see Figures 5.2 and 5.3.

The design process starts when the project manager agent and owner agent send their information to the design agents. Basically, the structural agent receives information, such as project budget and soil conditions at this point. The architectural agent (AA) starts to generate the problem statement and an initial problem solution. After generating the initial solution, the AA displays it graphically for inspection as a 3-D configuration and sends the information to the facilitator.

The SA will be notified when certain information is received, translated, and forwarded by the facilitator, see Figure 5.4. These messages will be queued automatically and asynchronously without the user's intervention. In additions, they will not be incorporated into the agent's database until the designer issues the import object command from the console's menubar, see Figure 5.5. Figure 5.6 shows the information received from facilitator in OML syntax. The structural agent will create these objects as persistent objects in the database. The designer can save these objects in the object database for future use through committing a transaction.

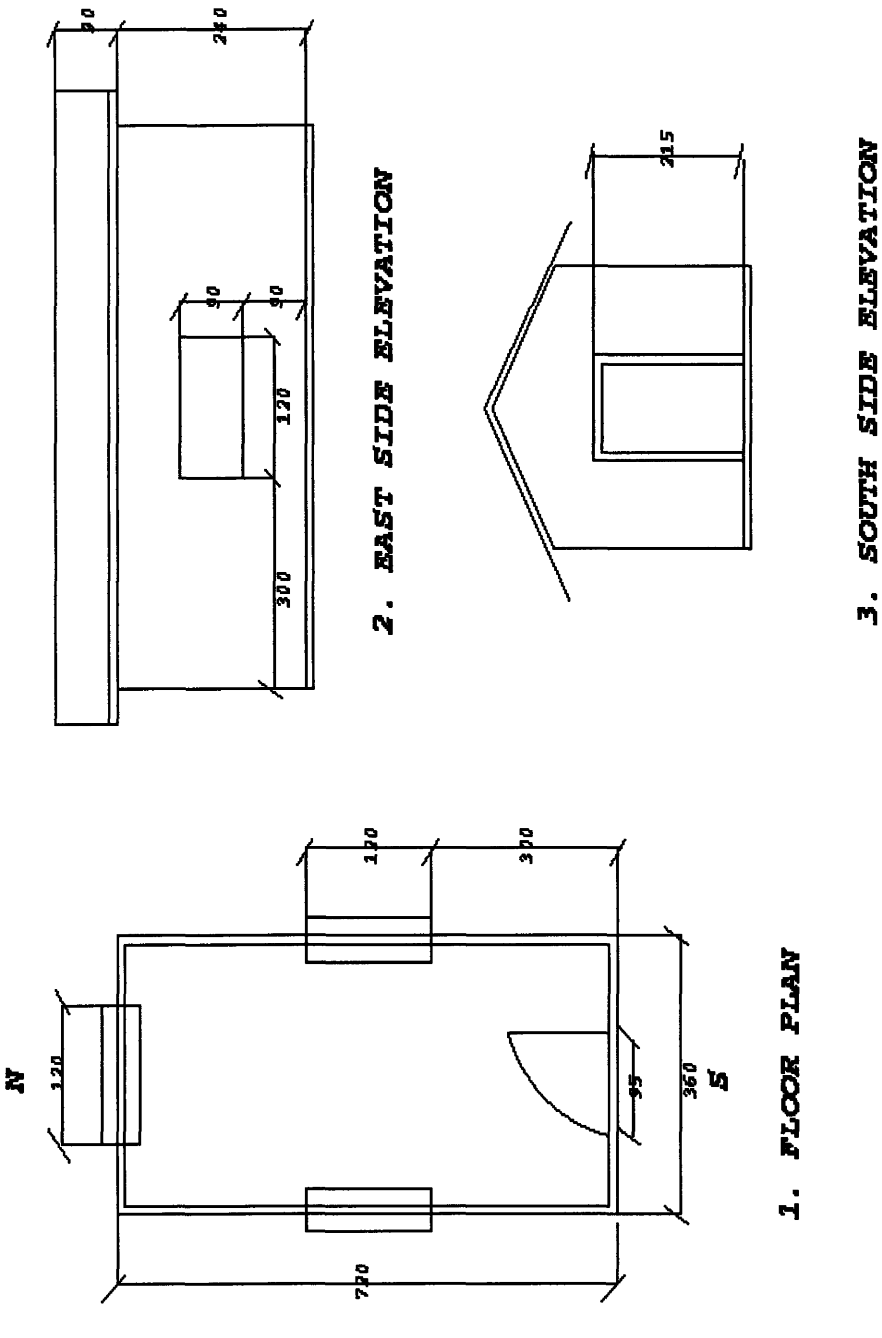
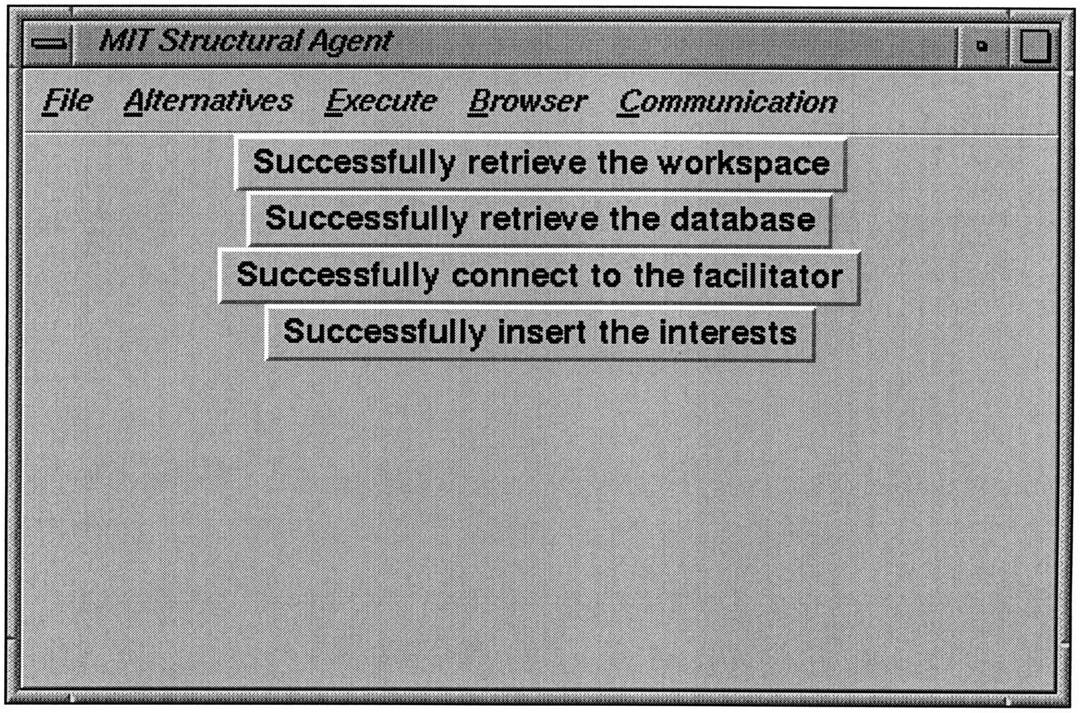
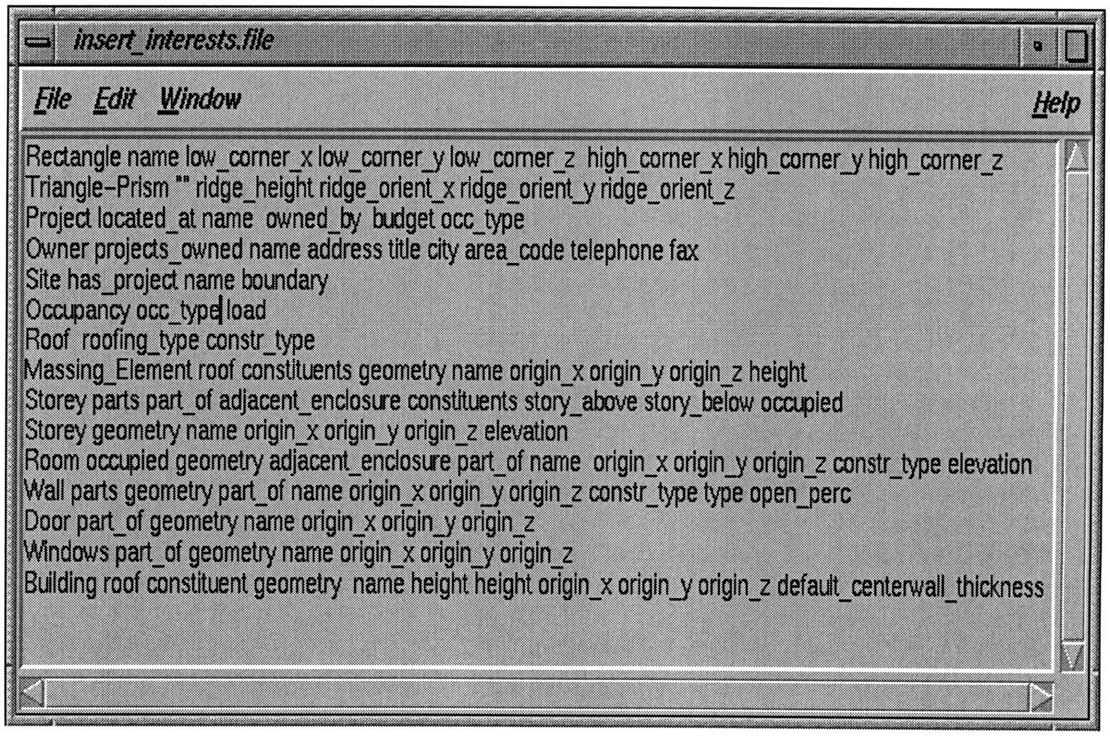


Figure 5.1: Cabin Plan



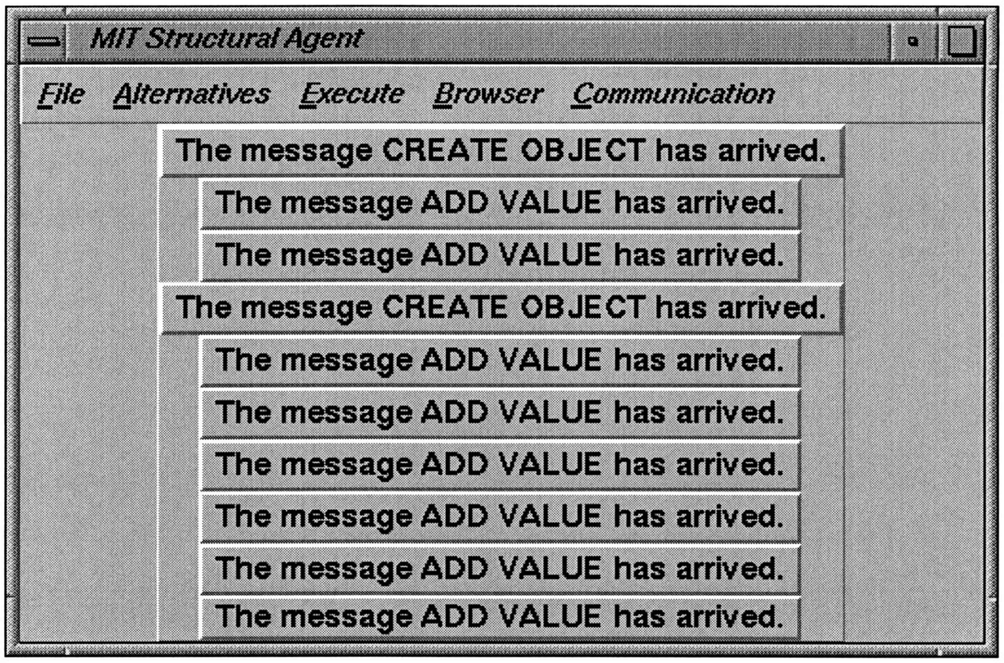
Successfully retrieve the workspace
Successfully retrieve the database
Successfully connect to the facilitator
Successfully insert the interests

Figure 5.2: Setup the working environment



```
Rectangle name low_corner_x low_corner_y low_corner_z  high_corner_x high_corner_y high_corner_z
Triangle-Prism "" ridge_height ridge_orient_x ridge_orient_y ridge_orient_z
Project located_at name  owned_by  budget occ_type
Owner projects_owned name address title city area_code telephone fax
Site has_project name boundary
Occupancy occ_type load
Roof  roofing_type constr_type
Massing_Element roof constituents geometry name origin_x origin_y origin_z height
Storey parts part_of adjacent_enclosure constituents story_above story_below occupied
Storey geometry name origin_x origin_y origin_z elevation
Room occupied geometry adjacent_enclosure part_of name  origin_x origin_y origin_z constr_type elevation
Wall parts geometry part_of name origin_x origin_y origin_z constr_type type open_perc
Door part_of geometry name origin_x origin_y origin_z
Windows part_of geometry name origin_x origin_y origin_z
Building roof constituent geometry  name height height origin_x origin_y origin_z default_centerwall_thickness
```

Figure 5.3: Declare agent's interests



The message CREATE OBJECT has arrived.
The message ADD VALUE has arrived.
The message ADD VALUE has arrived.
The message CREATE OBJECT has arrived.
The message ADD VALUE has arrived.
The message ADD VALUE has arrived.
The message ADD VALUE has arrived.
The message ADD VALUE has arrived.
The message ADD VALUE has arrived.
The message ADD VALUE has arrived.

Figure 5.4: Notify the designer that data is being received.

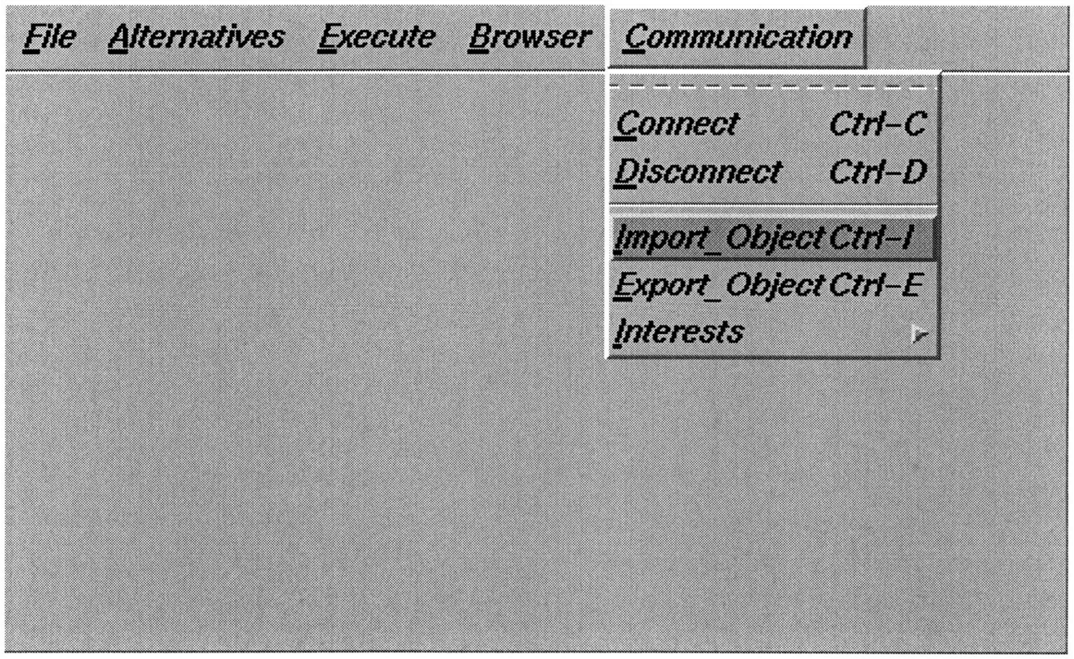
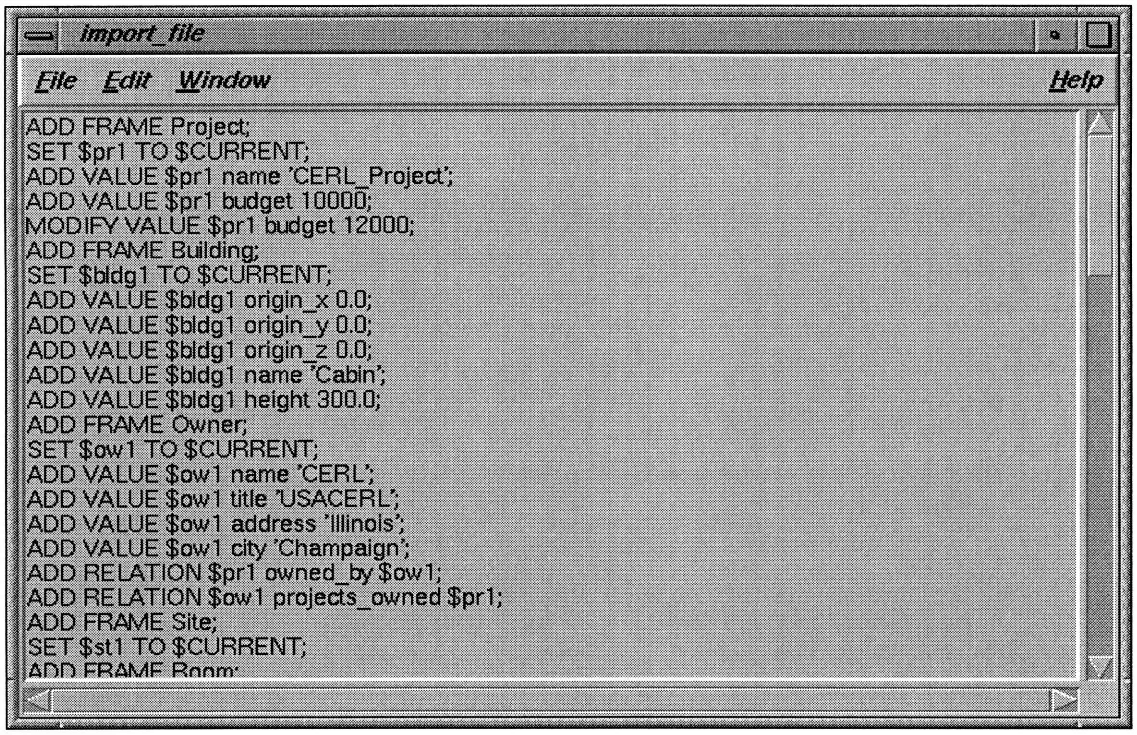


Figure 5.5: Incorporate objects from message queue to database.



```
ADD FRAME Project;
SET $pr1 TO $CURRENT;
ADD VALUE $pr1 name 'CERL_Project';
ADD VALUE $pr1 budget 10000;
MODIFY VALUE $pr1 budget 12000;
ADD FRAME Building;
SET $bldg1 TO $CURRENT;
ADD VALUE $bldg1 origin_x 0.0;
ADD VALUE $bldg1 origin_y 0.0;
ADD VALUE $bldg1 origin_z 0.0;
ADD VALUE $bldg1 name 'Cabin';
ADD VALUE $bldg1 height 300.0;
ADD FRAME Owner;
SET $ow1 TO $CURRENT;
ADD VALUE $ow1 name 'CERL';
ADD VALUE $ow1 title 'USACERL';
ADD VALUE $ow1 address 'Illinois';
ADD VALUE $ow1 city 'Champaign';
ADD RELATION $pr1 owned_by $ow1;
ADD RELATION $ow1 projects_owned $pr1;
ADD FRAME Site;
SET $st1 TO $CURRENT;
ADD FRAME Room;
```

Figure 5.6: Information received from facilitator

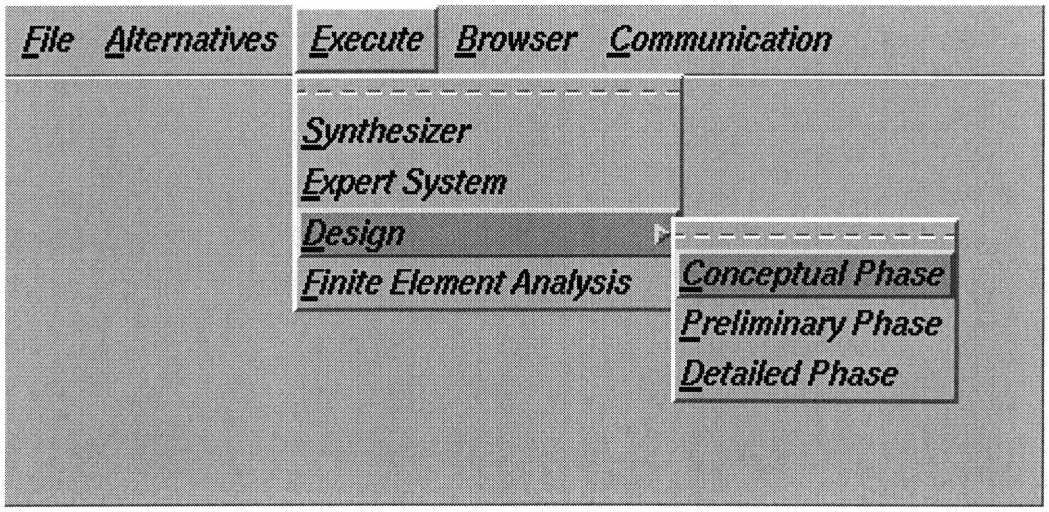


Figure 5.7: Starting conceptual design

Conceptual Design Phase

Based on the input from OA, PMA, and AA, we start the conceptual phase for structural design. The SA will generate the configuration as shown in Figure 5.8. From this figure, we see there are two **Struct_system** alternatives generated. The structural agent will send a recommended structural system, which is a frame structure, and an alternative one, which is a bearing wall design, to the facilitator. Detailed attributes of these objects may be seen by Inspector (see Figure 5.10). Presumably, the agent will pursue the recommended system if the architect or project manager has no objection to this choice.

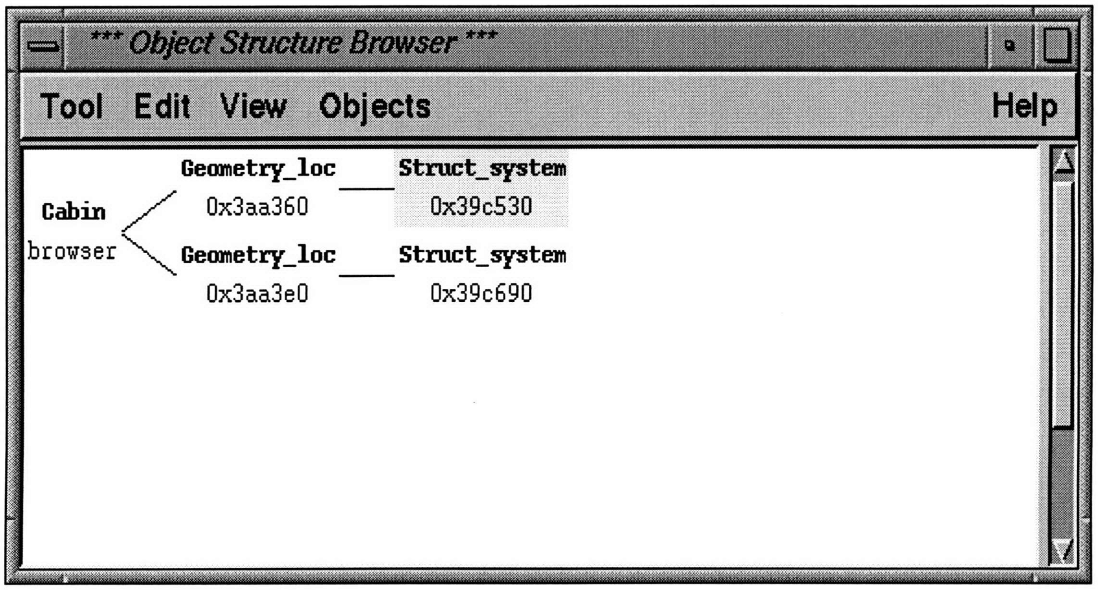


Figure 5.8: Conceptual configuration of Cabin.

Preliminary Design Phase

After AA approves the structural system, we proceed to Phase II for preliminary design. For the design of steel structures, one must consider: the roof slope; the roof decking material; the grade of steel to be used; the structural arrangement of the main supporting members; the roof framing plan; column design; and the bracing of the entire system.

The SA will generate a configuration as pictured Figure 5.9. From this figure, we see two physical **Bent** objects being created. One is the interior bent, the other the exterior; both are used in two distinct geometric locations. Each bent object contains physical **Column, Footing, and Truss** objects. These physical design objects are placed in different geometric locations relative to their parent objects. The above system (Figure 5.9) will be sent to AA for approval. The SA initiates negotiation with AA for any modification of shape or location of structural components and determines if a redesign is required. Figures 5.10 and 5.11 shows the inspector, which gives the attribute values and also allows the designer to access and modify the value.

There are two special attributes shown in Figure 5.12 which are included in each design object. One is the modified_since_last_transmission whose value is 1 if any attribute of the current object or any of its aggregation parent objects has been modified, since this agent's last transmission of information to facilitator. If no modifications have been made, the valie is 0. The other is modified_since_last_design whose value is 1 if any attribute of the current object or parent objects has been modified, since the agent's last design or redesign has been performed. If no attributes have been modified, the value is 0. These two attributes are important for the designer to know if the current object's slot value has been updated properly or not. In terms of collaborative design, the agent should propagate the changes it makes to other parties. The design agent should realize the status of the object in a global sense. That is, it should determine whether or not if the current design is synchronous with others. This feature offers a solution to the problem.

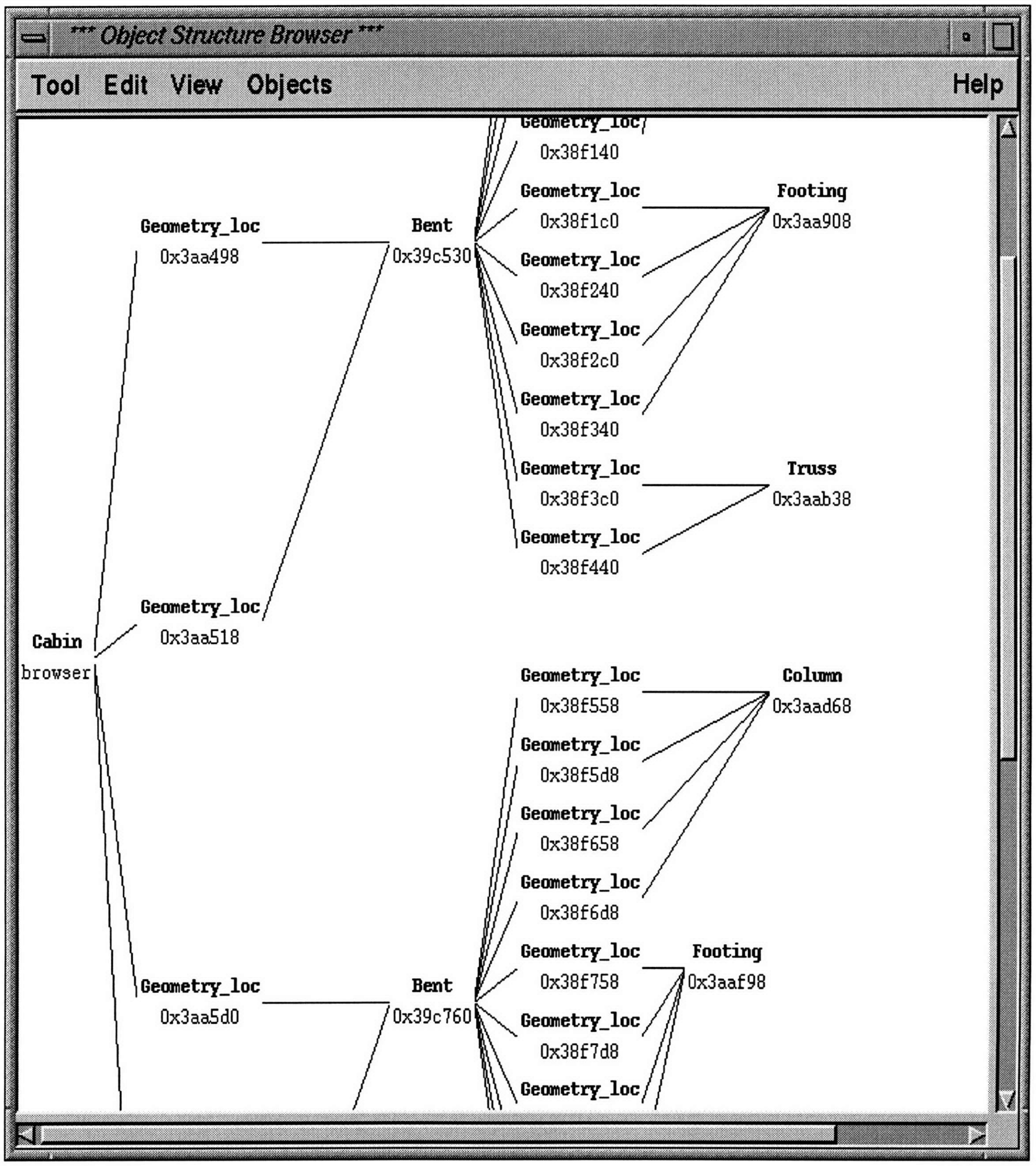
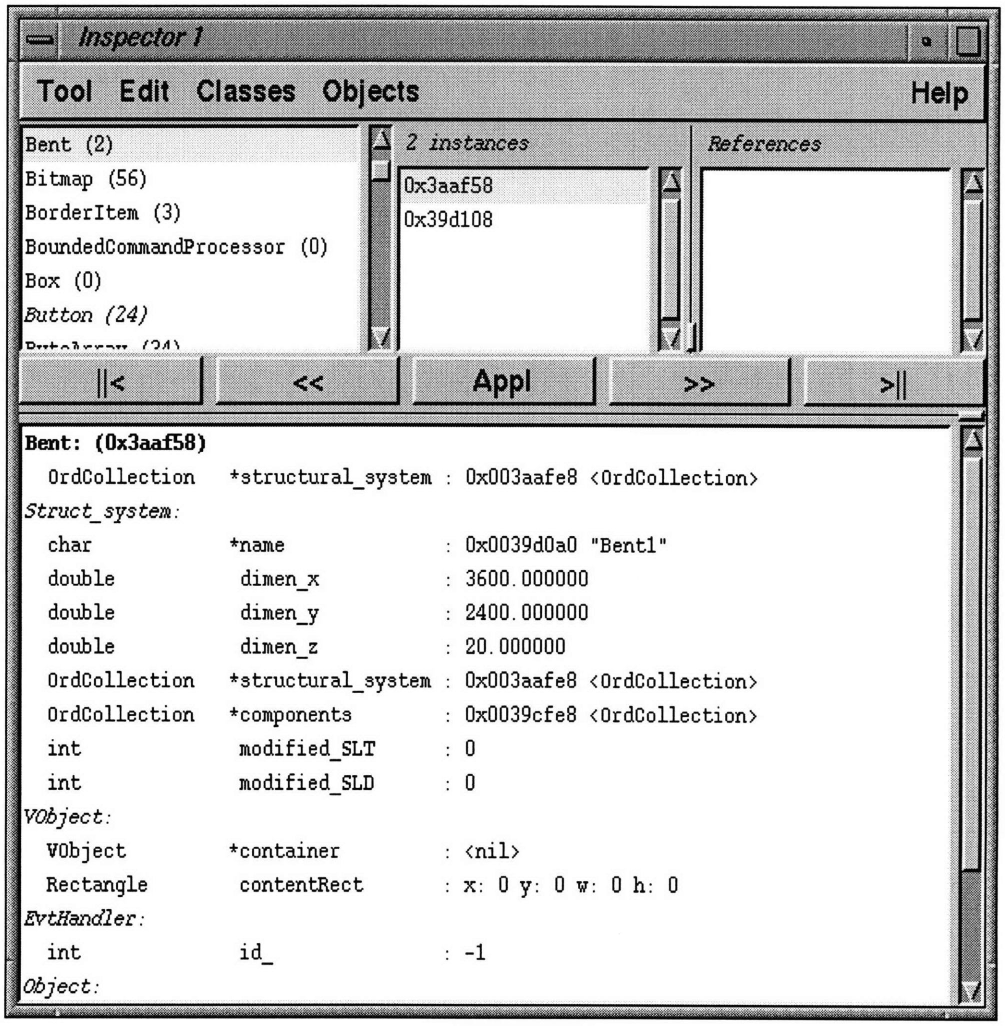


Figure 5.9: Preliminary Design for Cabin

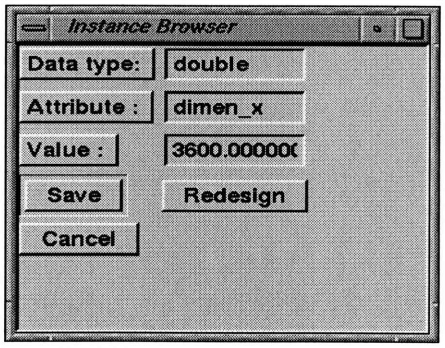


```
Bent (2)
Bitmap (56)
BorderItem (3)
BoundedCommandProcessor (0)
Box (0)
Button (24)
```

```
0x3aaf58
0x39d108
```

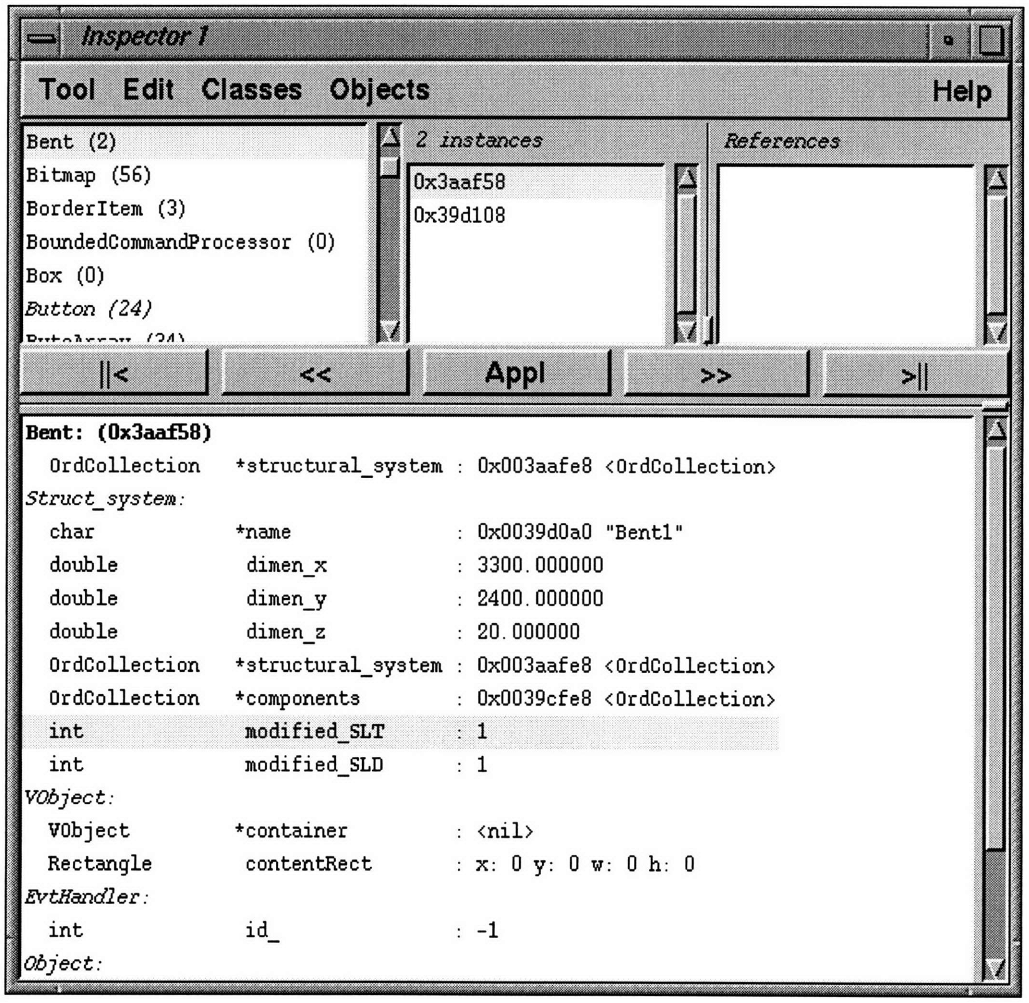
```
Bent: (0x3aaf58)
  OrdCollection   *structural_system : 0x003aafe8 <OrdCollection>
Struct_system:
  char            *name              : 0x0039d0a0 "Bent1"
  double           dimen_x           : 3600.000000
  double           dimen_y           : 2400.000000
  double           dimen_z           : 20.000000
  OrdCollection   *structural_system : 0x003aafe8 <OrdCollection>
  OrdCollection   *components        : 0x0039cfe8 <OrdCollection>
  int              modified_SLT      : 0
  int              modified_SLD      : 0
VObject:
  VObject         *container         : <nil>
  Rectangle        contentRect       : x: 0 y: 0 w: 0 h: 0
EvtHandler:
  int              id_               : -1
Object:
```

Figure 5.10: Inspector shows the attributes of Bent.



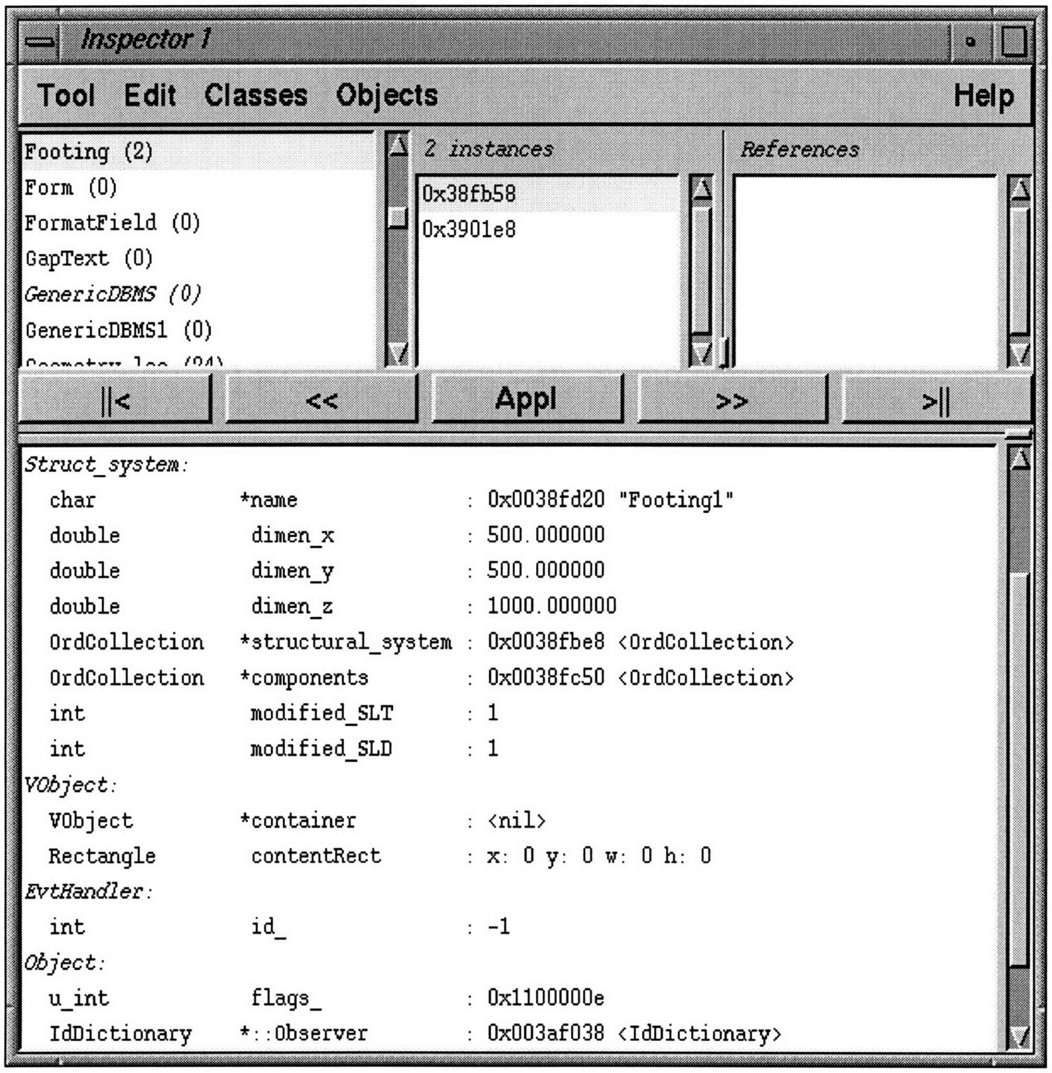
Data type:	double
Attribute :	dimen_x
Value :	3600.00000(

Figure 5.11: Instance Browser



```
Bent: (0x3aaf58)
  OrdCollection   *structural_system : 0x003aafe8 <OrdCollection>
Struct_system:
  char            *name              : 0x0039d0a0 "Bent1"
  double           dimen_x           : 3300.000000
  double           dimen_y           : 2400.000000
  double           dimen_z           : 20.000000
  OrdCollection   *structural_system : 0x003aafe8 <OrdCollection>
  OrdCollection   *components        : 0x0039cfe8 <OrdCollection>
  int              modified_SLT      : 1
  int              modified_SLD      : 1
VObject:
  VObject         *container         : <nil>
  Rectangle        contentRect       : x: 0 y: 0 w: 0 h: 0
EvtHandler:
  int              id_               : -1
Object:
```

Figure 5.12: Modified Since Last Transmission and Design.



```
Footing (2)
Form (0)
FormatField (0)
GapText (0)
GenericDBMS (0)
GenericDBMS1 (0)
```

```
0x38fb58
0x3901e8
```

```
Struct_system:
  char            *name              : 0x0038fd20 "Footing1"
  double           dimen_x           : 500.000000
  double           dimen_y           : 500.000000
  double           dimen_z           : 1000.000000
  OrdCollection   *structural_system : 0x0038fbe8 <OrdCollection>
  OrdCollection   *components        : 0x0038fc50 <OrdCollection>
  int              modified_SLT      : 1
  int              modified_SLD      : 1
VObject:
  VObject         *container         : <nil>
  Rectangle        contentRect       : x: 0 y: 0 w: 0 h: 0
EvtHandler:
  int              id_               : -1
Object:
  u_int            flags_            : 0x1100000e
  IdDictionary    *::Observer        : 0x003af038 <IdDictionary>
```

Figure 5.13: Footing Object is also being marked modified.

5.2 Firestation

The second example given in this chapter is a firestation design, see Figure 5.14. As we see from Figure 5.15, the architectural plan is more complicated than in the cabin example. But from the structural engineering perspective, since it is also a single story building, there is not too much difference between these two cases.

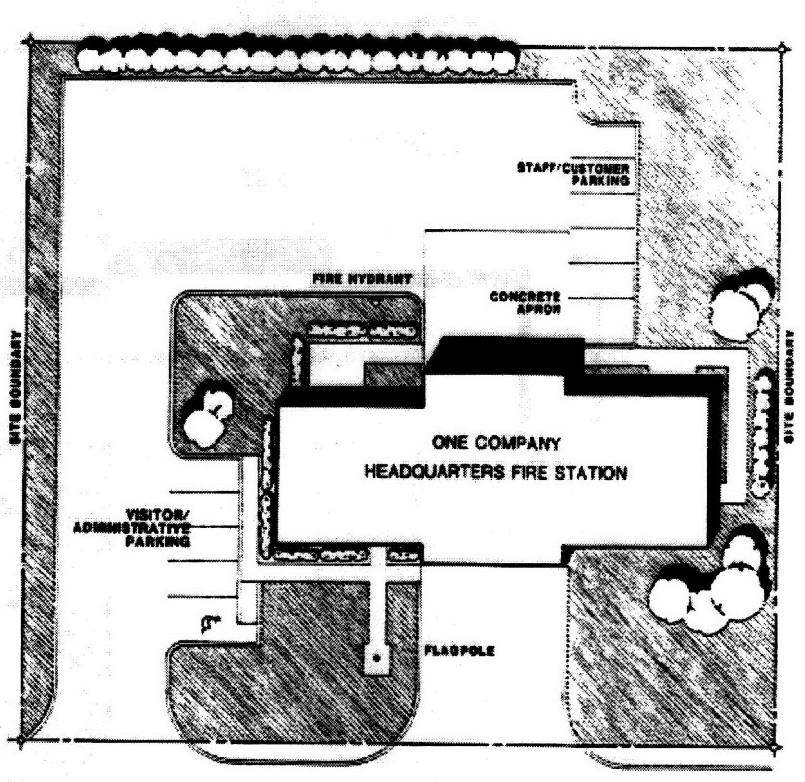


Figure 5.14: One Company Firestation

There are three designs for a firestations in the specifications, including one-company satellite; one-company headquarter; and two-company headquarter. In the following sections, we use the one-company headquarter design as the example. From an architectural point of view, the headquarter is seen as a massing element, while the branch is regarded as another massing element. The functionality of these two distinct massing element is quite different.

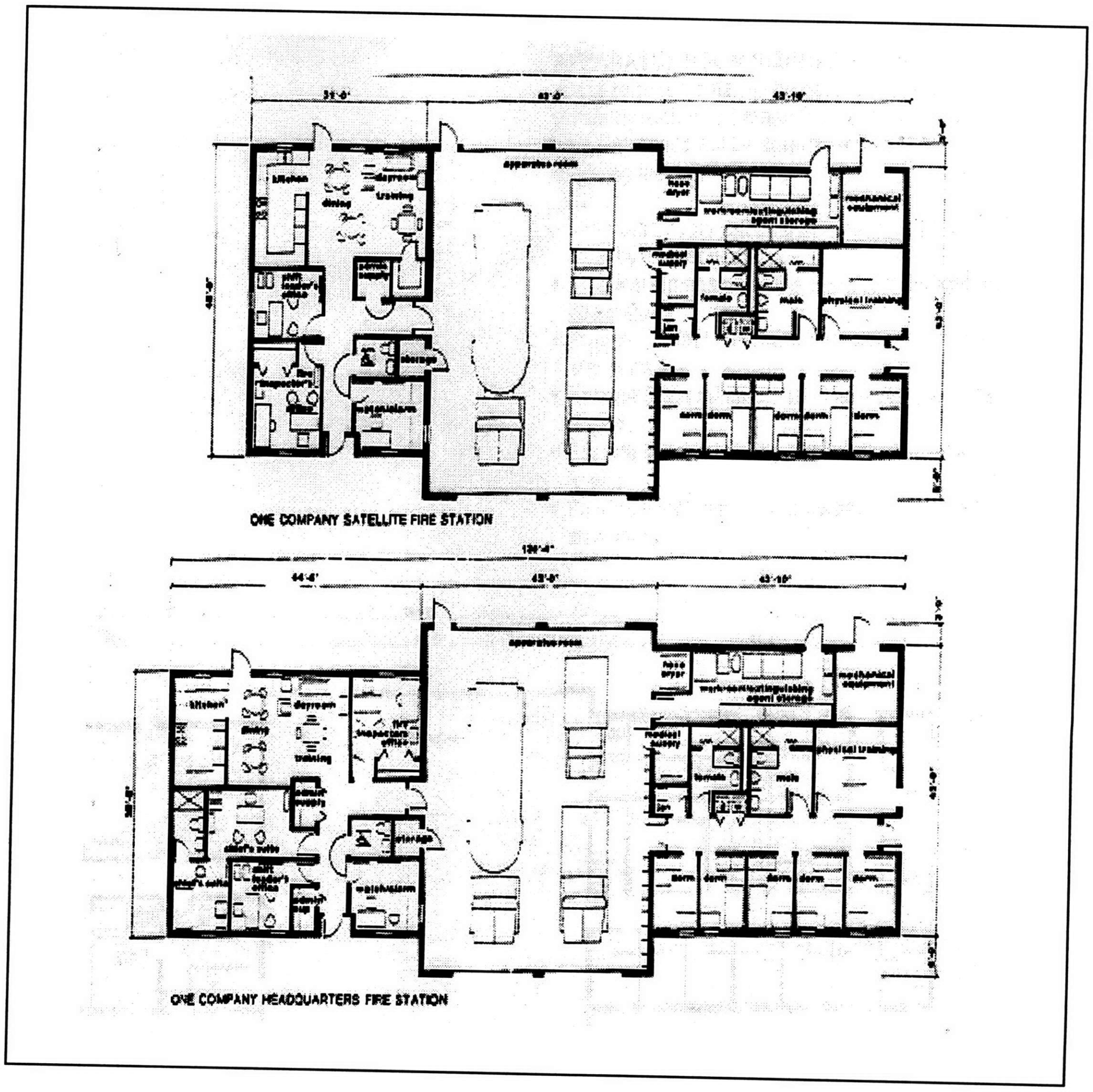


Figure 5.15: One Company Fire Station Plan

Conceptual Design Phase

The conceptual design phase is quite similar to the cabin case. After receiving the architectural requirements from the facilitator, the structural agent generates two conceptual structural framework (one is recommended, the other alternative) and sends both schemes to the facilitator. In this case, as seen in Figure 5.16, a conceptual structural scheme which recommends a frame structural system will be generated in this phase. The other alternative scheme is a bearing wall structure. Both systems will be sent to the facilitator for approval from other agents.

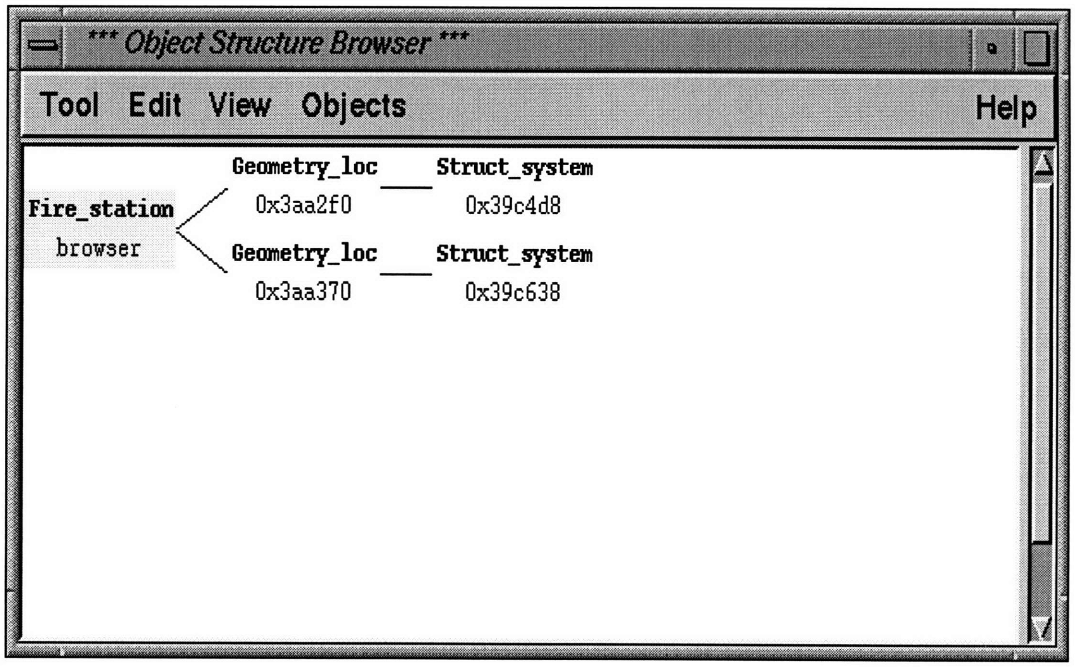


Figure 5.16: Conceptual Configuration for Fire Station

Preliminary Design Phase

In this phase, the two branches of the firestation are identified as the same structural system put into different locations due to the symmetry. Figure 5.18 shows the hierarchy of the structural systems and components. The structural system for a one-company headquarter consists of two bents which intersect with each other at a right angle. The structural system for the branch contains 6 bents separated in equal distance. Each bent has two columns, two footings, and a truss system. Basically, the design of the branch is quite similar to the cabin design example.

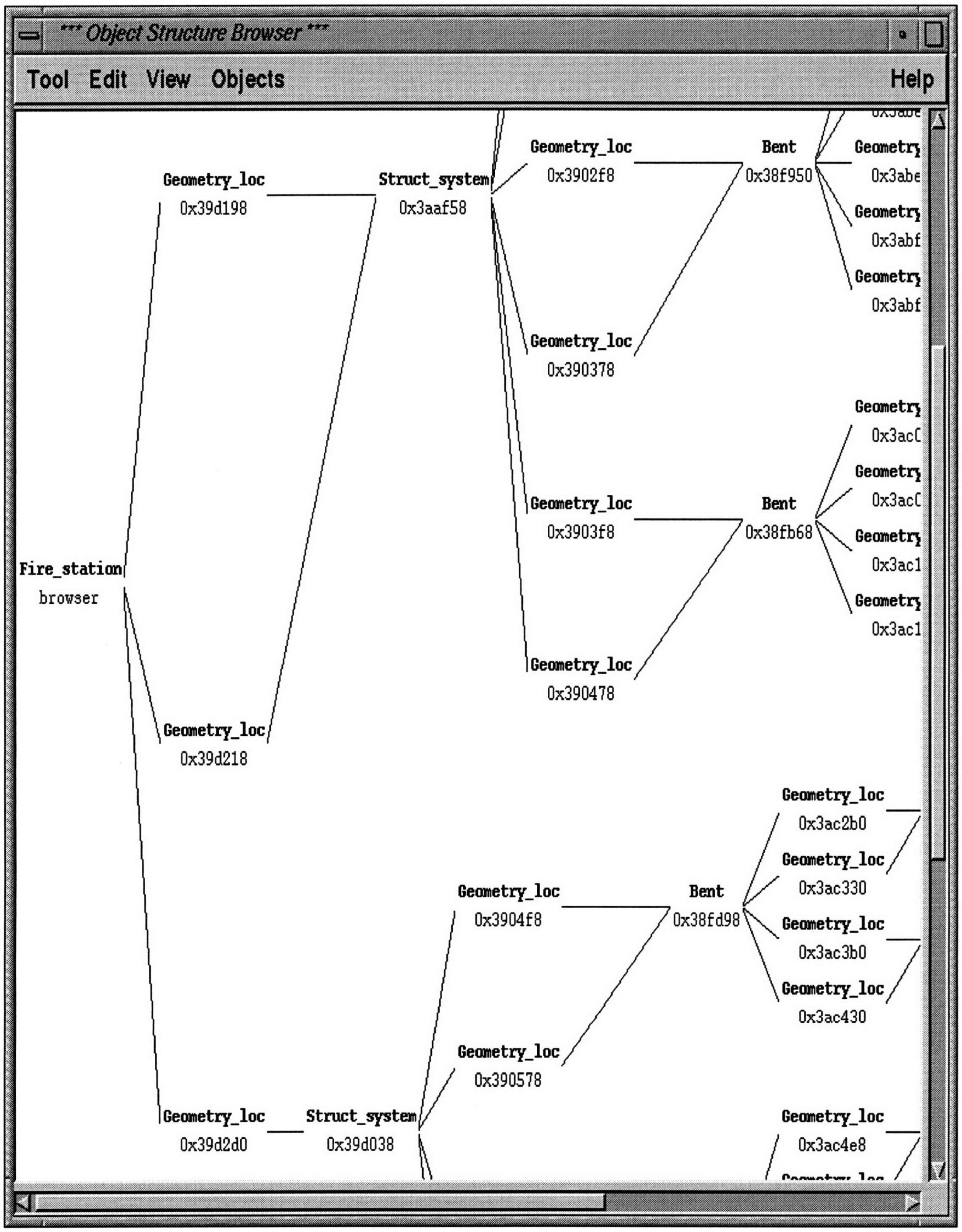


Figure 5.17: Preliminary Structural Configuration for Fire Station

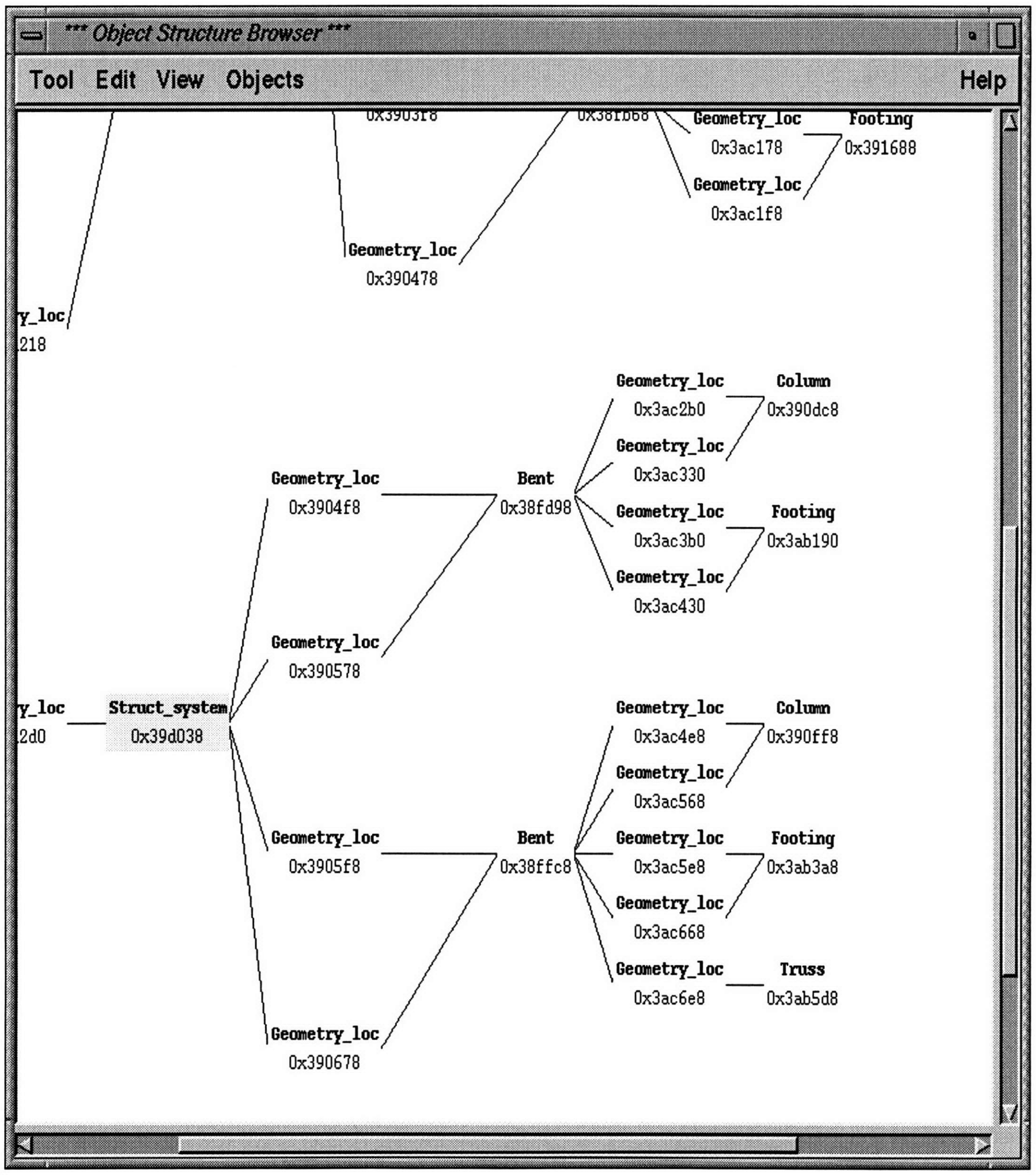
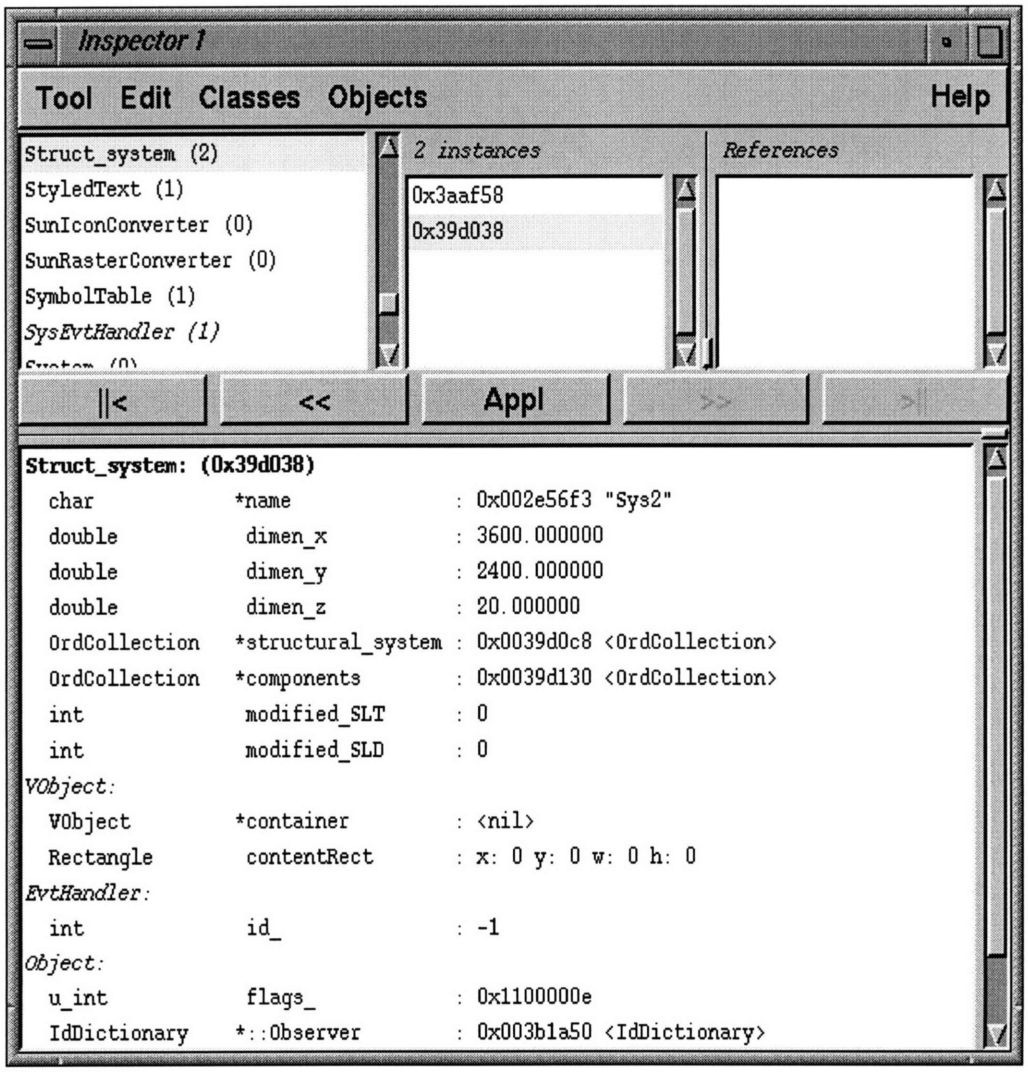


Figure 5.18: The structural system to be redesigned

Redesign

Through the Object Structure Browser, the user is allowed to access and modify the design object and, whenever necessary, to perform the redesign. Take Figure 5.18 as an example: If the designer would like to change the x-direction dimension of the structural system, he/ she can access the object through the inspector (Figure 5.19) and click on the attribute.

The system will prompt an instance browser window, see Figure 5.20. Such a change will propagate to the objects down the hierarchy. As we see in Figures 5.21 and 5.22, the bent contained in the structural system is also marked “modified”.



```
Struct_system (2)
StyledText (1)
SunIconConverter (0)
SunRasterConverter (0)
SymbolTable (1)
SysEvtHandler (1)
```

```
0x3aaf58
0x39d038
```

```
Struct_system: (0x39d038)
  char           *name               : 0x002e56f3 "Sys2"
  double          dimen_x            : 3600.000000
  double          dimen_y            : 2400.000000
  double          dimen_z            : 20.000000
  OrdCollection  *structural_system  : 0x0039d0c8 <OrdCollection>
  OrdCollection  *components         : 0x0039d130 <OrdCollection>
  int             modified_SLT       : 0
  int             modified_SLD       : 0
VObject:
  VObject        *container          : <nil>
  Rectangle       contentRect        : x: 0 y: 0 w: 0 h: 0
EvtHandler:
  int             id_                : -1
Object:
  u_int           flags_             : 0x1100000e
  IdDictionary   *::Observer         : 0x003b1a50 <IdDictionary>
```

Figure 5.19: Inspector for Structural System 2

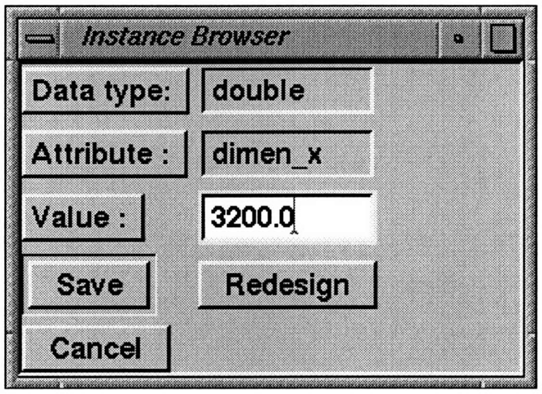
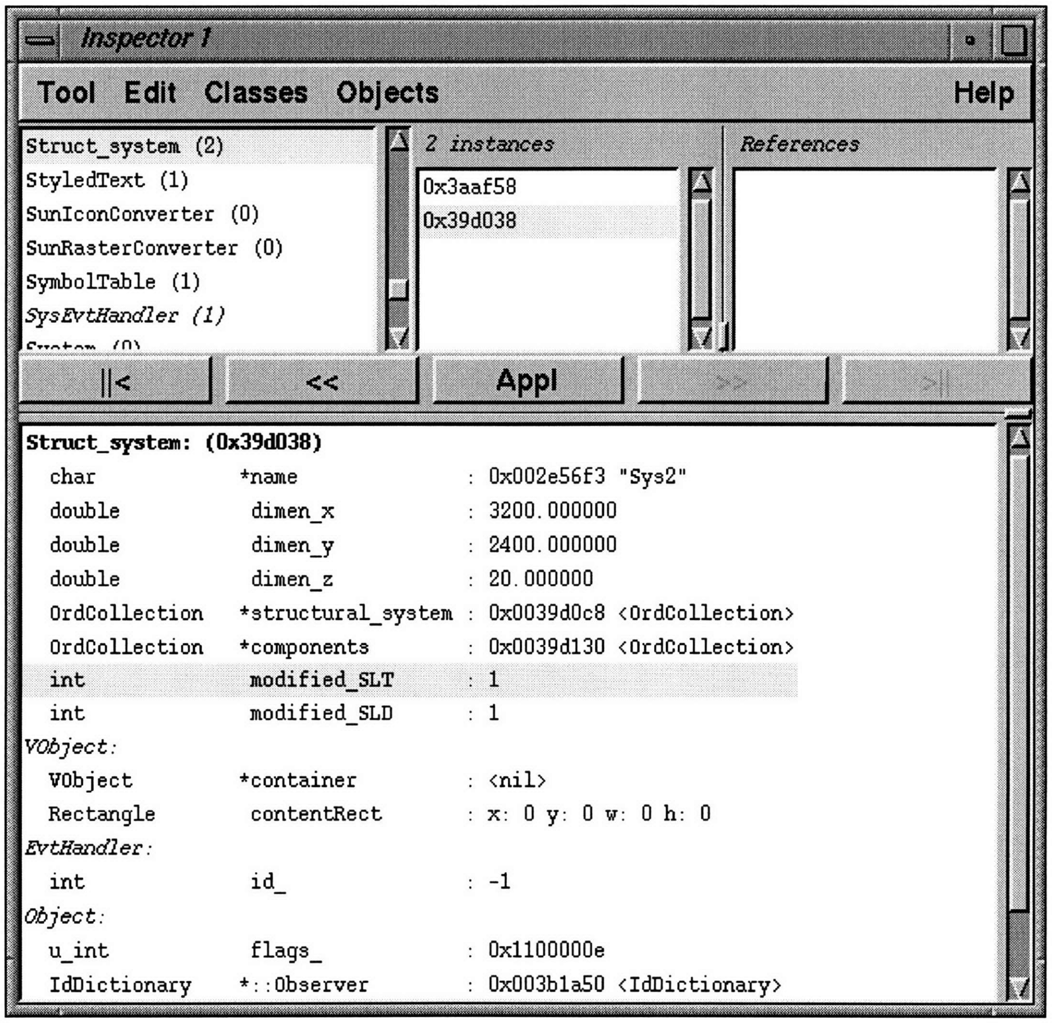


Figure 5.20: Change Slot Value

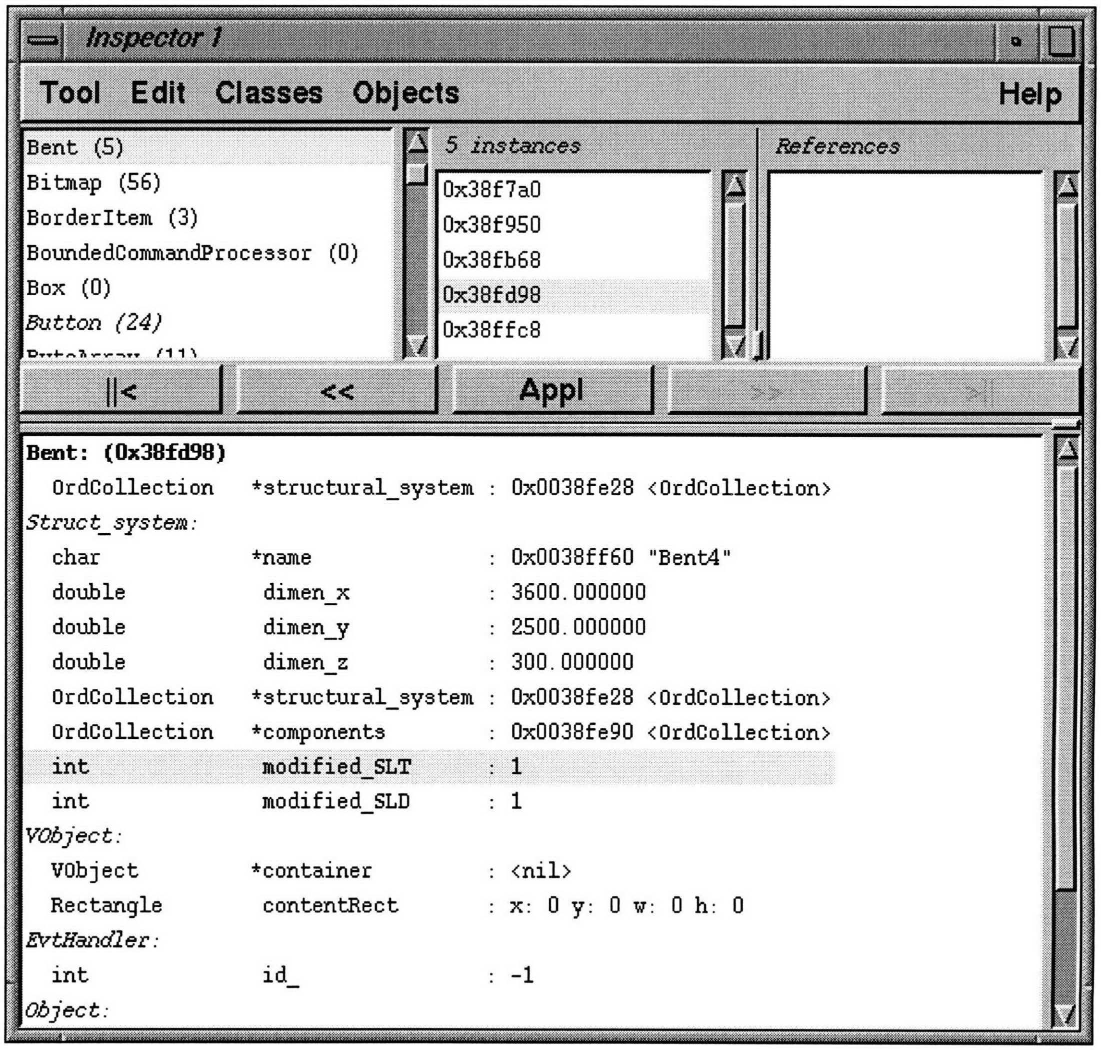


```
Struct_system (2)
StyledText (1)
SunIconConverter (0)
SunRasterConverter (0)
SymbolTable (1)
SysEvtHandler (1)
```

```
0x3aaf58
0x39d038
```

```
Struct_system: (0x39d038)
  char           *name                : 0x002e56f3 "Sys2"
  double          dimen_x             : 3200.000000
  double          dimen_y             : 2400.000000
  double          dimen_z             : 20.000000
  OrdCollection  *structural_system   : 0x0039d0c8 <OrdCollection>
  OrdCollection  *components          : 0x0039d130 <OrdCollection>
  int             modified_SLT        : 1
  int             modified_SLD        : 1
VObject:
  VObject        *container           : <nil>
  Rectangle       contentRect         : x: 0 y: 0 w: 0 h: 0
EvtHandler:
  int             id_                 : -1
Object:
  u_int           flags_              : 0x1100000e
  IdDictionary   *::Observer          : 0x003b1a50 <IdDictionary>
```

Figure 5.21: The structural system marked “modified” after the change.



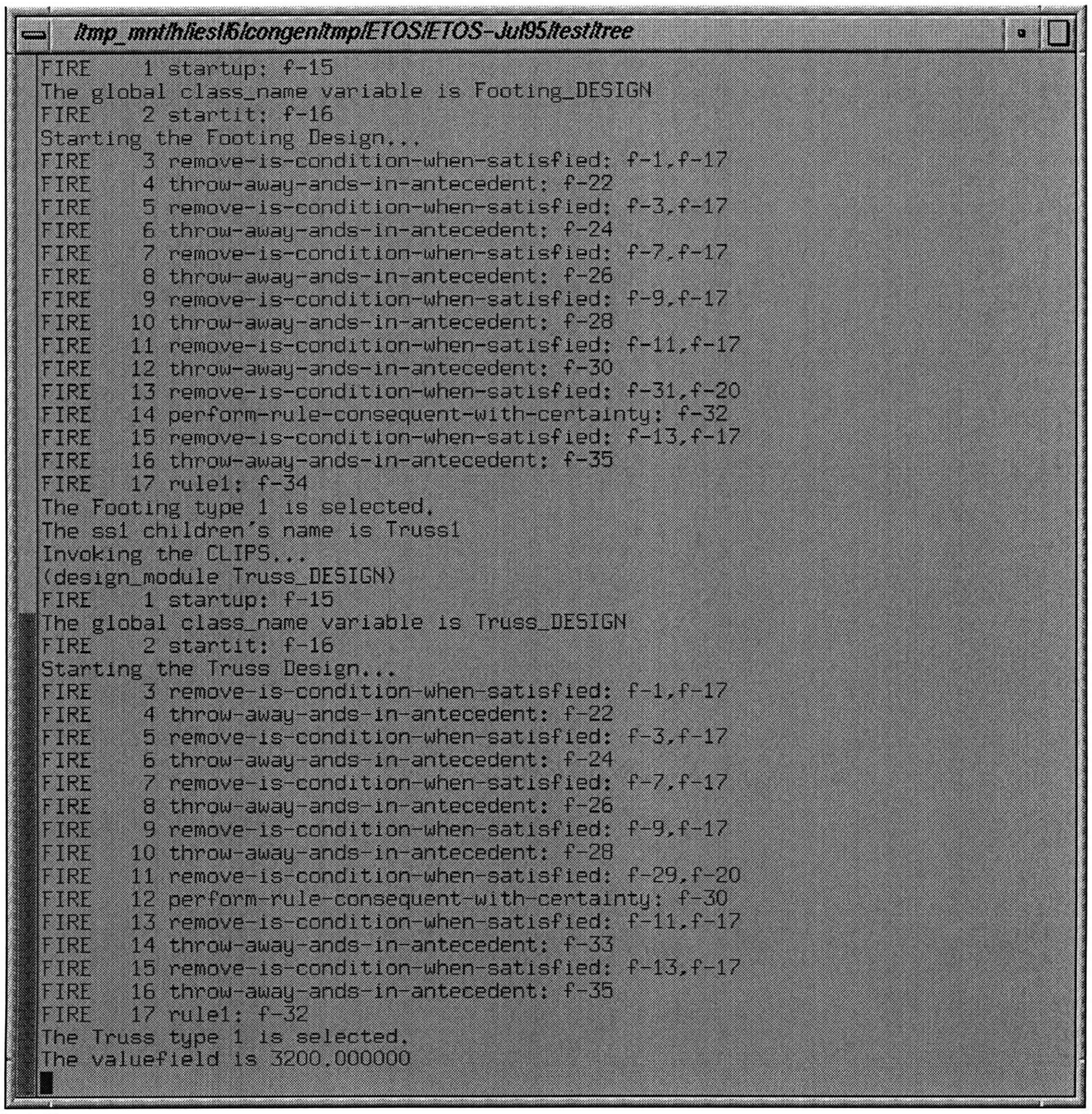
```
Bent (5)
Bitmap (56)
BorderItem (3)
BoundedCommandProcessor (0)
Box (0)
Button (24)
```

```
0x38f7a0
0x38f950
0x38fb68
0x38fd98
0x38ffc8
```

```
Bent: (0x38fd98)
  OrdCollection   *structural_system : 0x0038fe28 <OrdCollection>
Struct_system:
  char            *name              : 0x0038ff60 "Bent4"
  double           dimen_x           : 3600.000000
  double           dimen_y           : 2500.000000
  double           dimen_z           : 300.000000
  OrdCollection   *structural_system : 0x0038fe28 <OrdCollection>
  OrdCollection   *components        : 0x0038fe90 <OrdCollection>
  int              modified_SLT      : 1
  int              modified_SLD      : 1
VObject:
  VObject         *container         : <nil>
  Rectangle        contentRect       : x: 0 y: 0 w: 0 h: 0
EvtHandler:
  int              id_               : -1
Object:
```

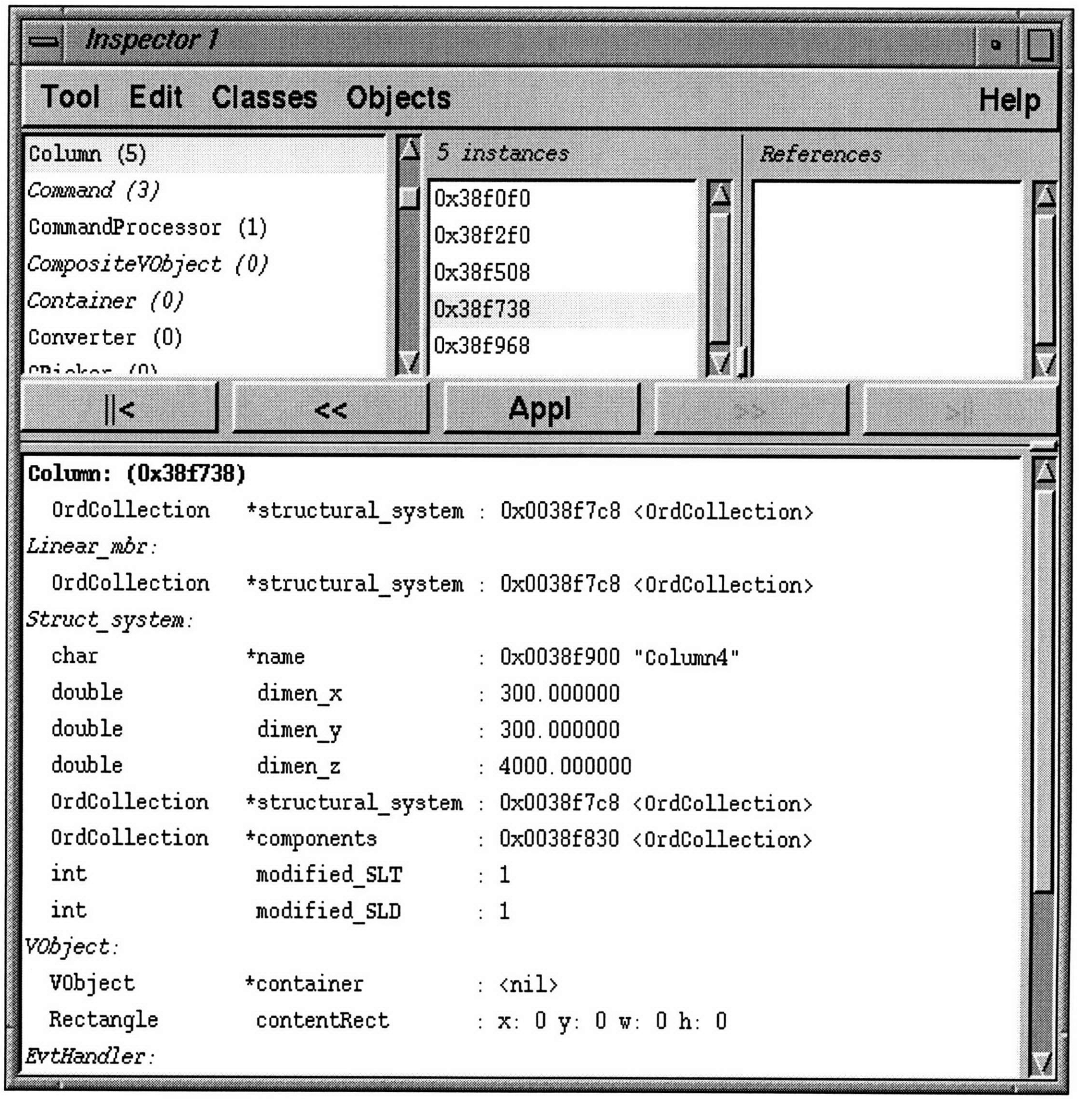
Figure 5.22: Bent4 is marked “modified”

The designer is free to modify or redesign any other components and can trigger the inference engine to perform the redesign from the highest level of the hierarchy. The structural agent will invoke the design rules associated with each object to check the validity of the modification and perform the redesign, if necessary, see Figures 5.23 through 5.25.



```
FIRE    1 startup: f-15
The global class_name variable is Footing_DESIGN
FIRE    2 startit: f-16
Starting the Footing Design...
FIRE    3 remove-is-condition-when-satisfied: f-1,f-17
FIRE    4 throw-away-ands-in-antecedent: f-22
FIRE    5 remove-is-condition-when-satisfied: f-3,f-17
FIRE    6 throw-away-ands-in-antecedent: f-24
FIRE    7 remove-is-condition-when-satisfied: f-7,f-17
FIRE    8 throw-away-ands-in-antecedent: f-26
FIRE    9 remove-is-condition-when-satisfied: f-9,f-17
FIRE   10 throw-away-ands-in-antecedent: f-28
FIRE   11 remove-is-condition-when-satisfied: f-11,f-17
FIRE   12 throw-away-ands-in-antecedent: f-30
FIRE   13 remove-is-condition-when-satisfied: f-31,f-20
FIRE   14 perform-rule-consequent-with-certainty: f-32
FIRE   15 remove-is-condition-when-satisfied: f-13,f-17
FIRE   16 throw-away-ands-in-antecedent: f-35
FIRE   17 rule1: f-34
The Footing type 1 is selected.
The ss1 children's name is Truss1
Invoking the CLIPS...
(design_module Truss_DESIGN)
FIRE    1 startup: f-15
The global class_name variable is Truss_DESIGN
FIRE    2 startit: f-16
Starting the Truss Design...
FIRE    3 remove-is-condition-when-satisfied: f-1,f-17
FIRE    4 throw-away-ands-in-antecedent: f-22
FIRE    5 remove-is-condition-when-satisfied: f-3,f-17
FIRE    6 throw-away-ands-in-antecedent: f-24
FIRE    7 remove-is-condition-when-satisfied: f-7,f-17
FIRE    8 throw-away-ands-in-antecedent: f-26
FIRE    9 remove-is-condition-when-satisfied: f-9,f-17
FIRE   10 throw-away-ands-in-antecedent: f-28
FIRE   11 remove-is-condition-when-satisfied: f-29,f-20
FIRE   12 perform-rule-consequent-with-certainty: f-30
FIRE   13 remove-is-condition-when-satisfied: f-11,f-17
FIRE   14 throw-away-ands-in-antecedent: f-33
FIRE   15 remove-is-condition-when-satisfied: f-13,f-17
FIRE   16 throw-away-ands-in-antecedent: f-35
FIRE   17 rule1: f-32
The Truss type 1 is selected.
The valuefield is 3200.000000
```

Figure 5.23: Text screen shows the rules fired.



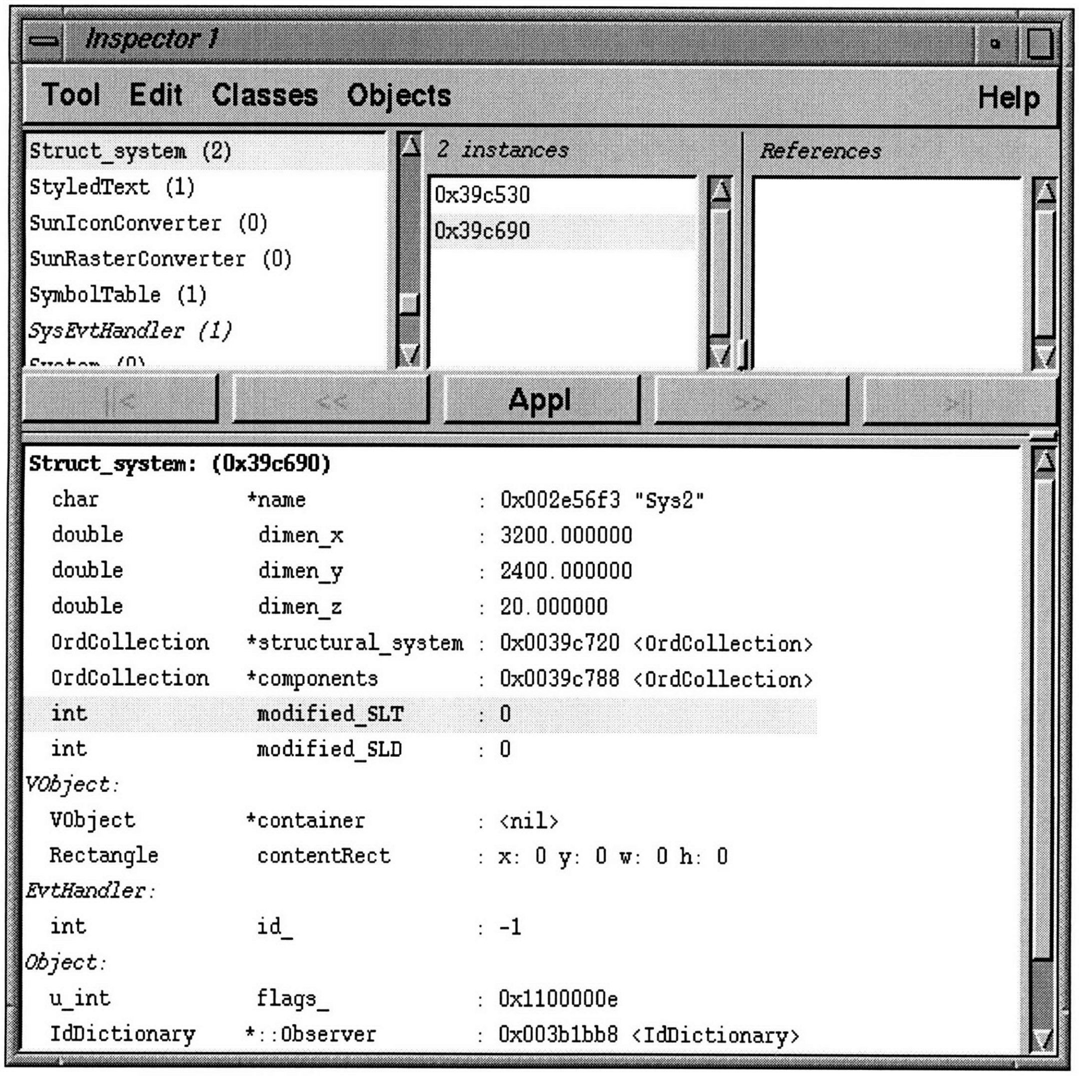
```
Column (5)
Command (3)
CommandProcessor (1)
CompositeVObject (0)
Container (0)
Converter (0)
```

```
5 instances
0x38f0f0
0x38f2f0
0x38f508
0x38f738
0x38f968
```

References

```
Column: (0x38f738)
  OrdCollection   *structural_system : 0x0038f7c8 <OrdCollection>
Linear_mbr:
  OrdCollection   *structural_system : 0x0038f7c8 <OrdCollection>
Struct_system:
  char            *name              : 0x0038f900 "Column4"
  double           dimen_x           : 300.000000
  double           dimen_y           : 300.000000
  double           dimen_z           : 4000.000000
  OrdCollection   *structural_system : 0x0038f7c8 <OrdCollection>
  OrdCollection   *components        : 0x0038f830 <OrdCollection>
  int              modified_SLT      : 1
  int              modified_SLD      : 1
VObject:
  VObject         *container         : <nil>
  Rectangle        contentRect       : x: 0 y: 0 w: 0 h: 0
EvtHandler:
```

Figure 5.24: Column4 is also marked modified



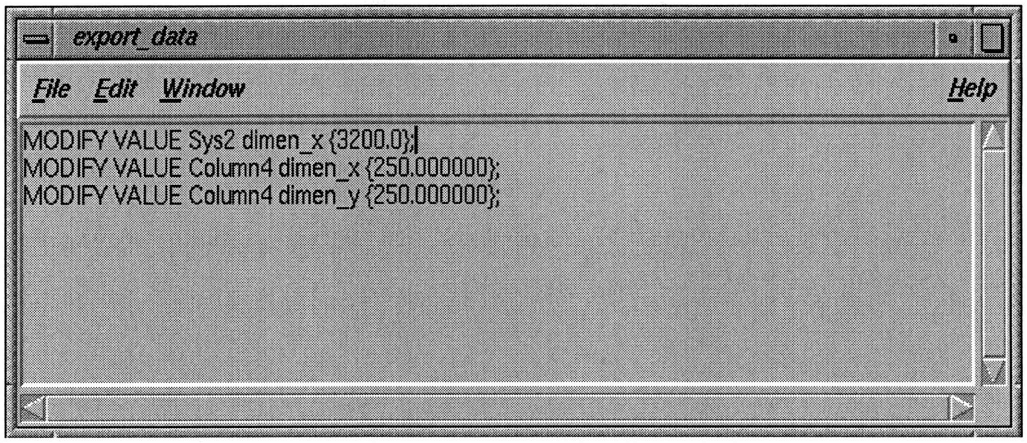
```
Struct_system (2)
StyledText (1)
SunIconConverter (0)
SunRasterConverter (0)
SymbolTable (1)
SysEvtHandler (1)

0x39c530
0x39c690

Struct_system: (0x39c690)
  char            *name                : 0x002e56f3 "Sys2"
  double           dimen_x             : 3200.000000
  double           dimen_y             : 2400.000000
  double           dimen_z             : 20.000000
  OrdCollection   *structural_system   : 0x0039c720 <OrdCollection>
  OrdCollection   *components          : 0x0039c788 <OrdCollection>
  int              modified_SLT        : 0
  int              modified_SLD        : 0
VObject:
  VObject         *container           : <nil>
  Rectangle        contentRect         : x: 0 y: 0 w: 0 h: 0
EvtHandler:
  int              id_                 : -1
Object:
  u_int            flags_              : 0x1100000e
  IdDictionary    *::Observer          : 0x003b1bb8 <IdDictionary>
```

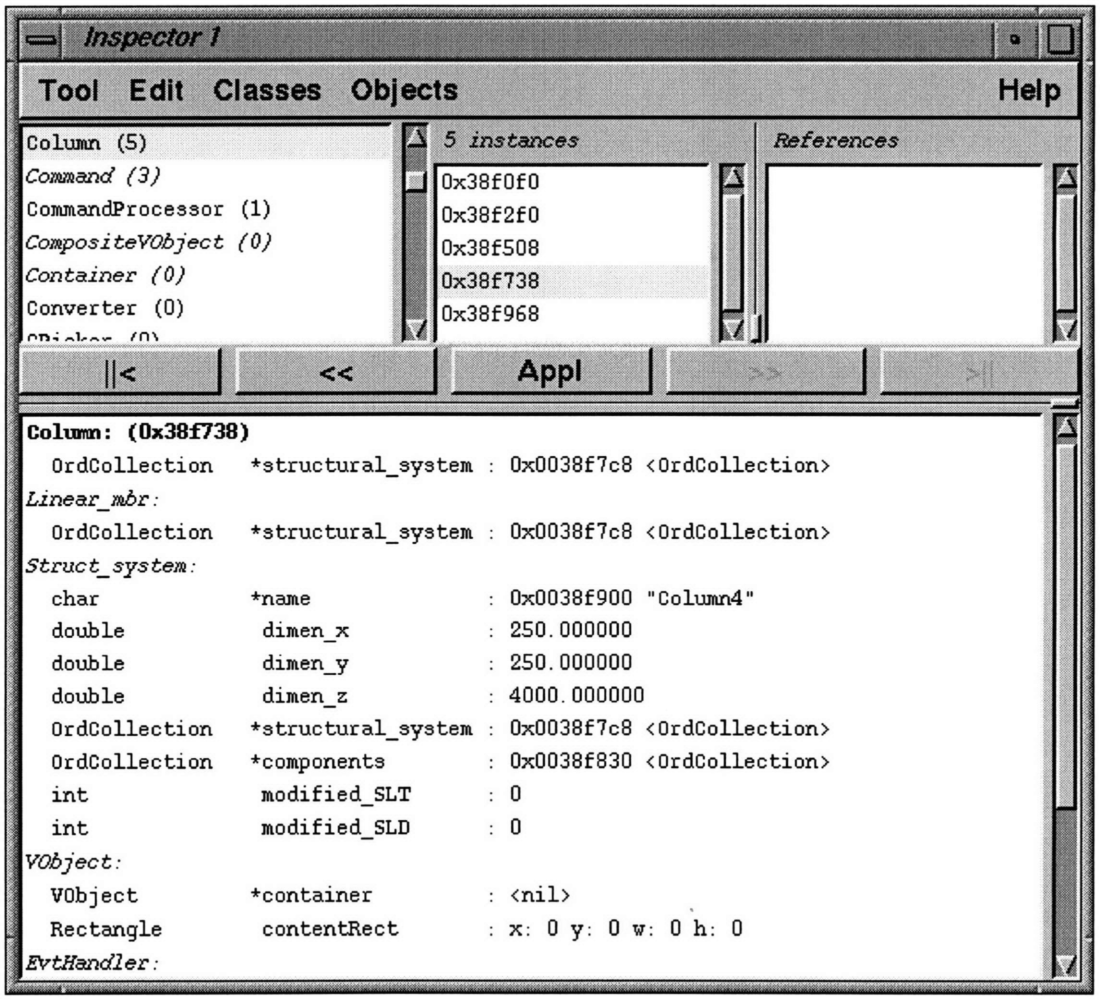
Figure 5.25: After redesign is done.

After the redesign is finished, the designer can transmit these changes to the facilitator. As we see in Figure 5.25, after the message is sent, the inspector will set the attributes modified_since_last_transmission and modified_since_last_design to 0. Figure 5.26 shows that the agent exports the modified data to the facilitator. Figure 5.27 shows that the column has been redesigned to a new size.



```
MODIFY VALUE Sys2 dimen_x {3200.0};
MODIFY VALUE Column4 dimen_x {250.000000};
MODIFY VALUE Column4 dimen_y {250.000000};
```

Figure 5.26: Export the modified data



```
Column: (0x38f738)
  OrdCollection   *structural_system : 0x0038f7c8 <OrdCollection>
Linear_mbr:
  OrdCollection   *structural_system : 0x0038f7c8 <OrdCollection>
Struct_system:
  char            *name              : 0x0038f900 "Column4"
  double           dimen_x           : 250.000000
  double           dimen_y           : 250.000000
  double           dimen_z           : 4000.000000
  OrdCollection   *structural_system : 0x0038f7c8 <OrdCollection>
  OrdCollection   *components        : 0x0038f830 <OrdCollection>
  int              modified_SLT      : 0
  int              modified_SLD      : 0
VObject:
  VObject         *container         : <nil>
  Rectangle        contentRect       : x: 0 y: 0 w: 0 h: 0
EvtHandler:
```

Figure 5.27: Column4 after redesign.

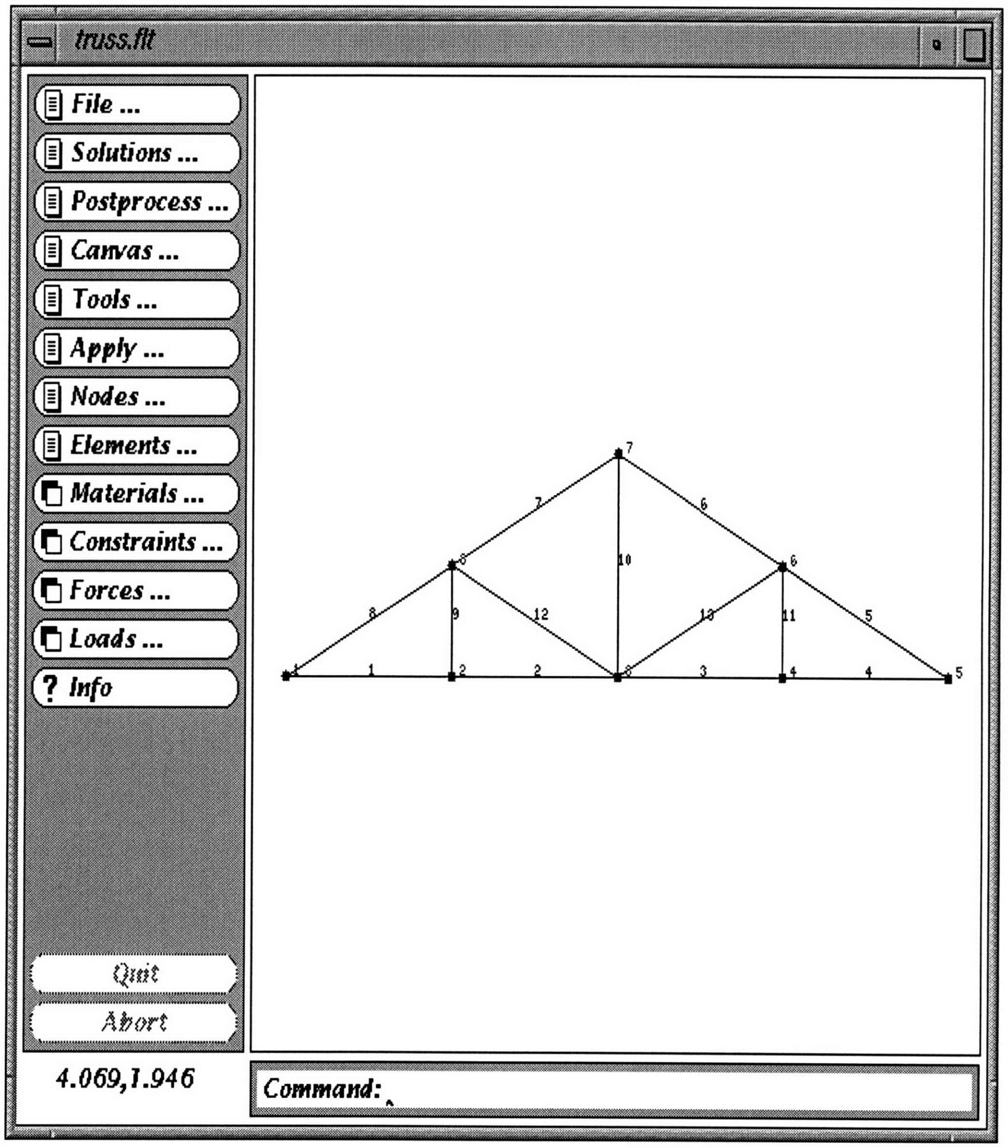


Figure 5.28: Detailed Design for Roof Truss

Detailed Design Phase

In this phase, a simple finite element program is utilized to analyze the structure. Figure 5.28 shows a simple truss under analysis. The designer can specify loading and bounrday conditions, assign materials, and change the system configuration to verify different design alternatives. The program computes stress under the user-specified constraints and displays the result graphically. The designer should manually and carefully check the result to verify that the design conforms to building code. If it does, the design process can come to a close. If not, the designer must either try another alternatives or return to the previous design stage and look for another alternative. With the aid of this numerical analysis program, the designer can achieve a much more precise design.

Chapter 6

Conclusion and Future Work

6.1 Current Status

This document describes the implementation of a structural design agent developed to be used in collaborative, multidisciplinary design in a federation architecture. The central idea is to understand the problems and processes associated with facilitating cooperation with other design agents during an example process to complete the cabin and firestation designs and to understand what is needed to streamline the whole process in a collaborative fashion.

Object-oriented methodology is employed as the principal modeling technique in the structural design agent. This agent performs design in a hierarchical fashion, from rough parameters and concepts to detailed components passed to facilitators. Object-oriented methodology proved to be a very successful solution to modeling complex relationships between design objects. The Structural Engineering Vocabulary is proposed as a general framework for modeling any kind of structural system. Enumeration and duplicative detail are neatly avoided, and data consistency is enhanced.

The structural agent is built to simulate the real-life structural design process which is essentially broken down into three phases in our implementation. Alternatives for the overall structural system are chosen in the conceptual phase. Preliminary sizing of major structural components are determined during the preliminary phase. Optimal sizing and location of structural members are given during the final detailed design phase. Negotiation and conflict resolution occur during and between phases. Iteration within and to earlier phases is possible.

Several software tools are integrated in the process of building the structural agents. Visualization of relationships between design objects and support for redesign are two main features for the user front-end. The integration of the object-oriented system with a rule-based expert system proves to provide an elegant solution to harnessing the advantages of both system paradigms. The expert system capabilities are embedded within individual objects by associated a rule base with any object class.

6.2 Experience with the Federation Architecture

This project was conceived as a test of concepts and facilities of a federation architecture, which is part of the Knowledge Sharing Initiative (KSI) project at Stanford University. The objectives of the KSI project are to develop technology and methodologies that will enable the sharing and reusing of knowledge bases. Many impediments must be overcome to achieve these goals. Two central problems addressed in this project are the following:

• **Heterogeneous Representation Languages:** There is no single knowledge representation that is best for all problems, nor is there likely to be one. The choice of one form of knowledge representation over another can have a big impact on a system's performance. Thus, in many cases, sharing and reusing knowledge will involve translating from one representation to another. Tools are needed that can help automate the translation process.

• **Heterogeneous Ontologies:** Even if the representation problems are resolved, it can still be difficult to combine two knowledge bases or establish effective communications between them. The absence of a shared vocabulary presents a further barrier, which could be removed through the development of shared sets of explicitly defined terminology, sometimes called ontologies. For such ontologies to be useful, the definitions provided must include declarative constraints that specify the semantics of the terms being defined, and the ontology must provide procedural methods that enforce those constraints when the terms are used in an application.

In particular, the objectives of the project are to develop technology and methodologies for building ontologies in a form that is translatable into the specialized representation languages of multiple application system environments, and interchanging the reusable content of knowledge bases, including ontologies, among specialized representation languages.

The interlingua representation language being developed in the Knowledge-Sharing Effort is called a Knowledge Interchange Format (KIF). KIF is intended to be a language for communication (" literary publication" of knowledge) and is designed to make the epistemological level content of a knowledge base clear to the reader, but not to support automated reasoning in that form.

The problems involved in interchanging knowledge bases are not yet well understood, and there is an open debate as to whether a generally useful interlingua can be specified. In the KSI project, they are attempting to inform that debate by developing tools for translating knowledge bases into and out of KIF, and using those tools to conduct knowledge interchange experiments that will substantially test the viability and adequacy of KIF as an interlingua.

In this framework, the facilitator acts primarily as a message switching device, translating and routing messages on the basis of interests expressed by individual agents in their own vocabulary. Messages handled by the facilitator can be basically categorized as follows:

- construct and destruct instances

- add, delete, and modify slot values
- add and delete relationships
- declare and forget interests

The agent has to establish a connection with the facilitator and register the communication process by declaring interests. Whenever the agent has certain information to broadcast, it just sends all the messages to the facilitator without having to specify the destination of messages. The facilitator takes the responsibility in routing those messages to the destination based on agents' interests. The vocabulary translation is also performed by the facilitator to transform one kind of data abstraction to another.

Our experience indicates that, as a collaborative design support system, the federation architecture still has some shortcomings which must be improved. There are two major requirements which the facilitator should satisfy for the structural design agent.

6.2.1 Translation between different schemas.

One basic assumption of the federation architecture is that the facilitator will perform the translations between different data structures. In the current implementation, the facilitator fails to meet this requirement. For example, a beam object in the structural engineering vocabulary maps to two objects in the architectural vocabulary, see Figure 6.1. One is a functional description object, the other a physical component. The facilitator should be able to combine the two objects into a format that the structural agent can understand and create a second object going the other way. Because the structural vocabulary is recursively defined, the facilitator must deal with complicated relationships between objects to perform the translation correctly. For example, data contained within a hierarchy of objects must be used to construct a transformation matrix to determine the geometric location of a basic object in the framework of another vocabulary. This creates the need for either memory in the translation process between messages, or queries for data needed in the translation. The memory needs to relate geometric location recursively to components, for example. According to our experience, to solve the problems of translation, the facilitator should either memorize the relationships between those messages it sends and receives, or provide the query capability between design agents. To be more precise, before it began translating, the facilitator must first comprehend the multiple hierarchical relationships between design objects in different schemas. Due to the recursive nature of structural engineering vocabulary, one-to-one and one-to-two mapping will not succeed in this context.

Let's take the col2 object in Figure 6.1 as an example. First of all, the facilitator must recognize that this objects appears four lines in the structure, twice each in loc1 and loc2 . For each copy, the facilitator would like to map the geometric location of this object to the corresponding object in the architectural vocabulary, it has to add the coordinates relative to the root object to obtain to absolute coordinates. Besides, the facilitator has to create the second object, either the functional description or physical component object, so that the Architectural Agent can understand and capture the information sent by the Structural Agent. In this context, the facilitator must realize the whole relationship hierarchy in both schemas so that the mapping can be done correctly. On the other hand, if the facilitator wants to transform the geometric location from Architectural Agent to its counterpart in Structural Agent, it has to combine both the physical and functional objects, which are transmitted separately, to a single object so that the Structural Agent can understand the information correctly. Therefore, the translation problem would become extremely difficult if the facilitator does not have memory or query capability.

6.2.2 Robust Communication Service.

The current implementation of communications API and facilitator had some fundamental shortcomings, aside from the translation issues, which made message-based collaboration between design agents very difficult. We found the following problems:

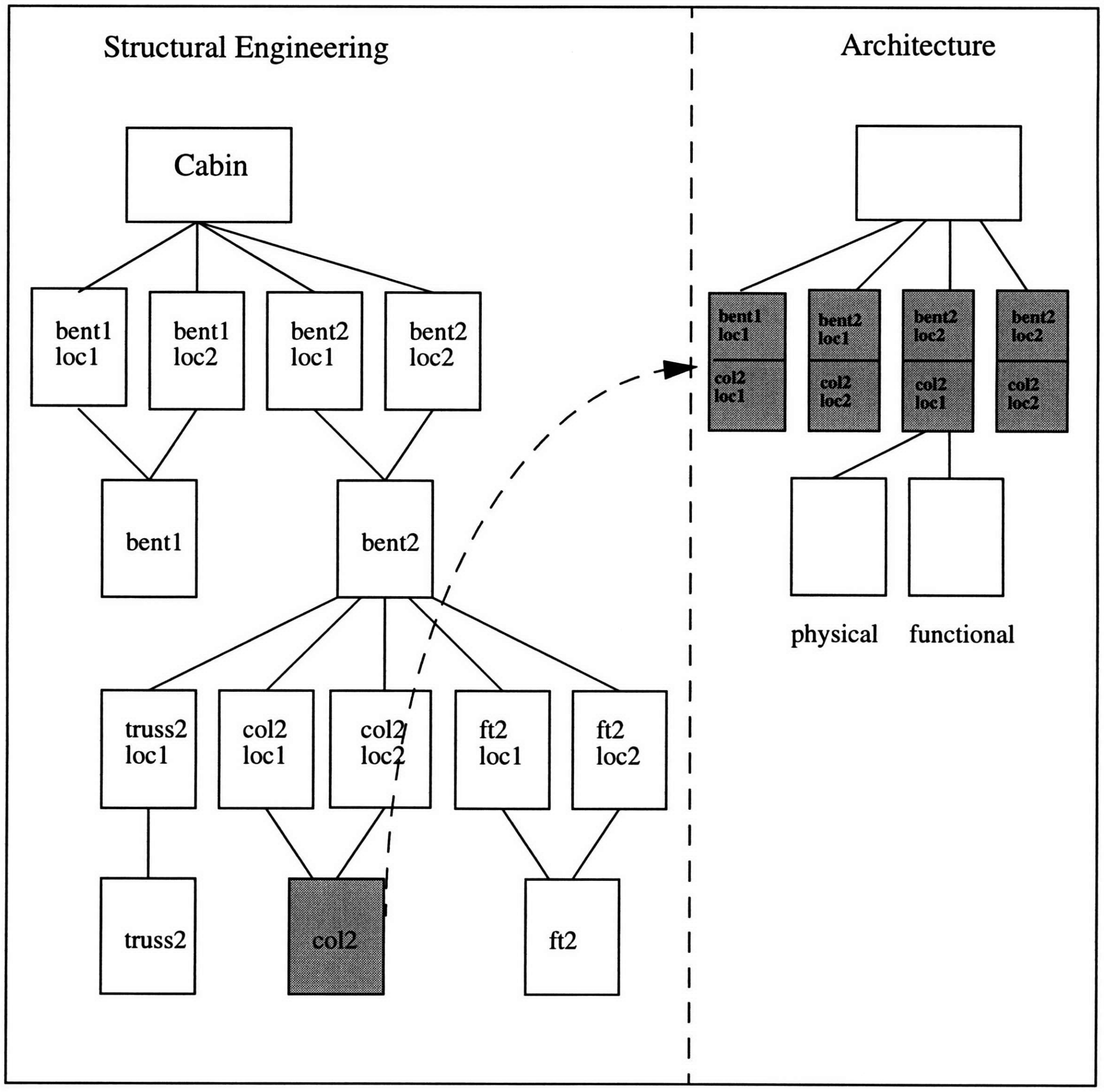


Figure 6.1: Vocabulary Translation between Architectural and Structural Agents

- **Message Order** - The facilitator periodically failed to transmit the messages in their original or logical sequence. Changing order can cause serious problems. For example, if the ADD_VALUE to an object message arrives earlier than the

CREATE_OBJECT message, the agent will find the former message invalid since it can not retrieve the object and then change the slot value. Consequently, keeping the messages sent and received in a logical order is essential in guaranteeing the success of federation architecture.

- **Dropping of Messages** - Sometimes, the facilitator dropped the messages it received. This is a serious fault, because incomplete information results in the agent's malfunction.
- **Case Sensitivity** - Due to the nature and implementation of KIF and LISP, the API was developed without the ability to distinguish between upper or lower case text. But in C++, the major agent development language, case sensitivity is enforced. This created fundamental incompatibilities . As a result, a set of conversion routines were needed and were implemented to transform all uppercase API messages into a case-sensitive message formats. Such an approach lacks generality.
- **Requires Hand Coding** - There is no message looping or high level language constructs provided by the API; therefore, extensive hand coding is required for the programmer to use the API. In our implementation, we take advantage of OML, which provides generic message handling, to interface with the API. We had to provide a set of translation functions to serve as a translator between these two utilities in order to avoid hand coding all possible messages related to our interests or results. The facilitator API provides only for message passing, while the OML was needed to take the responsibility for internal object manipulation. This combination turns out to be very useful and effective in communicating messages, as well as directly manipulating design objects. Hand coding for each object class was thus avoided.

But the API itself is inadequate.

These problems should be resolved to guarantee that the agent communicates correctly. The stability of the facilitator is also an important issue which cannot be ignored. Our experience reflected that, even for a small collaborative project like cabin design, the facilitator crashed frequently for unknown reasons.

6.3 Better API Support

The current implementation of API is straightforward, easy to understand and simple to use. According to our experience, however, it still requires some enhancements in order to support the design collaboration.

There are three primary goals for bettering the API support:

• Ease of integration with existing code.

• Ease of determining synchronization points.

• Ease of supporting queries.

The first requirement means that given existing source code, it should be easy to modify the program so that it can make use of the API. The second requirement means that it should be easy to ensure that messages are sent out when objects are changed locally, and remote changes are easily incorporated when they are received. The third requirement means that the facilitator needs be able to support query in order to complete the translation. Furthermore, for supporting negotiations between agents, querying is apparently shown to be an effective solution. If an application is going to participate in a federation architecture, the API should be able to provide some support for queries.

The applications with which we are integrating use a number of objects to represent its information. The API should not be required to access these object directly. Instead, the

API must be able to deal with object descriptions or embedded objects. An object description is a bridge between the API and the application being integrated. The object description is capable of performing the mapping between the information model of the API, and the information model of the application. CMU group's Object Modeling Language [20][21] provides an excellent example. The API should provide a compiler that takes C++ object declarations and produces code for the corresponding object description. The object description code is then compiled and linked with the application and the API binaries. The result is an executable where both the application and the API can manipulate the embedded object declarations. The API never directly modifies the application's "real" or "concrete" objects. Therefore, the API can be written in such a way that it need not know beforehand what an application's objects look like. This also allows the API to act as a knowledge store on behalf of the application. Since the object descriptions know what the concrete objects look like, they are able to answer questions about the object's data.

The object descriptions should support simple methods that correspond to KQML performatives, like "create", "delete", "modify", etc. The API defers these actions to the object description. The object description, having been created with knowledge of the concrete object, in turn carries out the appropriate action. In the case of "assign", for example, the object description is responsible for mapping between the slot name and the actual C++ slot that will be modified.

There are three ways in which the object description can be associated with their corresponding concrete objects. The object description can be wrapped around the concrete object. It can be a constituent of the concrete object. A third option is for the API to maintain a mapping between objects descriptions and concrete objects.

The API itself should be relatively simple. It would be responsible for sending messages to and from the network. It would parse the KQML message to determine the performative, and the object to which the performative should be applied. The API would find the object descriptions in a hash table, and invoke the method that corresponds to the performative. The object description itself would then modify the state of the concrete object in the appropriate way. High-level programming constructs should be provided to hide these low-level details from the programmer.

The handling of queries would work in much the same way. The API would never need to interface with the application in order to respond to a query. The API would look through its collection of object descriptions, and determine the response to the query by considering each object description. The response would be formulated and sent back to the facilitator without any input on behalf of the application itself.

Synchronization could be handled by the addition of access methods for the concrete objects. If objects are only modified through the use of these methods, then the object descriptions can be modified in the same place. This guarantees that a object description is always synchronized with its concrete object.

6.4 Recommended Enhancements to the Structural Agent

At this point the Structural Agent will perform design for simple structures using its hierarchical knowledge base, which embedding a realistically complex vocabulary, it contains limited knowledge and design process management tools. In terms of enhancing structural agent's capabilities, there are three goals to pursue.

6.4.1 Integration of Design Product and Process

The Structural Engineering Vocabulary proposed in this document does not include the design process information. The synthesis of design product and process representation is an important feature in terms of providing a good design tool [81]. Through the object-oriented modeling, the vocabulary should provide methods to record the decision-making process. This information can be very useful in the process of negotiating with other agents.

6.4.2 Versioning Capability.

Versioning capability is needed in order to maintain design alternatives during the design process. Versioning will enable the user to store gradual enhancements for an object. In addition, it will help the user trace who makes each change to an object. In current implementation, the object structure browser is not able to display versioning relationships. In addition, the successor to the OML [23] will support this functionality.

6.4.3 Better Visualization Support.

It would be good if we could visualize the 3D geometry relationships between design components in a viewer and allow the designer to manipulate the object interactively. Some programming tools like Open Inventor Graphics Library not only provide object-oriented modeling capabilities and form a well-structured hierarchy of visual objects, but also allow the user to manipulate the objects interactively.

References

[1] J. D. Biedermann and D. E. Grierson. "A Generic Model for Building Design." *Engineering with Computers,* pages 173-184, 1995

[2] T. Khedro, M. R. Genesereth, and P. M. Teicholz. "Agent-Based Framework for Integrated Facility Engineering." *Engineering with Computers*, pages 94-107, 1993.

[3] T. Khedro, M. R. Genesereth, and P. M. Teicholz. "FCDA: A Framework for Collaborative Distributed Multidisciplinary Design." *Proceedings of the Workshop on AI in Collaborative Design, 11th National Conference on AI,* July, 1993.

[4] P. W. H. Chung and R. Goodwin. "Representing Design History", *Artificial Intelligence in Design '94, 1994.*

[5] J. A. Abdalla and G. H. Powell. "Version Management Needs for Structural Engineering Design." *Engineering with Computers,* 1991

[6] K. R. Dittrich. "Object-Oriented Data Model Concepts." *Advances in Object-oriented Database System,* 1994.

[7] D. H. Lee and D. R. Decker. "Collaborative Engineering-Design Support System." *Artificial Intelligence in Design,* 1994.

[8] J. Giarratano and G. Riley. *Expert Systems - Principles and Programming,* Addison-Wesley, Reading, Mass., 1989.

[9] T. Lewis. *Object-Oriented Application Frameworks,* Manning Publications Company, 1995.

[10] B. Stroustrup. *The C++ Programming Language,* Addison-Wesley, Reading, Mass., 1992.

[11] M. A. Ellis and B. Stroustrup. *The Annonated C++ Reference Manual,* Addison-Wesley, Reading, Mass., 1990.

[12] J. A. Abdalla and G. H. Powell. "An Object Design Framework for Structural Engineering." *Engineering with Computers,* pages 1995.

[13] T. Khedro, M. Case, U. Flemming, M. R. Genesereth, R. Logcher, C. Pedersen, J. Snyder, D. Sriram, and P. Teicholz "Development of multi-institutional testbed for collaborative facility engineering infrastructure", *Computing in Civil Engineering, Proceedings of second Congress.* ASCE, pages 1308-1315, 1995.

[14] S. J. Fenves, U. Flemming, C. Hendrickson, M. Lou Maher, R. Quadrel, M. Trek, and R. Woodbury. *Concurrent Computer-Integrated Building Design,* Prentice Hall, 1994.

[15] R. Sause and G. H. Powell, "A Design Process Model for Computer Integrated Structural Engineering." *Engineering with Computer,* pp.129-143, 1990.

[16] J. Kim and C. William Ibbs. "Comparing Object-oriented and Relational Data Models for Project Control." *Journal of Computing in Civil Engineering,* ASCE, pages 348-369, 1992.

[17] D. Sriram, R. Logcher, "MIT DICE Project." *IEEE Computer,* pages 64-65, 1993.

[18] D. Sriram, R. Logcher, S. Fukuda *Computer-aided cooperative product development* : MIT-JSME workshop, MIT, Cambridge, USA, November 20/21, 1989

[19] D. Sriram. *Knowledge-based approaches for structural design.* Southampton ; Boston : Computational Mechanics, 1987.

[20] J. Snyder and U. Flemming. *ACL Object Modeling Language Specifications.* 1995

[21] J. Snyder and U. Flemming, *ACL Object Modeling Language Language Binding Reference,* 1995.

[22] U. Flemming and R. Woodbury. "SEED Overview." *Journal of Architectural Engineering,* ASCE, 1995.

[23] J. Snyder and U. Flemming, "SPROUT: A Modeling Language for SEED." *Journal of Architectural Engineering,* ASCE, pages 195-203, 1995.

[24] S. J. Fenves, "Conceptual Structural Design for SEED." *Journal of Architectural Engineering*, ASCE, pages 179-186, 1995.

[25] S. C-Y. Lu. *SWIFT Concurrent Engineering Technology for the Facility Delivery Processes,* 1994.

[26] G. Booch. *Object-Oriented Analysis and Design,* Benjamin/Cummings, 1994.

[27] J. Rambaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Analysis and Modeling,* Prentice Hall, 1991.

[28] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. "The ObjectStore Database System." *Communications of the ACM, pp. 50-63, October, 1991.*

[29] R. Sause and G. H. Powell. "A Design Process Model for Computer Integrated Structural Engineering" *Engineering with Computers,* pages 129-143, 1990

[30] E. A. Edmonds, L. Candy, R. Jones, and B. Soufi. " Support for Collaborative Design : Agent and Emergence." *Communications of the ACM,* pages 41-47, July, 1994.

[31] J. Favela, A. Wong, and A Chakravarthy. "Supporting Collaborative Engineering Design." *Engineering with Computer,* pages 125-132, 1993.

[32] D. R. Rigopoulos and I. J. Oppenheim. "Intelligent Objects for Synthesis of Structural Systems." *Journal of Computing in Civil Engineering,* ASCE, pages 266-281, 1992.

[33] J.W. Baugh and D. R. Rehak. "Data Abstraction in Engineering Software Development." *Journal of Computing in Civil Engineering,* ASCE, pages 282-301, 1992.

[34] C. S. Krishnamoorthy, H. Shivakumar, S. Rajeev, and S. Suresh. "A Knowledge-based System with Generic Tools for Structural Engineering." *Structural Engineering Review,* pages 121-131, 1993.

[35] J. A. Abdalla and G. H. Powell. "Version Management Needs for Structural Engineering Design." *Engineering with Computers,* pages 131-143, 1991.

[36] A. Kemper and G. Moerkotte. *Object-oriented Database Management - Applications in Engieering and Computer Science,* Prentice Hall, 1994.

[37] C. L. Dym and R. E. Levitt. *Knowledge-based Systems in Engineering,* McGraw-Hill, 1991.

[38] ObjectStore. *Using the ObjectStore C++ API,* ObjectStore Design. Inc., 1995.

[39] C. J. Date. *An Introduction to Database Systems,* Addison-Wesley, 1995.

[40] H. Adeli. *Expert Systems in Construction and Structural Engineering,* Chapman and Hall, 1988.

[41] M. D. Carroll and M. A. Ellis. *Designing and Coding Reusable C++,* Addison-Wesley, 1995.

[42] D. Collins, *Designing Object-Oriented User Interface,* Benjamin/Cummings, 1995.

[43] R. A. Coleman, *Strutural Systems Design,* Prentice-Hall Inc., 1980.

[44] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk, and N. Ballou. "Data Model Issues for Object-oriented Applications." *ACM Transactions on Information System,* 1992.

[45] R. H. Katz. "Toward a Unified Framework for Version Modeling in Engineering Database." *ACM Computing Surveys,* pages 375-408, December, 1990.

[46] H. Chou and W. Kim. "A Unifying Framework for Version Control in a CAD Environment." *Proceedings of the 12th VLDB Conference,* pages 336-346, August 1986.

[47] D. Sriram, S. Ahmed, and R. Logcher. "A Transaction Management Framework for Collaborative Engineering." *Engineering with Computers,* pages 213-232, 1992.

[48] M. Case, *The Discourse Model for Collaborative Engineering Design: A Distributed and Asynchronous Approach,* Ph.D. Thesis, University of Illinois, 1994.

[49] K. R. Dittrich and R. A. Lorie. "Version Support for Engineering Database Systems." *IEEE Transactions on Software Engineering,* pages 429-437, 1988.

[50] International Conference of Building Officials. *Uniform Building Code,* Vol.2, Structural Engineering and Design Provisions, 1994.

[51] American Institute of Steel Construction. *Manual of Steel Construction - Load and REsistance Factor Design,* 1995.

[52] T .Y. Lin and S. D. Stotesbury. *Structural Concepts and Systems for Architects and Engineers,* Van Nostrand Reinhold, 1988.

[53] Wolfgang Schueller. *The Vertical Building Structure,* Van Nostrand Reinhold, 1990.

[54] B. S. Smith and A. Coull. *Tall Building Structures - Analysis and Design,* John Wiley and Sons, Inc., 1991.

[55] P. Jackson. *Introduction to Expert Systems,* Addison-Wesley, 1990.

[56] E. Bertino and L. Martino. *Object-oriented Database Systems - Concepts and Architecture,* Addison-Wesley, 1993.

[57] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-oriented Database System - The Story of O_2*, Morgan Kaufmann Publishers, 1992.

[58] B. S. Taranath. *Structural Analysis and Design of Tall Buildings,* McGraw-Hill Inc., 1988.

[59] W. Zhong, C. Qiu, X Liu, X. Qin, and J. Liu. "A Knowledge-Based System for Concrete Building Structural Design." *Microcomputers in Civil Engineering,* pages 271-281, 1993.

[60] G. P. Luth, D. Jain, H. Krawinkler, and K. H. Law. "A Formal Approach to Automating Conceptual Structural Design, Part I: Methodology." *Engineering with Computers,* pages 79-89, 1991.

[61] M. Lou Maher. *Case-Based Reasoning in Design,* MIT course 4.273 Lecture Notes, 1994.

[62] C-K. Soh and A-K. Soh. "Example of Intelligent Structural Design System." *Journal of Computing in Civil Engineering,* ASCE, pages 329-345, 1988.

[63] D. V. Morse and C. Hendrickson. "Model for Communication in Automated Interactive Engineering Design." *Jounrnal of Computing in Civil Engineering,* ASCE, pages 4-24, 1991.

[64] J. Ganguly, D. Sriram, and E. Kausel. "Integrated Framework for Deriving Behavior from Structural Descriptions." *Journal of Computing in Civil Engineering,* ASCE, pages 391-411, 1991.

[65] C. K. Krishnamoorthy, C. S. Rao, and S. Rajeev. "A Design Synthesizer for KBES." *Structural Engineering Review,* pages 107-119, 1993.

[66] B. Curtis, M. I. Kellner, and J. Over. "Process Modeling." *Communications of the ACM,* pages 75-90, September, 1992.

[67] R. Sause and G. H. Powell. "A Design Process Model for Computer Integrated Structural Engineering - Design Phases and Tasks." *Engineering with Computers,* pages 145-160, 1991..

[68] D. J. Fraser. *Conceptual Design and Preliminary Analysis of Structures,* 1981.

[69] C. C. Pouangare. *Strategies for the Conceptual Design of Structures,* M.S. Thesis, Department of Civil and Environmental Engieering, MIT, 1987.

[70] F. S. Chehayeb. *A framework for engineering knowledge representation and problem solving,* Ph.D. Dissertation, Department of Civil and Environmental Engineering, MIT, 1987.

[71] J. M. Becker. *A Structural Design Process: Philosophy and Methodology,*

[72] H. J. Cowen and F. Wilson. *Structural Systems,* 1981.

[73] H. Parker and J. Ambrose. *Simplified Engineering for Architects and Builders, 1993.*

[74] T. Yagiu. *Modeling Desing Objects and Processes,* 1991.

[75] T. L. De Fazio.*Concurrent Design of Products and Processes*, 1989.

[76] American Society of Civil Engineers, *Expert System for Civil Engineers - Knowledge Representation*, ASCE, 1992.

[77] J. Ambrose. *Building Structures,* John Wiley and Sons, Inc., 1993.

[78] American Institute of Architects, *Professional Handbook,* AIA, 1993.

[79] J. Ousterout, *Tcl/Tk Toolkit,* Addison-Wesley, 1994.

[80] A. Weinand, E. Gamma, R. Marty "ET++ - An Object-Oriented Application Framework in C++." *Object-Oriented Programing Systems, Languages, and Application Conference Proceedings,* ACM Press, pages 46-57, 1988.

[81] S. R. Gorti, *From Form to Symbol : A Framework for Design Evolution,* Ph.D. dissertation, Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, 1994.

Appendix A

Structural Design Rules in CLIPS

Most structural rules of thumb are expressed as ratios, such as a truss that has a span to depth ratio of 8. As long as depths and spans are in terms of feet, the relationship is easy to perceive. In the truss example, for every 8 inch of span, the truss will be about 1 inch deep. Most of these rules are compiled and collected from design code, textbook, and experience of structural engineers.

```
;;;=========================================================
;;;    MIT Agent Structural Design Knowledge Base in CLIPS
;;;=========================================================
(defrule startup
(declare (salience 5000))
=>
(printout t "Starting the rule-based structural design expert system" crlf))
```

System Routines

```
(defmodule MAIN (export ?ALL))

(deffunction MAIN::ask-question (?question ?allowed-values)
  (printout t ?question)
  (bind ?answer (read))

  (if (lexemep ?answer) then (bind ?answer (lowcase ?answer)))
  (while (not (member ?answer ?allowed-values)) do
     (printout t ?question)
     (bind ?answer (read))
     (if (lexemep ?answer) then (bind ?answer (lowcase ?answer)))) ?answer
)
```

```
(deftemplate MAIN::attribute
  (slot name)
  (slot value)
  (slot certainty (default 100.0)))

(defrule MAIN::startup
  (declare (salience 1000))
  (design_module ?class_name)
=>
  (set-fact-duplication TRUE)
  (printout t "The global class_name variable is " ?class_name crlf)
  (assert (design_done no))
  (focus ?class_name )
)

(defrule MAIN::combine-certainties ""
 (declare (salience 100))
;;;       (auto-focus TRUE))
?rem1 <- (attribute (name ?rel) (value ?val) (certainty ?per1))
?rem2 <- (attribute (name ?rel) (value ?val) (certainty ?per2))
(test (neq ?rem1 ?rem2))
=>
(retract ?rem1)
(modify ?rem2 (certainty (/ (- (* 100 (+ ?per1 ?per2)) (* ?per1 ?per2))
100))))

(defrule MAIN::determine_struct_system_frame
(*struct_system* ~frame)
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel structural_system)
        (eq ?val frame)
        (> ?cer 50.0)))
=>
(printout t "The Frame system is selected" crlf)
(retract ?ss)
(assert (*struct_system* frame))
)

(defrule MAIN::determine_struct_system_wall
(*struct_system* ~wall)
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel structural_system)
       (eq ?val bearing_wall)
       (> ?cer 50.0)))
=>
     (printout t "The bearing_wall system is selected" crlf)
     (retract ?ss)
     (assert (*struct_system*  bearing_wall))
     )
```

```
(defmodule QUESTIONS (import MAIN ?ALL)(export ?ALL))

(deftemplate QUESTIONS::question
  (slot attribute (default ?NONE))
  (slot the-question (default ?NONE))
  (multislot valid-answers (default ?NONE))
  (slot already-asked (default FALSE))
  (multislot precursors (default ?DERIVE)))

(defrule QUESTIONS::ask-a-question
  ?f  <- (question (already-asked FALSE)
        (precursors)
        (the-question ?the-question)
        (attribute ?the-attribute)
        (valid-answers $?valid-answers))
=>
(modify ?f (already-asked TRUE))
(assert (attribute (name ?the-attribute)
    (value (ask-question ?the-question ?valid-answers)))))

(defrule QUESTIONS::precursor-is-satisfied
   ?f <- (question (already-asked FALSE)
       (precursors ?name is ?value $?rest))
       (attribute (name ?name) (value ?value))
=>
(if (eq (nth 1 ?rest) and)
then (modify ?f (precursors (rest$ ?rest)))
else (modify ?f (precursors ?rest))))

(defrule QUESTIONS::precursor-is-not-satisfied
  ?f <- (question (already-asked FALSE)
     (precursors ?name is-not ?value $?rest))
       (attribute (name ?name) (value ~?value))
=>
(if (eq (nth 1 ?rest) and )
then (modify ?f (precursors (rest$ ?rest)))
else (modify ?f (precursors ?rest))))
```

```
(defmodule RULES (import MAIN ?ALL)(export ?ALL))
(deftemplate RULES::rule
  (slot certainty  (default 100.0))
  (multislot if)
  (multislot then))

(defrule RULES::throw-away-ands-in-antecedent
  ?f <- (rule (if and $?rest))
  =>
(modify ?f (if ?rest)))
(defrule RULES::throw-away-ands-in-consequent
  ?f <- (rule (then and $?rest))
  =>
  (modify ?f (then ?rest)))

(defrule RULES::remove-is-condition-when-satisfied
  ?f <- (rule (certainty ?c1)
         (if ?attribute is ?value $?rest))
  (attribute (name ?attribute)
        (value ?value)
        (certainty ?c2))
  =>
  (modify ?f (certainty (min ?c1 ?c2)) (if ?rest)))

(defrule RULES::remove-is-not-condition-when-satisfied
  ?f <- (rule (certainty ?c1)
         (if ?attribute is-not ?value $?rest))
```

```
 (attribute (name ?attribute) (value ~?value) (certainty ?c2))
 =>
 (modify ?f (certainty (min ?c1 ?c2)) (if ?rest)))

(defrule RULES::perform-rule-consequent-with-certainty
 ?f <- (rule (certainty ?c1)
        (if)
        (then ?attribute is ?value with certainty ?c2 $?rest))
 =>
 (modify ?f (then ?rest))
 (assert (attribute (name ?attribute)
        (value ?value)
        (certainty (/ (* ?c1 ?c2) 100)))))

(defrule RULES::perform-rule-consequent-without-certainty
 ?f <- (rule (certainty ?c1)
        (if)
        (then ?attribute is ?value $?rest))
 (test (or (eq (length$ ?rest) 0)
       (neq (nth 1 ?rest) with)))
 =>
 (modify ?f (then ?rest))
 (assert (attribute (name ?attribute) (value ?value) (certainty ?c1))))
```

Design Rules for Conceptual Structural Systems

```
(defmodule Struct_system_DESIGN (import RULES ?ALL))
(deffacts the_system_design_rules
;;; the design rules

(rule (if rc_availability is yes and material is steel)
    (then structural_system is frame with certainty 80.0))

(rule (if rc_availability is no and material is wood)
    (then structural_system is bearing_wall with certainty 80.0 ))

(rule (if rc_availability is yes and material is rc)
    (then structural_system is bearing_wall with certainty 60.0
        structural_system is frame with certainty 60.0))

(rule (if soil is sand)
    (then footing is mat_footing with certainty 50.0)
)
(rule (if soil is clay)
    (then footing is combined_footing with certainty 50.0))

(rule (if soil is tight-sand)
    (then footing is spread_footing with certainty 60.0)))
(rule (if structural_system is frame and material is steel)
    (then economic_span_depth_ratio is 20 with certainty 80.0)
    (then spacing_height_range is 16 to 40 with certainty 80.0))

(rule (if structural_system is frame and material is concrete)
    (then economic_span_depth_ratio is 20 to 25 with certainty 80.0)
    (then spacing_height_range is 16 to 40 with certainty 80.0))

(rule (if structural_system is frame and material is pc)
    (then economic_span_depth_ratio is 25 to 30 with certainty 80.0)
    (then spacing_height_range is 16 to 40 with certainty 80.0))

(rule (if structural_system is frame and material is timber)
    (then economic_span_depth_ratio is 15 to 20 with certainty 80.0)
    (then spacing_height_range is 16 to 40 with certainty 80.0))

(rule (if structural_system is frame and material is laminated_wood)
    (then economic_span_depth_ratio is 15 to 20 with certainty 80.0)
    (then spacing_height_range is 16 to 40 with certainty 80.0))

(defrule Struct_system_DESIGN::startit
?done <- (design_done no)
    =>
    (retract ?done)
    (printout t "Starting the Struct_system Design..." crlf)
    (load-facts "Struct_system_Design_Knowledge_Base")
    (focus RULES)
    (assert (design_done yes))
    )
```

```
(defrule Struct_system_DESIGN::rule1
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel structural_system)
      (eq ?val frame)
      (> ?cer 50.0)))
=>
    (printout t "The frame structure system is selected. " crlf)
    (dbmake_instance Struct_system)
    )
(defrule Struct_system_DESIGN::rule2
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel structural_system)
      (eq ?val bearing_wall)
      (> ?cer 50.0)))
=>
(printout t "The bearing wall system is selected. " crlf)
(dbmake_instance Struct_system))
```

Design Rules for Column

```
(defmodule Column_DESIGN (import RULES ?ALL))
(defrule Column_DESIGN::startit
?done <- (design_done no)
=>
    (retract ?done)
    (printout t "Starting the Column Design..." crlf)
    (load-facts "Column_Design_Knowledge_Base")
    (focus RULES)
    (assert (design_done yes)))

(defrule Column_DESIGN::rule1
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_column)
      (eq ?val col_type_1)
      (> ?cer 50.0)))
=>
    (printout t "The column type 1 is selected. " crlf)
    (dbmake_instance Column)
    )

(defrule Column_DESIGN::rule2
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_column)
      (eq ?val col_type_2)
      (> ?cer 50.0)))
=>
(printout t "The column type 2 is selected. " crlf)
(dbmake_instance Column)
)
(deffacts Column_DESIGN::the_column_design_rules
;;; the design rules

(rule (material is rc and type is round)
    (then minimum_size is 12 with certainty 80.0))

(rule (if material is rc and type is round)
    (then minimum_size is 6 with certainty 80.0))

(rule (if material is rc and type is rect)
    (then minimum_cross_section_area is 120 with certainty 80.0)
    (then minimum_thickness is 8 with certainty 80.0))

(rule (if material is steel type is long)
    (then max_unbraced_height is 20.0 * cross_section with certainty 80.0)

(rule (if material is reinforced_masonry)
    (then min_dimension is 12.0 with certainty 80.0)
    (then max_height is 20 * min_dimension with certainty 80.0))

(rule (if material is plain_masonry)
    (then min_dimension is 12.0 with certainty 80.0)
    (then max_height is 10 * min_dimension with certainty 80.0))

(rule (if material is solid_wood)
    (then min_dimension is 16.0 with certainty 80.0)
    (then max_height is 30 * min_dimension with certainty 80.0))

(rule (if material is composite_wood)
    (then min_dimension is 20.0 with certainty 80.0)
    (then max_height is 60 * min_dimension with certainty 80.0))

(rule (if structural_system is column)
    (then unbraced_length is from 10 to 30 with certainty 80.0))

(rule (if structural_system is column and material is steel)
    (then economic_span_depth_ratio is 40 with certainty 80.0)
    (then spacing_height_range is 20 to 30 with certainty 80.0))

(rule (if structural_system is column and material is rc)
    (then economic_span_depth_ratio is 20 with certainty 80.0)
    (then spacing_height_range is 18 to 25 with certainty 80.0))

(rule (if structural_system is column and material is precast)
    (then economic_span_depth_ratio is 20 with certainty 80.0)
    (then spacing_height_range is 20 to 30 with certainty 80.0))

(rule (if structural_system is column and material is timber)
    (then economic_span_depth_ratio is 20 with certainty 80.0)
    (then spacing_height_range is 20 to 30 with certainty 80.0))

(rule (if structural_system is column and material is composite)
    (then economic_span_depth_ratio is 20 with certainty 80.0)
    (then spacing_height_range is 25 to 30 with certainty 80.0))
```

Design Rules for Truss System

```
(defmodule Truss_DESIGN (import RULES ?ALL))

(defrule Truss_DESIGN::startit
?done <- (design_done no)
=>
    (retract ?done)
    (printout t "Starting the Truss Design..." crlf)
    (load-facts "Truss_Design_Knowledge_Base")
    (focus RULES)
    (assert (design_done yes)))
```

```
(defrule Truss_DESIGN::rule1
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_Truss)
       (eq ?val truss_type_1)
       (> ?cer 50.0)))
=>
    (printout t "The Truss type 1 is selected. " crlf)
    (dbmake_instance Truss))

(defrule Truss_DESIGN::rule2
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_Truss)
       (eq ?val truss_type_2)
       (> ?cer 50.0)))
=>
    (printout t "The Truss type 2 is selected. " crlf)
    (dbmake_instance Truss))

(deffacts Truss_DESIGN::the_Truss_design_rules
;;; the design rules

(rule (if material is steel and type is flat)
    (then span is 30 to 220 with certainty 100.0)
    (then spacing is 12 to 20 with certainty 100.0)
    (then economic_span_depth_ratio is 12 with certainty 80.0))

(rule (if material is steel and type is triangular)
    (then span is 30 to 150 with certainty 100.0)
    (then spacing is 12 to 20 with certainty 100.0)
    (then economic_span_depth_ratio is 7 to 8 with certainty 80.0))

(rule (if material is wood and type is flat)
    (then span is 40 to 160 with certainty 100.0)
    (then spacing is 12 to 20 with certainty 100.0)
    (then economic_span_depth_ratio is 8 to 10 with certainty 80.0))

(rule (if material is wood and type is tri)
    (then span is 40 to 100 with certainty 100.0)
    (then spacing is 12 to 20 with certainty 100.0)
    (then economic_span_depth_ratio is 10 with certainty 80.0))

(rule (if material is wood and type is hinged)
    (then span is 20 to 150 with certainty 100.0)
    (then spacing is 8 to 20 with certainty 100.0)
    (then economic_span_depth_ratio is 25 with certainty 80.0))
(rule (if structural_system is truss and material is steel)
    (then economic_span_depth_ratio is 10 to 12 with certainty 80.0)
    (then spacing_height_range is 20 to 25 with certainty 80.0))

(rule (if structural_system is truss and material is timber)
    (then economic_span_depth_ratio is 6 to 10 with certainty 80.0)
    (then spacing_height_range is 12 to 20 with certainty 80.0))

(rule (if structural_system is truss and material is pc)
    (then economic_span_depth_ratio is 10 with certainty 80.0)
    (then spacing_height_range is 25 to 30 with certainty 80.0))
(rule (if type is truss)
    (then economic_span_depth_ratio is 4 to 12 with certainty 80.0))
```

Design Rules for Footing

```
(defmodule Footing_DESIGN (import RULES ?ALL))
(defrule Footing_DESIGN::startit
?done <- (design_done no)
=>
    (retract ?done)
    (printout t "Starting the Footing Design..." crlf)
    (load-facts "Footing_Design_Knowledge_Base")
    (focus RULES)
    (assert (design_done yes)))

(defrule Footing_DESIGN::rule1
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_Footing)
      (eq ?val footing_type_1)
      (> ?cer 50.0)))
=>
```

```
    (printout t "The Footing type 1 is selected. " crlf)
    (dbmake_instance Footing))
(defrule Footing_DESIGN::rule2
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_Footing)
       (eq ?val footing_type_2)
       (> ?cer 50.0)))
=>
    (printout t "The Footing type 2 is selected. " crlf)
    (dbmake_instance Footing)
    )
(deffacts Footing_DESIGN::the_Footing_design_rules
;;; the design rules
(rule (if material is concrete)
    (then thickness is wall_thickness with certainty 80.0))
    (then width is 2.0 * wall_thickness with certainty 80.0))

(rule (if type is column)
    (then minimum_width is 3 with certainty 80.0)
```

Design Rules for Structural Wall

```
(defmodule Wall_DESIGN (import RULES ?ALL))
(defrule Wall_DESIGN::startit
?done <- (design_done no)
=>
    (retract ?done)
    (printout t "Starting the Wall Design..." crlf)
    (load-facts "Wall_Design_Knowledge_Base")
    (focus RULES)
    (assert (design_done yes)))

(defrule Wall_DESIGN::rule1
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_Wall)
       (eq ?val Wall_type_1)
       (> ?cer 50.0)))
=>
```

```
    (printout t "The Wall type 1 is selected. " crlf)
    (dbmake_instance Wall)
    )

(defrule Wall_DESIGN::rule2
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_Wall)
       (eq ?val Wall_type_2)
       (> ?cer 50.0)))
=>
(printout t "The Wall type 2 is selected. " crlf)
(dbmake_instance Wall)
)
(deffacts Wall_DESIGN::the_Wall_design_rules
;;; the design rules
(rule (if structural_system is wall and material is rc)
    (then minimum_thickness is 6 + (height-15)/25 with certainty 80.0))

(rule (if structural_system is wall and type is basement)
    (then minimum_thickness is 8 with certainty 80.0))

(rule (if structural_system is wall and material is rc)
    (then max_unbraced_length is 25 * thickness with certainty 80.0))

(rule (if structural_system is wall and material is plain_concrete)
    (then minimum_thickness is 7 with certainty 80.0))

(rule (if structural_system is wall and material is plain_concrete)
    (then max_unbraced_length is 22 * thickness with certainty 80.0))

(rule (if structural_system is wall and type is curtain)
    (then minimum_thickness is 4 with certainty 80.0))

(rule (if type is wall and type is curtain)
    (then max_unbraced_length is 30 * thickness with certainty 80.0))

(rule (if type is interior_wall and type is partition)
    (then minimum_thickness is 2 with certainty 80.0))

(rule (if type is interior_wall and type is partition)
    (then max_unbraced_length is 48 * thickness with certainty 80.0))

(rule (if type is bearing_wall and material is rc)
    (then minimum_thickness is 6 with certainty 80.0))

(rule (if type is bearing_wall and material is rc)
    (then max_unbraced_length is 25 * thickness with certainty 80.0))

(rule (if type is bearing_wall and material is masonry)
    (then minimum_thickness is 8 with certainty 80.0))

(rule (if type is bearing_wall and material is masonry)
    (then max_unbraced_length is 20 * thickness with certainty 80.0))

(rule (if type is bearing_wall and material is hollow_masonry)
    (then max_unbraced_length is 18 * thickness with certainty 80.0))

(rule (if type is bearing_wall and material is hollow_unit)
    (then minimum_thickness is 6 with certainty 80.0))

(rule (if type is bearing_wall and material is hollow_unit)
    (then max_unbraced_length is 25 * thickness with certainty 80.0))

(rule (if type is bearing_wall and material is masonry_panel)
    (then minimum_thickness is 4 with certainty 80.0))

(rule (if type is bearing_wall and material is masonry_panel)
    (then max_unbraced_length is 30 * thickness with certainty 80.0))

(rule (if type is bearing_wall and material is stone)
    (then minimum_thickness is 16 with certainty 80.0))

(rule (if type is bearing_wall and material is stone)
    (then max_unbraced_length is 14 * thickness with certainty 80.0))

(rule (if type is non_bearing_wall and material is stone)
    (then minimum_thickness is 4 with certainty 80.0))

(rule (if type is non_bearing_wall and material is stone)
    (then max_unbraced_length is 18 * thickness with certainty 80.0))
```

```
(rule (if type is load_bearing_wall and material is rc)
    (then max_unbraced_height is 25 with certainty 80.0))

(rule (if structural_system is load_bearing_wall and material is brick)
    (then economic_span_depth_ratio is 18 to 22 with certainty 90.0))

(rule (if structural_system is load_bearing_wall and material is concrete_block)
    (then economic_span_depth_ratio is 18 to 22 with certainty 90.0))

(rule (if structural_system is load_bearing_wall and material is brick_block)
    (then economic_span_depth_ratio is 18 to 22 with certainty 90.0))

(rule (if structural_system is load_bearing_wall and material is rc)
    (then economic_span_depth_ratio is 25 with certainty 90.0))

(rule (if structural_system is load_bearing_wall and material is precast)
    (then economic_span_depth_ratio is 25 with certainty 90.0))

(rule (if structural_system is load_bearing_wall and material is wood_stud)
    (then economic_span_depth_ratio is 25 with certainty 90.0))
```

Design Rules for Beam

```
(defmodule Beam_DESIGN (import RULES ?ALL))
(defrule Beam_DESIGN::startit
?done <- (design_done no)
=>
    (retract ?done)
    (printout t "Starting the Beam Design..." crlf)
    (load-facts "Beam_Design_Knowledge_Base")
    (focus RULES)
    (assert (design_done yes)))

(defrule Beam_DESIGN::rule1
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_Beam)
        (eq ?val Beam_type_1)
        (> ?cer 50.0)))
=>
    (printout t "The Beam type 1 is selected. " crlf)
    (dbmake_instance Beam)
    )

(defrule Beam_DESIGN::rule2
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_Beam)
        (eq ?val Beam_type_2)
        (> ?cer 50.0)))
=>
(printout t "The Beam type 2 is selected. " crlf)
;;;(dbmake_instance Beam)
)
(deffacts Beam_DESIGN::the_Beam_design_rules
;;; the design rules
(rule (if rc_availability is yes and Beam_material is rc)
    (then selected_Beam is Beam_type_1 with certainty 80.0)
)

(rule (if rc_availability is no and Beam_material is wood)
    (then selected_Beam is Beam_type_2 with certainty 80.0 )))

(rule (if structural_system is beam)
    (then economic_span_depth_ratio is 24 with certainty 80.0))

(rule (if type is beam and type cantilever)
    (then economic_overhang is 0.33 with certainty 80.0))

(rule (if type is beam and material is rc)
    (then typical_thickness is 8 to 12 with certainty 80.0)
    (then economic_span_depth_ratio is 12 to 16 with certainty 80.0))

(rule (if type is beam and type is continuous)
    (then typical_thickness is 8 to 10 with certainty 80.0)
    (then economic_span_depth_ratio is 16 with certainty 80.0))

(rule (if type is beam and material is rc)
    (then min_laterial_support is 32 with certainty 80.0))
```

```
(rule (if type is beam and material is pc and type is I)
    (then thickness is 12 to 16 with certainty 80.0)
    (then economic_span_depth_ratio is 20 to 25 with certainty 90.0))

(rule (if type is beam and material is pc and type is T)
    (then flange_thickness is 8 to 10 with certainty 80.0)
    (then web_thickness is 8 with certainty 100.0)
    (then economic_span_depth_ratio is 30 with certainty 90.0))

(rule (if type is beam and material is steel)
    (then span_range is 10 to 60 with certainty 100.0)
    (then total_bay_size is 20 to 24 with certainty 80.0)
    (then economic_span_depth_ratio is 20 with certainty 80.0)
    (then sectional_modulus is depth * weight_per_foot with certainty 80.0))

(rule (if type is girder and material is steel)
    (then span_range is 20 to 80 with certainty 100.0)
    (then economic_span_depth_ratio is 14 with certainty 80.0))

(rule (if type is beam and material is wood)
    (then thickness_range is 2 to 14 with certainty 100.0)
    (then span_range is 8 to 32 with certainty 100.0)
    (then economic_span_depth_ratio is 16 to 20 with certainty 80.0))

(rule (if type is beam and material is glued_laminated_wood)
    (then thickness_range is 3.25 to 9 with certainty 100.0)
    (then span_range is 16 to 50 with certainty 100.0)
    (then economic_span_depth_ratio is 20 to 24 with certainty 80.0))

(rule (if type is joist and material is precast)
    (then spacing_range is 3.25 to 9 with certainty 100.0)
    (then span_range is 20 to 45 with certainty 100.0)
    (then economic_span_depth_ratio is 20 to 24 with certainty 80.0)
    (then depth is 4.0 * width_of_flange with certainty 80.0))

(rule (if type is joist and material is pc)
    (then spacing_range is 3.25 to 9 with certainty 100.0)
    (then span_range is 40 to 60 with certainty 100.0)
    (then economic_span_depth_ratio is 32 with certainty 80.0)
    (then depth is 4.0 * width_of_flange with certainty 80.0))

(rule (if type is joist and material is steel and type is openweb)
    (then spacing_range is 24 with certainty 100.0)
    (then span_range is 8 to 48 with certainty 100.0)
    (then economic_span_depth_ratio is 20 to 24 with certainty 80.0)
    (then depth is 4.0 * width_of_flange with certainty 80.0))

(rule (if type is joist and material is wood and type is floor)
    (then spacing_range is 16 with certainty 100.0)
    (then span_range is 8 to 25 with certainty 100.0)
    (then economic_span_depth_ratio is 20 with certainty 80.0))
(rule (if structural_system is beam and material is steel)
    (then economic_span_depth_ratio is 20 to 25 with certainty 80.0)
    (then spacing_height_range is 20 to 40 with certainty 80.0))

(rule (if structural_system is beam and material is rc)
    (then economic_span_depth_ratio is 16 to 21 with certainty 80.0)
    (then spacing_height_range is 20 to 40 with certainty 80.0))

(rule (if structural_system is beam and material is pc)
    (then economic_span_depth_ratio is 10 to 20 with certainty 80.0)
    (then spacing_height_range is 20 to 60 with certainty 80.0))

(rule (if structural_system is beam and material is timber)
    (then economic_span_depth_ratio is 20 to 30 with certainty 80.0)
    (then spacing_height_range is 15 to 30 with certainty 80.0))

(rule (if structural_system is beam and material is laminated_wood)
    (then economic_span_depth_ratio is 20 to 30 with certainty 80.0)
    (then spacing_height_range is 15 to 30 with certainty 80.0))

(rule (if structural_system is beam and material is composite)
    (then economic_span_depth_ratio is 24 with certainty 80.0)
    (then spacing_height_range is 20 to 40 with certainty 80.0))

(rule (if structural_system is beam and material is post_tension_concrete)
    (then economic_span_depth_ratio is 26 to 35 with certainty 80.0)
    (then spacing_height_range is 20 to 60 with certainty 80.0))
(rule (if structural_system is wall_beam and material is steel)
    (then economic_span_depth_ratio is 8 to 10 with certainty 80.0)
    (then spacing_height_range is 12 to 24 with certainty 80.0))

(rule (if structural_system is wall_beam and material is concrete)
    (then economic_span_depth_ratio is 8 to 10 with certainty 80.0)
    (then spacing_height_range is 12 to 24 with certainty 80.0))
```

Design Rules for Slab

```
(defmodule Slab_DESIGN (import RULES ?ALL))
(defrule Slab_DESIGN::startit
?done <- (design_done no)
=>
    (retract ?done)
    (printout t "Starting the Slab Design..." crlf)
    (load-facts "Slab_Design_Knowledge_Base")
    (focus RULES)
    (assert (design_done yes)))

(defrule Slab_DESIGN::rule1
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_Slab)
      (eq ?val Slab_type_1)
      (> ?cer 50.0)))
=>
    (printout t "The Slab type 1 is selected. " crlf)
    (dbmake_instance Slab)
    )

(defrule Slab_DESIGN::rule2
?ss <- (attribute (name ?rel) (value ?val) (certainty ?cer))
(test (and (eq ?rel selected_Slab)
      (eq ?val Slab_type_2)
      (> ?cer 50.0)))
=>
(printout t "The Slab type 2 is selected. " crlf)
;;;(dbmake_instance Slab)
)
```

```
(deffacts Slab_DESIGN::the_Slab_design_rules
;;; the design rules
(rule (if rc_availability is yes and Slab_material is rc)
    (then selected_Slab is Slab_type_1 with certainty 80.0)
)
(rule (if type is slab and material is rc)
    (then economic_span_depth_ratio is 20 to 35 with certainty 80.0))

(rule (if type is slab and material is rc)
    (then thickness_span_ratio is 0.05  with certainty 80.0))

(rule (if structural_system is slab and material is rc)
    (then max_steel_beam_spacing is 8 with certainty 70.0)
    (then typical thickiness is 4 with certainty 80.0))

(rule (if structural_system is slab and material is rc and type is waffle)
    (then span is 25 to 40 with certainty 80.0)
    (then max_long_to_short_ratio is 1.33 with certainty 80.0)
    (then economic_span_depth_ratio is 25 with certainty 80.0))

(rule (if structural_system is slab and material is rc and type is flat)
    (then span is 16 to 36 with certainty 80.0)
    (then thickness is 6 to 12 with certainty 80.0)
    (then max_long_to_short_ratio is 1.33 with certainty 80.0)
    (then economic_span_depth_ratio is 30 to 40 with certainty 80.0))
```

Appendix B

ACL API Specification

class Facilitator methods

Function Name

Facilitator::**Facilitator**(void)

Description

Constructor for the **Facilitator** class.

Parameters

None

Return Value

None

Example of Use

Facilitator**fac = new* **Facilitator***;*

This creates a new instance of the **Facilitator** object. This will probably be called once at the beginning of the program.

Function Name

bool **Facilitator**::connect(char* agent_name, char* facilitator_name, char* facilitator_address, int port_num)

Description

Called when an agent wants to make a connection to the facilitator. Sets up the network connection. Called when an agent starts running, or after he has disconnected and wishes to connect again.

Parameters

char* agent_name //unique name by which other agents know the connecting agent.

char* facilitator_name //unique name by which the facilitator is known.

char* facilitator_address //the facilitator address that the agent is trying to connect to.

int port_num //the port number of the facilitator that the agent is trying to connect to.

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->connect("stanford-agent", "dcepo", "hpdce.stanford.edu", 4010);

This connects the agent named "stanford-agent" to the facilitator named "dcepo" to port 4010 of "hpdce.stanford.edu"

Function Name

bool **Facilitator**::disconnect(void)

Description

Called when an agent temporarily no longer wishes to communicate with the facilitator. Closes the network connection.

Parameters

None

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->disconnect();

This disconnects the agent from the facilitator. The facilitator will remember the agent, and will save messages for the agent until the next time the agent connects.

Function Name

bool **Facilitator**::forget(void)

Description

Tells the facilitator to permanently forget all information about the calling agent. Used if an agent wishes to leave the project entirely, or wants to start over for some reason.

Parameters

None

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->forget();

The facilitator will delete the information about the agent. No messages will be saved for this agent.

Function Name

bool **Facilitator**::set_name_space(char* name_space)

Description

Sets the agent's namespace. Should be called after connect. Can be called only one time per connection.

Parameters

name_space

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->set_name_space("mit");

This sets the namespace to "stanford". All objects that the agent communicates about will belong to this namespace.

Function Name

bool **Facilitator**::register_callback(void (*callback)(**Message***))

Description

Registers a callback that will be called upon receipt of a message from the network. When a message is received from the network, the API checks to see if the user has registered a callback. If so, the **Message** is passed to this callback routine immediately. Otherwise, the default action takes place: the **Message** is put in a queue to be retrieved with **Facilitator**::get_message(). After registering a callback, the user may remove the callback by calling this method with NULL as the argument. This method was not a part of the original specifications. It was unofficially incorporated into version 1.11 as of 7/11/95.

Example of usage:

extern void my_callback_routine(**Message*** msg);

facilitator->register_callback(my_callback_routine);

Parameters

void (*callback)(**Message***) Routine that will be called when a message is received from the network. callback is a pointer to a function of return type void that takes a pointer to a **Message** object as an argument. If this argument is NULL, it will have the effect of removing a callback that had been previously registered.

Return Value

TRUE if successful, otherwise FALSE.

Function Name

char** **Facilitator**::get_connected_agents(void)

Description

Returns a NULL-terminated list of names of all agents currently connected.

Parameters

None

Return Value

Returns a pointer to a list of character strings on success. The final character string is NULL. Returns NULL on failure.

Example of Use

```
char** agent_names = fac->get_connected_agents();
while ( *agent_names != NULL ) {
    cout << *agent_names << endl;
    agent_names++;
}
```

Prints out the names of the connected agents, one per line.

Function Name

char** **Facilitator**::get_all_agents(void)

Description

Returns a NULL-terminated list of names of all of the agents that the facilitator has knowledge of.

Parameters

None

Return Value

Returns a pointer to a list of character strings on success. The final character string is NULL. Returns NULL on failure.

Example of Use

```
char** agent_names = fac->get_all_agents();
while ( *agent_names != NULL ) {
    cout << *agent_names << endl;
    agent_names++;
}
```

Prints out the names of all agents, one per line.

Function Name

bool **Facilitator**::message_waiting(void)

Description

Returns TRUE if there are messages waiting in the queue.

Parameters

None

Return Value

TRUE if there are messages waiting, otherwise FALSE.

Example of Use

```
if ( fac->message_waiting() ) {
    cout << "Message waiting!" << endl;
}
```

Prints a message if there is a message waiting. Usually followed by a call to get_message if there is a message waiting.

Function Name

Message* **Facilitator**::get_message(void)

Description

Returns the message that is at the head of the queue.

Parameters

None

Return Value

Returns the address of a **Message** object on success. Returns NULL on error or if there is no message waiting.

Example of Use

Message* msg = fac->get_message();

Gets an incoming message from the facilitator. Usually preceeded by a call to message_waiting.

Function Name

bool **Facilitator**::commit(void)

Description

Commits the current transaction. Sends the changes that the agent has been making to the facilitator. Until this is called, all operations are buffered.

Parameters

None

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->commit();

All messages generated by calls to methods such as construct_instance will be sent to the facilitator. Only messages that have been generated since the last call to commit or abort are sent.

Function Name

bool **Facilitator**::abort(void)

Description

Aborts the current transaction. The pending changes will not be sent to the facilitator.

Parameters

None

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->abort();

All messages generated by calls to methods such as construct_instance will be thrown away. Only messages that have been generated since the last call to commit or abort will be thrown away.

Function Name

bool **Facilitator**::construct_instance(char* class_name, char* object_name)

Description

Signals to the facilitator that an instance has been created.

Parameters

char* class_name The name of the class that is being created. char* object_name The name of the instance that is being created.

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->construct_instance(“column”, “col1”);

Constructs an instance of the column class called “col1”.

Function Name

bool **Facilitator**::destruct_instance(char* class_name, char* object_name)

Description

Signals to the facilitator that an instance has been destroyed.

Parameters

char* class_name; //The name of the class that is being created.

char* object_name; //The name of the instance that is being created.

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->destruct_instance(“column”, “col1”);

Destructs an instance of the column class called “col1”.

Function Name

bool **Facilitator**::add_value(char* class_name, char* object_name, char* slot_name, char* value)

Description

Adds a slot to an object that has been created with construct_instance.

Parameters

char* class_name; // The name of the class that is having a value added.

char* object_name // The name of the object that is having a value added.

char* slot_name // The name of the slot that is being added.

char* value // The value of the slot that is being added.

Conforms to the syntax described in the “Usage” section.

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->add_value(“column”, “col1”, “width”, “1”);

Adds the slot “width” to the column “col1”, and sets its value to 1.

Function Name

void **Facilitator**::modify_value(char* class_name, char* object_name, char* slot_name, char* value)

Description

Associates a new value with a slot of a class.

Parameters

char* class_name; // The class of the object whose slot value is being modified.

char* object_name; // The name of the object whose slot value is being modified.

char* slot_name; // The name of the slot whose value is being modified.

char* value; // The value of the slot that is being modified.

Conforms to the syntax outlined in the “Usage” section.

Return Value

None

Example of Use

fac->modify_value(“column”, “col1”, “width”, “2”);

Changes the slot “width” of the column “col1” to 2.

Function Name

bool **Facilitator**::delete_value(char* class_name, char* object_name, char* slot_name)

Description

Removes a slot from a class.

Parameters

char* class_name; //The class name of the object whose slot is being removed.

char* object_name; //The name of the object whose slot is being removed.

char* slot_name; //The name of the slot that is being removed.

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->delete_value("column", "col1", "width");

Deletes the slot "width" from the column "col1".

Function Name

bool **Facilitator**::add_relation(char* relation_name, char* parent_class_name, char* parent_object_name, char* child_class_name, char* child_object_name)

Description

Adds a relationship to a class.

Parameters

char* relation_name; // The name of the relationship to be added. eg, "has-a", "is-a".

char* parent_class_name; // The class name of the parent object.

char* parent_class_name; // The class name of the parent object.

char* child_class_name; // The class name of the child object.

char* child_object_name; //The name of the child object.

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->add_relation("has-a", "room", "room1", "door", "door1");

Adds a "has-a" relationship to room "room1". This relation indicates that room "room1" has a door "door1".

Function Name

bool **Facilitator**::delete_relation(char* relation_name, char* parent_class_name, char* parent_object_name, char* child_class_name, char* child_object_name)

Description

Deletes a relationship from a class.

Parameters

char* relation_name; // The name of the relation to be deleted. eg, "is-a", "has-a".

char* parent_class_name; // The class name of the parent object.

char* parent_class_name; // The class name of the parent object.

char* child_class_name; // The class name of the child object.

char* child_object_name; //The name of the child object.

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->delete_relation("has-a", "room", "room1", "door", "door1");

Deletes the "has-a" relationship from room "room1". This relation indicates that room "room1" has a door "door1".

Facilitator Methods for Inserting / Removing Interests

Function Name

bool **Facilitator**::int_all(char* class_name, char** slot_names, char** relation_names, int_action action)

Description

Depending on the value of action, this method inserts / removes interests about all operations for an object.

Parameters

char* class_name; // The name of the class whose relation you're interested in.

char** slot_names; // NULL terminated list of slot names that the agent is interested in. In general, this will be a list of all of the slot names of the class. If there is no slot name for the class, pass NULL for this argument.

char** relation_name; //NULL terminated list of relation names that the agent is interested in. In general, this will be a list of all of the relation names that the class may posess. If there is no relation name for the class, pass NULL for this argument.

int_action action The value for action is either k_int_insert (for inserting the interests) or k_int_remove (for removing the interests).

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

char* slot_names[] = {"width", "height", "x", "y", NULL};

char* relation_names[] = {"has-a", "is-a", NULL};

fac->int_all("column", slot_names, relation_names, k_int_insert);

Specifies that the agent is interested in all actions that happen to column objects, including actions that affect the slots "width", "height", "x" and "y", and the "is-a" and "has-a" relations.

Function Name

bool **Facilitator**::int_construct_instance(char* class_name, int_action action)

Description

Indicates to the facilitator that the agent is interested / uninterested (depending on the value of action) in receiving messages about the construction of a certain class.

Parameters

char* class_name; //The class that that the agent is interested / uninterested in.

int_action action; // The value for action is either k_int_insert (for inserting the interest) or k_int_remove (for removing the interest).

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->int_construct_instance("column", k_int_insert);

Specifies that the agent is interested in column construction messages.

Function Name

bool **Facilitator**::int_destruct_instance(char* class_name, int_action action)

Description

Indicates to the facilitator that the agent is interested / uninterested (depending on the value of action) in receiving messages about the destruction of a certain class.

Parameters

char* class_name; // The class that that the agent is interested / uninterested in.

int_action action; // The value for action is either k_int_insert (for inserting the interest) or k_int_remove (for removing the interest).

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->int_destruct_instance("column", k_int_insert);

Specifies that the agent is interested in column destruction messages.

Function Name

bool **Facilitator**::int_add_value(char* class_name, char* slot_name, int_action action)

Description

Indicates to the facilitator that the agent is interested / uninterested (depending on the value of action) in receiving messages about adding values to a class.

Parameters

char* class_name; //The class that that the agent is interested / uninterested in.

char* slot_name; //the slot value that that the agent is interested / uninterested in.

int_action action The value for action is either k_int_insert (for inserting the interest) or k_int_remove (for removing the interest).

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->int_add_value("column", "width", k_int_insert);

Specifies that the agent is interested in messages that indicate that the slot value "width" has been added to a column.

Function Name

bool **Facilitator**::int_delete_value(char* class_name, char* slot_name, int_action action)

Description

Indicates to the facilitator that the agent is interested / uninterested (depending on the value of action) in receiving messages about deleting values from a class.

Parameters

char* class_name; //the class that that the agent is interested / uninterested in.

char* slot_name; //the slot value that that the agent is interested /uninterested in.

int_action action; //The value for action is either k_int_insert (for inserting the interest) or k_int_remove (for removing the interest).

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->int_delete_value("column", "width", k_int_insert);

Specifies that the agent is interested in messages that indicate that the slot value "width" has been deleted from a column.

Function Name

bool **Facilitator**::int_modify_value(char* class_name, char* slot_name, int_action action)

Description

Indicates to the facilitator that the agent is interested / uninterested (depending on the value of action) in receiving messages about changing slot values.

Parameters

char* class_name; // the class that that the agent is interested / uninterested in.

char* slot_name; //the slot value that that the agent is interested / uninterested in.

int_action action The value for action is either k_int_insert (for inserting the interest) or k_int_remove (for removing the interest).

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->int_modify_value("column", "width", k_int_insert);

Specifies that the agent is interested in messages that indicate that the slot value "width" of a column has been modified.

Function Name

bool **Facilitator**::int_add_relation(char* class_name, char* relation_name, int_action action)

Description

Indicates to the facilitator that the agent is interested / uninterested (depending on the value of action) in receiving messages about relations being added.

Parameters

char* class_name; // the class name whose relation you're interested / uniterested in.

char* relation_name; // the name of the relation that you are interested / uniterested in.

int_action action; // the value for action is either k_int_insert (for inserting the interest) or k_int_remove (for removing the interest).

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->int_add_relation("column", "has-a", k_int_insert);

Specifies that the agent is interested if a "has-a" relationship is added to a column.

Function Name

bool **Facilitator**::int_delete_relation(char* class_name, char* relation_name, int_action action)

Description

Indicates to the facilitator that the agent is interested / uninterested (depending on the value of action) in receiving messages about relations being added.

Parameters

char* class_name; // the class whose relation you're interested / uninterested in.

char* relation_name; //the relation that you are interested / uninterested in. eg, “is-a”, “has-a”.

int_action action; //The value for action is either k_int_insert (for inserting the interest) or k_int_remove (for removing the interest).

Return Value

TRUE if successful, otherwise FALSE.

Example of Use

fac->int_delete_relation(“column”, “has-a”, k_int_insert);

Specifies that the agent is interested if a “has-a” relationship is deleted from a column.

class Message Methods

Function Name

int **Message**::type(void)

Description

Returns the type of the **Message**.

Parameters

None

Return Value

ACL_ObjectCreate, ACL_ObjectDestroy, ACL_AddValue, ACL_ModifyValue, ACL_ModifyObject, ACL_DeleteValue, ACL_AddRelation, ACL_DeleteRelation

---

Function Name

char* **Message**::msg_class(void)

Description

Returns class that a message refers to.

Parameters

None

Return Value

NULL on failure. Returns the class name on success.

Function Name

char* **Message**::object(void)

Description

Returns the global name of the object that the message refers to.

Parameters

None

Return Value

NULL on failure. Returns the object name on success.

Function Name

char* **Message**::slot(void)

Description

Returns the name of the slot that the message refers to.

Parameters

None

Return Value

NULL on failure. Returns the slot name on success.

Function Name

char* **Message**::value(void) Description

Returns a character string that represents the value of the message. The character string conforms to the grammar that is outlined in the "Usage" section.

Parameters

None

Return Value

NULL on failure. Returns the slot value on success.

Function Name

char* **Message**::child_class(void)

Description

Returns the class name of the child object that the message refers to.

Parameters

None.

Return Value

NULL on failure. Returns the class name on success.

Function Name

char* **Message**::child_object(void)

Description

Returns the global name of the child object that the message refers to.

Parameters

None.

Return Value

NULL on failure. Returns the object name on success.

Function Name

bool **Facilitator**::request_attribute_value(Undetermined)

Description

Requests the value of an attribute of a specific object.

Parameters

Undetermined

Return Value

Undetermined

Function Name

bool **Facilitator**::request_relation_value(Undetermined)

Description

Requests the value of a relationship of a specific object.

Parameters

Undetermined

Return Value

Undetermined

Function Name

bool **Facilitator**::tell_attribute_value(Undetermined)

Description

Announces the value of an attribute of a specific object.

Parameters

Undetermined

Return Value

Undetermined

Function Name

bool **Facilitator**::tell_relation_value(Undetermined)

Description

Announces the value of a relationship of a specific object.

Parameters

Undetermined

Return Value

Undetermined

Appendix C

BNF Grammar of Object Modeling Language

Items which are italicized are grammar element names. Items in bold type are keywords in the language. Items enclosed in single quotes are literal characters of the language. Alternative right hand side clauses are separated by a vertical bar (|). Any grammar clause name (for example <class-name>) which ends with "-name" is intended to represent a character string for the name of an object. Grammar elements enclosed within brackets (i.e. [ ]) are optional elements within the clause.

D.1 Base Data Type

Base data types define the underlying representation of a value. The notation described here is borrowed from SQL.

<base-data-type>:

number `(' <precision> [ `,' <decimals>] `)' |

char `(' <length> `)' |

date

D.2 Aggregation

An aggregation defines constraints on a collection of base data type values. Matrix and interval aggregations are defined for real numbers.

<aggregation> :

<base-data-type> **RANGE** <low-literal> <high-literal> |

<base-data-type> **SET** <literal-list> |

<base-data-type> **ONEOF** <literal-list> |

<base-data-type> **CONST** |

<base-data-type> **SINGLE** |

<base-data-type> **MANY** |

<base-data-type> **EXPR** |

MATRIX <rows> <columns> |

INTERVAL <low-bound> <high-bound> |

VECTOR <domain> |

TUPLE `(` <element-list> `)'

D.3 Element List

<element-list>:

<element> |

<element-list> `,' <element>

<element>:

<attribute-name> <domain>

D.4 Domain

A domain is defined by a name and an aggregation. By naming a set of values, other systems (e.g. a facilitator) can easily map values from one to another.

<domain> : **DECLARE DOMAIN** <domain-name> <aggregation> `;'

D.5 Subdomain

Subdomains refine a domain by adding additional constraints on the parent domain. When a subdomain is declared, no relationship with the parent domain is maintained.

<subdomain> : **DECLARE SUBDOMAIN** <sub-domain-name><domain-name> [<subaggregation>] `;'

<subaggregation>: **ONEOF** <literal-list> | /* parent must be **ONEOF** */

SET <literal-list> | /* parent must be **SET** or **MANY** */

RANGE <low-literal> <high-literal> /* low and high literal must be in the parent */

D.6 Class

A class defines a collection of attributes some of which can be inherited from a parent class. A predefined class named Object always exists.

<class> :

DECLARE CLASS <class-name> <class-category-name> **SUBCLASS OF** <class-name> `(` [<attribute-list>] `)' `;'

D.7 Forward Class

Forward classes are used to break circular references of classes.

<forward-class>:

DECLARE FORWARD CLASS <class-name> `;'

D.8 Attribute List

<attribute-list>:

<attribute> |

<attribute-list> `,' <attribute>

D.9 Attribute

<attribute>:

[<io-tag>] [DYNAMIC] **VALUE** <attribute-name> <domain-name> [<structured-value>] |

RELATION <attribute-name> <relationship-type-name> <class-name> |

TRIGGER <attribute-name> ON <trigger-action> `"<expression(1)> `"

<io-tag>:

INPUT | OUTPUT | PRIVATE

D.10 Trigger Action

Triggers can be specified for the addition/deletion of a frame instance to/from an instance network.

<trigger-action>:

ADD | DELETE

D.11 Literal List

<literal-list>:

<literal> |

<literal-list> `,' <literal>

D.12 Literal

<literal>:

<integer> | <real> | <string> | <date>| <argument> | NIL | MAXINT | MAX-FLOAT

D.13 Argument

Arguments are automatically numbered by the runtime system starting at 1.

<argument>:

`$' <integer>

D.14 Variable

<variable>:

`$' <variable-name>

D.15 Relationship Type

<relationship-type>:

DECLARE RELATIONTYPE <relationship-type-name> <container-type> ';'

D.16 Container Type

Container types specify the cardinality and duplication constraints on a relationship. Link defines that a single value can exist in the container. Vector is an ordered list. Set is an ordered list with no duplications.

<container-type>:

LINK|

VECTOR|

SET

D.17 Expression

<expression>:

`(` <expression> `)' |

<expression> `+' <expression> |

<expression> `-' <expression> |

<expression> `*' <expression> |

<expression> `/' <expression> |

<expression> `^' <expression> |

<expression> `;' <expression> |

<builtin-function> <arg-list> |

this `:' <slot-name> [<index>] <arg-list> |

this ':' <slot-name> [<index>] `:=' <expression> |

<variable> `:' <slot-name> <arg-list> |

<relation-slot-name> `:' <slot-name> <arg-list> |

<relation-slot-name> `:' <relation-index> `:=' <expression> |

<relation-slot-name> `:' <relation-index> |

<literal> |

<conditional-expression>

D.18 Index

<index>:

<indexElement> |

<index> <indexElement>

<memberIndex>:

<string> |

<argument>

<indexValue>:

<integer> |

<argument>

<indexElement>:

`{` <memberIndex> `}' |

`[` <indexValue> `]'

D.19 Relation Index

<relation-index>:

`[` [<indexValue>] `]'

D.20 Structured Value

<structured-value>:

<SimpleValueList> |

```
<StructuredValueList>

<StructuredValue>:

`{` <SimpleValueList> `}' |

`[` <SimpleValueList> `]' |

`{` <StructuredValueList> `}' |

`[` <StructuredValueList> `]'

<StructuredValueList>:

StructuredValue |

StructuredValueList StructuredValue

<SimpleValueList>:

SimpleValue|

SimpleValueList SimpleValue

<SimpleValue>:

  <string> | <integer> | <real> | NIL | <variable>
```

Appendix D

BNF Grammar of Instance Manipulation Language

In the section, we present the grammar for manipulating instances in the object model. An object model of the instance network is shown in Figure E. The instance manipulation language exists at the same level as the object declaration language. In other words, object definitions and instance manipulation commands can be intermingled.

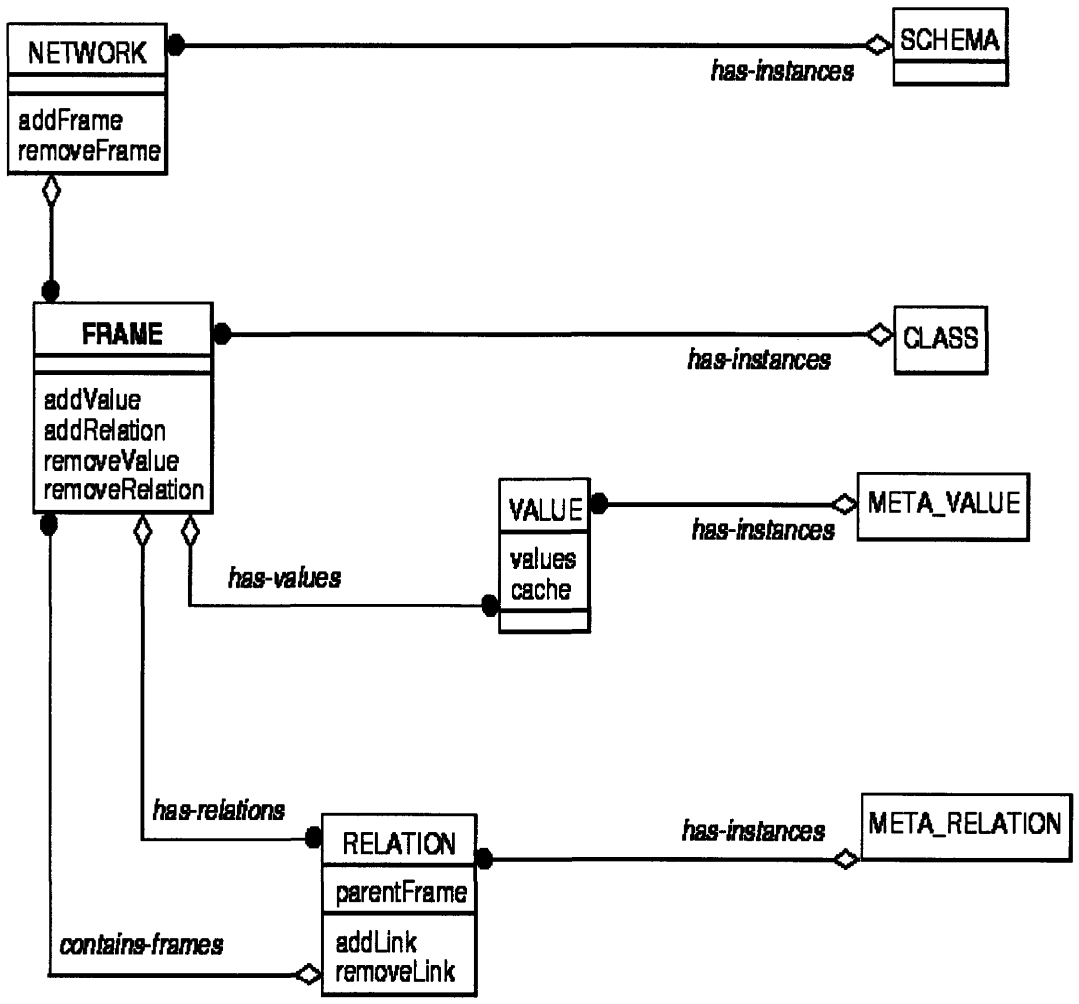


FIGURE 1. Instance Network in OMT Notation

E.1 Frame Action

<frame-action>:

ADD | DELETE

E.2 Value Actions

<value-actions>:

ADD | MODIFY | DELETE `;'

E.3 Frame Commands

<frame-command>:

<frame-action> **FRAME** <class-name> `;'

E.4 Value Commands

<value-command>:

<value-action> **VALUE** <class-name> <slot-name> <instance> <value-list> `;' |

<value-action> **VALUE** `$' <variable-name>(3) <slot-name> <value-list> `;'

E.5 Relationship Commands

<relationship-command>:

<frame-action> **RELATION** <class-name> <slot-name> <class-name> <instance> <instance> `;' |

<frame-action> **RELATION** `$' <variable-name> <slot-name> `$' <variable-name> `;'

E.6 Variable Setting Command

Variables can be used to store object identities and used later. This can be used to load and save a knowledge base without

referencing instance numbers. This scheme will work even when instances have a circular reference. The most common use of this

command is to capture the object identity of CURRENT.

<set-variable-command>:

SET `$' <variable-name> TO `$' <variable-name> |

SET `$' <variable-name> TO <class-name> <instance>

E.7 Evaluation Commands

When a slot is evaluated, arguments can be passed. Each argument is listed and is numbered from 1 to n in the evaluated slot. The runtime environment must support the notion of an argument stack similar to a programming language function call stack.

<evaluation-command>:

EVAL **VALUE** <class-name> <slot-name> <instance> [<value>] `;' |

EVAL **VALUE** `$' <variable-name> <slot-name> [ <value> ] `;'

E.8 Value

<value>:

<SimpleValueList> |

<StructuredValueList>

<StructuredValue>:

`{` <SimpleValueList> `}' |

`[` <SimpleValueList> `]' |

`{` <StructuredValueList> `}' |

`[` <StructuredValueList> `]'

<StructuredValueList>:

StructuredValue |

StructuredValueList StructuredValue