NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

A2: An Architectural Agent in a Collaborative Engineering Environment

Ulrich Flemming, Zeyno Aygen, James Snyder and Jonah Tsai

EDRC 48-38-96

A2: An Architectural Agent in a Collaborative Engineering Environment

Ulrich Flemming
Zeyno Aygen
James Snyder
Jonah Tsai

November 1996

ABSTRACT

This report describes the functionality of the Architectural Agent (A2) developed by a team at CMU as part of the US ACERL-sponsored ACL (Investigation of an Agent Communication Language) project, which addresses communication issues in a distributed computer-supported building design environment The core functionality of A2 consists of generating schematic layouts of the functional units in a spatial program; this functionality is provided by using SEED-Layout (SL) as a server; SL is a module in the Software Environment to Support the Early Phases in Building Design (SEED) currently under development at the EDRC. A2 is in addition served by the object database UniSQL and AutoCAD, used strictly for display purposes. These diverse components communicate through a message server with each other and a facilitator API provided by another team. The communication protocol and data translations are based on an active object model expressed in the specification language OML developed by the CMU team explicitly for the ACL project The CMU team also suggests an approach toward conflict management and negotiation for an environment like ACL and describes an OML-based protocol that implements the approach. It offers general conclusions about relevant aspects of the ACL project.

ACKNOWLEDGMENTS

The work described here was sponsored by the US Army Corps of Engineers Construction Engineering Research Laboratory (USACERL) under contract DACA-88-93-D-0004.

1 INTRODUCTION

This report describes the contribution of a team at CMU to the USACERL-sponsored ACL (Investigation of an Agent Communication Language) project. The goal of the ACL project was to investigate the use of concurrent engineering (CE) technologies to support collaboration between participants in the facility design and delivery process. The project recognized that the lack of a standardized representational syntax and semantics is a major barrier to implementing CE techniques in this application domain. It intended to determine fundamental functional requirements of a technology that would allow intelligent agents to cooperate in facility design.[1]

In this context, the term *agent* refers to a software configuration at a specific site that

- supports a specific design task, which is not necessarily automated; that is, it may involve human interaction and intervention,
- generates design information in its own internal representation, and
- is able to communicate its design decisions to other agents at different sites.

In order to secure the cooperation of a sufficient number of agents in the project, USACERL organized a team of researchers representing four major universities: the Massachusetts Institute of Technology (MIT), Stanford University (Stanford), the University of Illinois at Urbana-Champaign (UIUC), and Carnegie Mellon University (CMU). The collaboration between the teams was to make use of technologies developed independently by the four universities and USACERL, which included

- the Agent Collaboration Environment (ACE) developed by USACERL,[2] which in turn served as front-end for the energy analysis system BLAST developed by UIUC[3] and two additional agents assuming the tasks, respectively, of the building owner and project manager,
- the structural design agent developed by MIT,[4] and
- the architectural layout system SEED-Layout developed at CMU,[5] which was to execute all architectural tasks in the ACL project.

The sponsor had established at the outset that the cooperation of these systems was to be based on the agent-based Federation Architecture developed by Stanford, which intends to allow independent software agents to exchange information by communicating with *facilitators* through appropriately designed APIs. The facilitators are responsible for message brokering, based on interests registered with a facilitator by the individual agents, and data translations, based on production-type translation rules.[6] Figure 1 depicts the top-level agent connections in the ACL project.

---

1. **The goal statements are taken almost verbatim from Contract DAC88-93-D-0004(010) between the US Army Corps of Engineers Construction Engineering Research Lab and CMU.**
2. **McGraw, K. (1996) *Agent Collaboration Environment (ACE),* http://www.cecer.army.mil/pl/ace/.**
3. **BLAST User's Reference Vol. I and II (1991) BLAST Support Office, Dept. of Mechanical and Industrial Engineering, Univ. of Illinois at Urbana-Champaign**
4. **Chiou, J. and Logcher, R.D.(1996) Testing a Federation Architecture in Collaborative Design Process. Res. Report R96-01. Dept. of Civil and Environmental Engineering, MIT, Cambridge, MA**
5. **Flemming, U. and Chien, S.F (1995) "Schematic Layout Design in SEED Environment" *Journal of Architectural Engineering,* vol. 1 162-169**

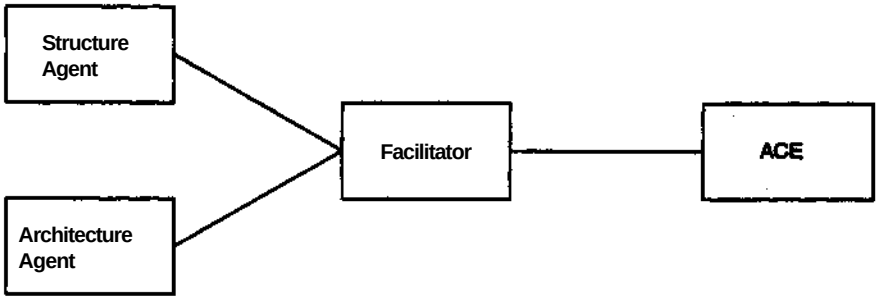


Figure 1. Top-Level ACL Agent Connections

The present report describes the contributions of the CMU team to the ACL project and summarizes its experience with the federated, facilitator-based ACL architecture. The sections of this report fall into three major parts that are by and large self-contained:

Agent Functionality

The first three sections describe the capabilities of the Architectural Agent (A2) developed by the CMU team in domain terms. Section 2 specifies the overall functionality of the agent in terms of the broad tasks it is able to support. Section 3 describes several scenarios meant to convey a more concrete understanding of the tasks and how they interact with tasks performed by other ACL agents in the overall design process as supported by the ACL environment. Section 4 identifies the 'use cases' as they emerge from these scenarios.

Agent Implementation

Section 5 describes data communication issues in the ACL project and introduces the approach taken by the ACL team to handle them, which is based on the Object Modeling Language OML developed by the CMU team specifically for the ACL project. Section 6 describes the current implementation of A2 as a configuration of independently developed subagents that communicate via an OML-based object model.

Conflict Management and Negotiation in the ACL project

Section 8 introduces an approach toward negotiation and conflict management that is suitable for a project like ACL. Section 9 specifies an OML-based negotiation protocol implementing the approach and illustrates the use of this protocol through two examples.

A concluding section summarizes the experience of the CMU team in the ACL project, especially with respect to the Federation Architecture.

---

6. T. Khedro, M. Case, U. Flemming, M. Genesereth, R. Lpgcher, C. Pedersen, J. Snyder, R. Sriram, and P. Teicholz (1995) "Development of Multi-Institutional Testbed for Collaborative Facility Engineering Infrastructure" *Proc. Second Congress on Computing in Civil Engineering* (Atlanta, GA, June 1995) J.P. Mohsen, ed., American Society of Civil Engineers, New York, NY, 1308-1315

2 OVERALL FUNCTIONALITY OF THE ARCHITECTURAL AGENT

This section specifies the overall functionality of the agent developed by the CMU team within the ACL project. We refer to this agent as the *architectural agent* (A2 for short) because its tasks cover the traditional responsibilities of architects during the preliminary building design phase. The core of the capabilities demanded from A2 is to be provided by SEED-Layout[7], one of three modules currently under development at the EDRC to establish SEED, a Software Environment to Support the Early Phases in Building Design[8].

The functionality of A2 consists of support for the tasks listed below, where *support' does not necessarily mean that A2 is able or expected to execute a task automatically. Tasks that involve complex evaluations and judgment are principally left to the user; A2 only has to record and store the user's decisions accurately. Other tasks may be executed automatically, interactively, or through a combination of these two 'generation modes[9]. For example, A2 may generate a building component like a door automatically and place it based on rules of thumb; the user may then inspect the shape and location of the component and modify any one of its attributes interactively. Only mundane routine tasks, like the display of a building component, are completely automated in A2.

Generation

High-level tasks to be supported:

- support the generation of an initial spatial configuration of the required functional units of a building, possibly on several floors, based on an overall project specification received from the owner or project manager agent. The core of this functionality is provided by the existing SEED-Layout (SL) module. However, an intermediate step is needed in which the project specification is translated into a spatial program consisting of a *context and functional units* to be allocated in this context; this is the general input expected by SL. In SEED, this input is generated by another module, SEED-Pro[9], which was not available for use in the ACL project. But we can work around this problem in the context of ACL because the building projects sent to A2 consist largely of standard building types, e.g. one of the four common firestation types (one-company satellite or headquarter or two-company satellite or headquarter found on Army bases). In this case, the user can retrieve a standard program from the database and expand or otherwise modify this program from within SL, using the operations provided by its problem specification component.
- support the addition of structural components received from the structural agent to the spatial configuration; this involves a translation from the format in which A2 receives a description of these components into its own internal representation. The purpose is to check if these components interfere with the architectural configuration or contradict in other ways the architect's objectives.

---

7. Hemming, U. and Chien, S.F. (1995) "Schematic Layout Design in SEED Environment" *Journal of Architectural Engineering,* vol. 1 pp. 162-169
8. Flemming, U. and Woodbury, R. (1995) "Software Environment to Support Early Phases in Building Design (SEED): Overview" *Journal of Architectural Engineering,* vol. 1 pp. 147-152
9. Akin, Ö., Sen, R., Donia, M. and Zhang, Y. (1995) "SEED-Pro: Computer-Assisted Architectural Programming in SEED" *Journal of Architectural Engineering,* vol. 1 pp. 153-16

- support the generation of non-loadbearing components such as windows, doors, partitions, and exterior enclosures. This is a functionality not provided by SL; that is, this task demands the development of additional generative capabilities for A2.
- display all components, be they spatial or physical, as 3-D objects to allow for an inspection by the user.

Since SL supports only the first of these tasks, we developed supplemental agents to provide the other generation tasks. In this way, we can leave SL essentially unchanged. However, this internal division of labor will be invisible to the other ACL agents. All communication with the outside (that is, SEED-external) world will go through A2, which will be perceived by the other ACL agents and facilitator as a single, homogeneous agent (or black box) that executes all of the required architectural tasks. In the following, the term A2 will refer to the entire architectural agent as it is perceived by the outside; that is, it includes the capabilities provided by SL.

Communication

Tasks to be supported by A2 when communicating design information:

- broadcast design information generated by A2 (including information generated by SL; see the preceding section)
- receive design information generated and broadcast elsewhere
- display the information received in an intelligible form; this covers not only the graphical display of building components as specified above, but also the display of analysis and evaluation results, queries, requests etc. generated by other agents
- support negotiation between agents and the users of agents about changes in both requirements and design.

3 SCENARIOS

The following scenarios describe the interactions between the tasks identified in the preceding section from a high-level, user perspective. Note that in all of these scenarios, the individual agents are treated as operational units that include their users. The decisions and comments that the agents broadcast may have been generated internally manually by the user or automatically by some piece of software. These distinctions are invisible to the outside world in the ACL environment; that is, the agents appear to the outside world as a black boxes that are completely defined by the inputs they receive and outputs they generate (a view introduced in the preceding section for SL). We adopt this perspective for the development of the following scenarios because it allows us to specify the contributions of other agents without knowing how they work internally.

However, we indicate occasionally the logic that leads to a specific output, especially when it comes to the architectural agent. The use cases in the next section describe in greater detail how the user and the A2 software interact internally in the performance of tasks; that is, the section provides an insider's view of A2.

Cabin Example

The sponsor provided the ACL teams with the design of a simple, one-room cabin to be used as a first case to test the interactions between agents. The first scenario outlines a simple linear process without iterations and negotiations that reproduces the given cabin design. It is the base scenario from which no step can be removed if we want to generate a complete design. That is, a complete demonstration of A2 in its simplest form has to support at least the steps outlined below.

Step I

A2 receives general project information, like the type and size of the building to be designed (and displays it to its user in an appropriate format).

Step 2

A2 generates an appropriate problem specification from the project specification (using the capabilities provided by the problem specification component of SL). The specification contains

- a top functional unit of class *building* named 'cabin'
- a context.

The building has a single constituent of class *massing-elemenu* which has two constituents, a *storey* and a *roof}*® The functional requirements on these are acquired from the owner agent, retrieved from a database of functional units, or set by the user.

The context contains the basic context specification, like the size of the buildable area available (as broadcast by the owner agent).

10. See Flemming and Chien (1995) for a detailed description of the functional unit classes handled by SL.

(In an extended scenario allowing for iterations and negotiation, A2 may broadcast the generated program and may receive comments from the owner agent.)

Step 3

A2 generates a layout of the storey inside the massing element with the given dimensions (a trivial task that is fully supported by SL; but the 2D layout generated by SL has to be extruded before it is broadcast).

A2 broadcasts a description of the generated spatial configuration.

Step 4

A2 receives descriptions of structural components (generated by the structural agent), translates them into its internal representation, and displays them to the user.

Step 5

A2 adds non-load-bearing enclosure elements, partitions, doors, windows, and finishing elements, if required, to the design.

A2 broadcasts descriptions of these elements.

Step 6

A2 receives evaluations and analyses, and displays them in the appropriate formats.

Firestations on Army Bases

The real test and demonstration domain agreed upon by the ACL teams is the design of firestations on Army bases. From the perspective of the CMU team, this task does not require a new scenario because the capabilities of SL are generic, that is, not building-type specific: the same operators can be used to design a cabin or a fire station. The only difference is in the functional units that have to be allocated and in the requirements associated with these functional units.

Conflict Detection and Negotiation

Different agents at remote sites may be interested in the same objects, but may also disagree about some properties of the objects of common interest. A standard example in building design is the spatial conflict that occurs when a mechanical engineer routes pipes through regions occupied by structural elements placed by the architect or structural engineer. The mechanical engineer may not know about the existence of the interfering part, but the architect or structural engineer, when discovering the spatial conflict, may register disagreement with the mechanical engineer's decision, who - in turn - may disagree with the element's current placement. The involved parties then have to negotiate about ways to resolve the conflict, which may involve additional participants like the project manager or owner.

Such conflicts cannot be ruled out in the ACL project - indeed, must be expected - because of the distributed design representation and the limited or specialized views with which each agent approaches the task at hand. For example, conflicts can be expected based on independent decisions made by the architectural and structural agent. The negotiation protocol specified in part 3 to handle these situations will be illustrated by using the following two scenarios.

Scenario 1: Disagreement about the type of a structural member

Step 1

The structural agent (SA) generates a structural system for the cabin configuration broadcast by A2, which includes a roof structure made up of solid roof joists supported by a ridge beam at one end and the external walls at the other end.

SA broadcasts descriptions of the structural members.

Step 2

A2 receives the descriptions and displays them as 3D objects.

It broadcasts a disagreement with SA*s decision about the roof structure (because the A2 user, upon inspecting the roof system, decided that it looks boring) and requests trusses.

Step 3

SA sees no problem with this request. It broadcasts a message that the elements of the roof structure be retracted.

SA generates a new roof design consisting of trusses and broadcasts their description.

Step 4

A2 receives and agrees with the new design. (Internally, it replaces the old roof structure with the new one in its representation of the current design and stores it persistently. It may save the old design persistently in the database before-hand.)

Scenario 2: Disagreement about the placement of a structural member

Step 1

A2 generates an internal partition to divide the cabin into two distinct areas and broadcasts its description.

The A2 user discovers that the partition does not fit neatly underneath a truss, thus making its extension to the roof awkward. He broadcasts, through A2, a disagreement with the placement of the truss and requests a location on top of the partition.

Step 2

SA receives this proposal, but disagrees with it. It broadcasts its disagreement and a comment stating that the design and spacing of the trusses are optimal from a cost perspective.

The owner agent broadcasts agreement with SA's comment and requests compliance from A2.

Step 3

The user of A2 gives in and generates a new location for the partition underneath an existing truss. A2 broadcasts a message describing the change in the partition's location.

At this point, SA and the owner agent must update their internal design representations. They may or may not save the old version.

4 USE CASES

This section divides the overall functionality of A2 described in Section 1 into *use cases,* a concept introduced by Jacobson et.al. to engineer the phases of an object-oriented software development process.[1] Use cases describe the functionality of the software system under development in terms of the specific tasks the users of the system are able to be perform in interaction with the system. The description uses the terms of the application domain and is implementation-independent.

The term *user* refers in the following to a user of A2 performing tasks normally associated with architects. The specifications below treat A2 as a single-user system. This does not mean that multiple users cannot use it. It only means that distinctions in use cases that depend on which user is involved are not made; more importantly, it means that we do not develop use cases dealing specifically with the sharing of responsibilities between users playing different "roles", with the "ownership" of information, and with negotiations between responsibilities and ownerships. These more complex issues would overburden us at the present time.

Initialize Project

The A2 administrator establishes a first-time connection with the facilitator and broadcasts A2's interests and capabilities, or its inputs and outputs, respectively.

Select Filtered Object Information

The user identifies to A2 the class of objects she is interested in. A2 displays the description of the class attributes, and the user selects the attributes of interest.

Display Filtered Object Information

A2 informs the user if and to what extent it has received filtered object information and displays an overview of that information. The user may request the display of all or only of parts of this information. A2 displays the selected information in an appropriate form (to be determined on a case-by-case basis).

Receive Project Specification

The user selects project information (see use case "select filtered object information'*) and requests its display. A2 displays the information received (if any; see use case "display filtered object information").

1. **Jacobson, I., Christerson, M.f Jonsson, Pf and Overgaard, G. (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach.* New York: Addison-Wesley**

Generate Problem Specification

In interaction with SL, the user generates a problem specification for the type of project received through A2, starting with an initial specification that is retrieved from a database. SL provides interactive editing capabilities, which the user may use to modify the retrieved specification if it does not fit entirely the received project description.

When the user is satisfied, she requests from SL to send the specification to A2, which receives and broadcasts it.

Modify Problem Specification

The user receives a change request for the current problem specification (either through A2 or an independent communication, for example, by fax or telephone).

The user attempts to incorporates the change into the current problem specification. Simple modifications may be done directly in A2. For more elaborate ones, SL may have to be used, in which case A2 has to transmit the problem specification to SL and employ SL to change it. After the change, the modified specification is sent back to A2 and broadcast from there.

SL offers initially the possibility to make the following types of changes:

1. Add a functional unit
2. Delete a functional unit
3. Change a constituent relation; that is, a functional unit becomes the constituent of a different functional unit
4. Modify a requirement parameter
5. Add a requirement, including settings for the initial parameters
6. Delete a functional requirement
7. Modify a context parameter

The changes can be broadcast as modifications to an existing specification or as an alternative to an existing specification (see use case "modify a layout" below).

Generate a Layout

The problem specification generated with the help of SL or received back from A2 after some negotiations allows SL to support actively the generation of a layout that satisfies the requirements in the specification and reflects other concerns of the user (architect) not explicitly stated (that is, their management rests solely wit the user). The use cases developed for SL describe the range of supporting capabilities offered by SL (see the SEED-Layout Requirements Specification[2]).

After the user is satisfied with a layout, he asks SL to send the layout to A2, which displays the components three-dimensionally on its screen and broadcasts the layout.

---

2. Flemming, U., Coyne, R., and Chien, S.F. (1994) *SEED-Layout Requirements Analysis,* http://seed.edrc.cmu.edu/SL/SL-start.book.html

Modify a Layout

When A2 (or the user) receives a request for a layout modification, this layout has to be send back to SL, except in trivial cases that allow for modifications directly in A2. Modifications that are possible right now range from simple ones like changing the dimensions of a design unit or changing its location to more complex ones requiring, for example, changing the entire organizational scheme of a floor, if not an entire building. SL offers a broad range of generation capabilities to realize these changes.

After the user is satisfied, she requests from SL to send the modified layout to A2, and A2 broadcasts it.

NOTE: The user may initiate this broadcast in two basic forms: (1) as a modification of an existing layout, which may nevertheless include new objects; or (2) as a new solution alternative. The essential difference between these two options is that in case 1, objects that existed before will be broadcast under their old identifiers, while in case 2, new identifiers will be created and broadcast for all objects in the layout.

Receive Structural Component Descriptions

The user selects structural component information using the new object filter (see use case "select filtered object information")* A2 displays the new components received (see use case "display filtered object information")

Modify a Structural Component

The user modifies interactively the location or shape of a structural component. When the user commits to the change, A2 broadcasts the change. This may generate a conflict and subsequent negotiations.

Generate All Physical Non-Load-Bearing Components of a Specific Type

The user requests A2 to generate all instances of a specific physical component that are needed, e. g. all doors. A2 adheres to the following script in general (see the following specific use cases for type-specific details):

After the user's request has been received, A2 displays the currently available technologies or predefined elements for the selected component type. The user selects one of these.

Guided by the selected technology, A2 generates all needed instances of the requested component type, using reasoning that is specific to the type and explained below, and displays them in a preview display. A2 maintains all relations between components that are important for subsequent queries or editing; for example, it records internally in which wall each door has been placed so that, when the user later moves that door to the right and left, it will always stay within the surfaces of the wall to which it belongs.

This sequence can be executed in two distinct ways:

1. All instances are created automatically ("place all'*)
2. A2 first shows a location for a new instance and places an element there only if this is explicitly requested by the user ("place and find next")

When the user commits to the generated components, possibly after some modifications (see the change use cases below), A2 stores their descriptions persistently and broadcasts them.

Generate All Internal Partitions

A2 generates all internal partitions of a selected type that enclose all rooms and avoid structural elements like columns.

Generate All External Enclosures

A2 generates all external walls or enclosures of a selected type that are non-structural.

Generate All Internal Doors

A2 generates all internal doors of a selected type between rooms with required adjacencies or public access requirements. It computes the width from the functional type of the rooms and selects a specific location based on rules-of-thumb, like "if space permits, leave at least one foot on the hinge side of the door".

Generate All Windows

A2 generates all windows of the selected type for rooms with an explicit natural lighting requirement or for rooms of a type that implies such a requirement. It computes the window size from the room area.

Generate All Floor Finishes

A2 generates all floor finishes of selected types for all rooms.

Generate All Wall Finishes

A2 generates all wall finishes of selected types for all rooms.

Generate All Ceiling Finishes

A2 generates all ceiling finishes of selected types for all rooms.

Generate All Roof Coverings

A2 generates all roof coverings of a selected type.

Generate a Door

The user requests A2 to generate a door in a selected enclosure or partition.

A2 displays the currently available door types depending on whether an exterior or interior door is required. The user selects a type.

A2 generates a door of the selected type and places it in the selected wall. A2 displays the door.

When the user commits to the door, A2 stores its description persistently and broadcasts its description.

Generate a Window

The user requests A2 to generate a window in a selected enclosure.

A2 displays the currently available window types. The user selects a type.

A2 generates a window of the selected type and places it in the selected wall. A2 displays the window.

When the user commits to the window, A2 stores its description persistently and broadcasts its description.

Change the Location of a Component

The user selects a component and moves it interactively to a different location without changing its shape.

When the user commits to the change, A2 updates the description of the component and broadcasts this description.

Change the Shape of a Component

The user selects a component and changes its shape in the preview display using 'pulling' or 'pushing[9] operations on corners, edges, or faces.

When the user commits to the change, A2 updates the persistent description of the component and broadcasts this description.

Change a Finish

The user requests changing the finish of a selected component.

A2 shows the available finishes, and the user selects one. A2 updates the display of the finish; this will involve at least using a color appropriate to the finish material, if not some form of texture mapping, when this is appropriate.

When the user commits to the change, A2 updates the persistent description of the component's finish (probably an attribute value) and broadcasts this description.

Receive Energy Analysis

This use case works analogously to the use case "receive structural component description."

5 CONCEPT MAPPING BASED ON THE OBJECT MODELING LANGUAGE OML

Concept Mapping

A basic issue in developing collaborative engineering systems is the representation of the product and process information that must be communicated between agents. This product information should include the geometry of the physical parts of the product and their relationships, non-geometric information (e.g., details on functionality of the parts, constraints, and design intent), and multiple levels of abstraction. The software used in the ACL project by the participating agents is so diverse that for pragmatic reasons, a common design representation shared by all agents could not be found. The group as a whole decided therefore that the individual agents in the ACL project maintain their own internal representations and generate from these the views most appropriate to the tasks they execute. That is, the ACL project takes a de-centralized approach to agent coupling (and, by implication, product modeling). This also eliminates the possibility of having a central agent controlling the *scheduling* of agent activities.

Some of the participating agents 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. It is the responsibility of the facilitator to translate between the different representations in a way that maintains consistency with the other agents' representations.

The scenarios in Section 3 illustrate concretely some obvious and important implications of choosing a distributed product model. The communication between agents is primarily about objects representing building elements or components, to which comments in the broadest sense of the term are added during negotiations. A component in which several agents are interested will have an independent representation in each agent's internal design representation. In fact, the information clustered around one component in one agent's representation may be distributed over several components in the representation of another agent. As a result, objects are not communicated directly or 'shared'; rather, the agents exchange *descriptions* of objects, which we were careful to demonstrate in the scenarios. In this situation, it is of utmost **importance that *each agent keeps its internal representation of the current design components up-to-date,*** based on the descriptions it receives about newly created objects or about changes made to existing objects by other agents.

It is likely that an agent may create additional versions of a design component that differ from the version that it broadcasts to other agents. An agent is free to save (i.e. store persistently) different versions of the objects of interest, which may or may not originate in the agent itself, depending on how it structures and manages its design tasks. But we strongly suggest that each agent store persistently at least all versions of the objects *it* creates and broadcasts so that it will be able to answer queries about prior design states; this may become important, for example, if one wants to unravel the reasons behind certain conflicts and how they were resolved. A related issue is who decides when a design is finished and which versions of the objects it contains are part of the final design. Again, this is not automatically clear in a distributed product model that is updated in unpredictable ways by the asynchronous decisions of distributed agents. The nego-

tiation protocol introduced in part 3 is designed to be able to handle this issue, (the ACL team decided as a whole not to address this issue initially).

The lack of a common design representation that could be shared by all agents in the ACL project implies that *the system as a whole needs a mapping mechanism.* Concepts and objects in different local representations must be mapped, or even re-synthesized, when they are communicated among agents. On the other hand, not every agent has access to the whole design; in fact, an agent sees only those portions of a design that it understands and is interested in.

Concept mapping and object re-synthesis may require attribute transformations, which - in turn - may require attribute extraction from related concepts other than the concept being mapped. We illustrate this by a simple example. Figure 2 and Figure 3 show two different wall concepts, one of which may be used by an architectural agent and the other one by a structural design agent. The figure uses the widely used OMT notation for the specification of objects models.[1]

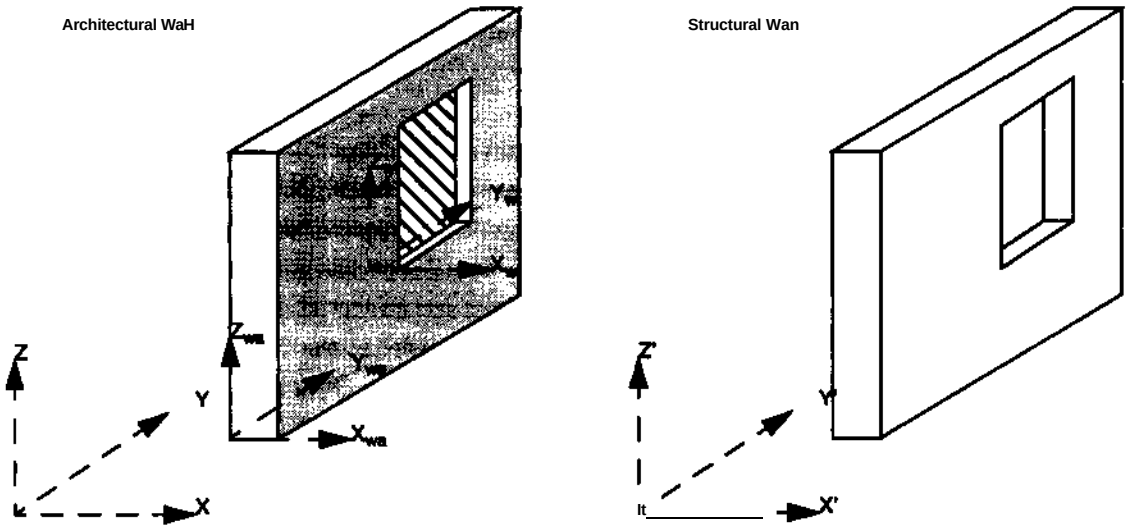


Figure 2. Architectural Wall and Structural Wall

1. **Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F, & Lorensen, W. (1991). Object-Oriented Modeling and Design. Englewood Cliffs: Prentice-Hall.**

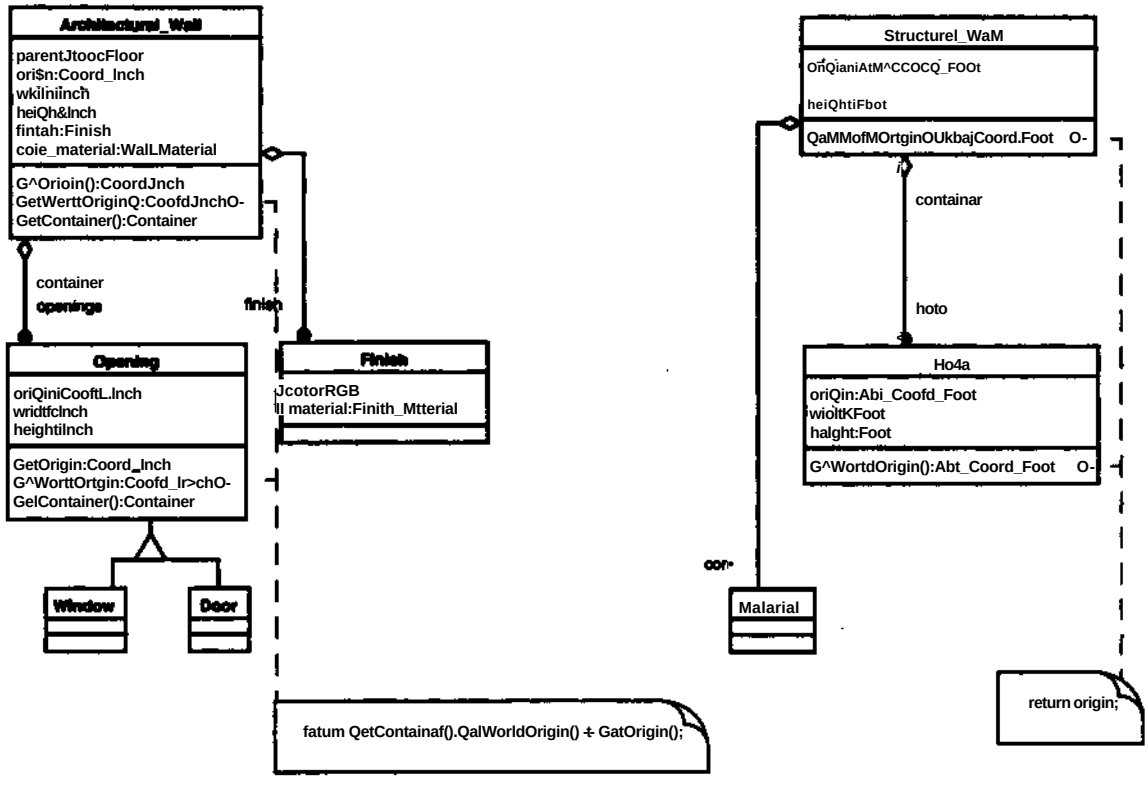


Figure 3. OMT Model of Architectural Wall and Structural Wall

In the architectural wall and window objects, each object is defined in its own coordinate system, where the window coordinate system is relative to the wall coordinate system. The coordinate system of the architectural wall objects is, in turn, relative to a single world-coordinate system. In contrast, the structural objects are defined in one single world coordinate system, where all coordinates are relative to a single origin.

A quick glance at Figure 3 reveals that the mapping from Architectural_Wall to Structural_Wall requires

1. coordinate transformations for origin, width and height
2. synthesis of object StructuraLWall::core based on the attributes of Architectural_Wall::core_material
3. synthesis of object StructuraLWall::hole based on the attributes of ArchitecturaLWall::openings.

The mapping from Structural_Wall to Architectural_Wall requires

1. coordinate transformations for origins, width and height
2. synthesis of object Architectural_Wall::core_material based on the attributes of StructuraLWall::core.

Note that this example is by no means completely worked out; it is only intended to illustrate the notion of concept mapping. The complexity of concept mapping increases significantly when the mapping

process requires recursive concept mappings. The knowledge required for correct concept mappings has to come from the developers who design and implement the concepts.

The Object Modeling Language OML

The facilitator is responsible for concept mapping in the ACL project; thus, communication between top level ACL agents must go through the facilitator before the information communicated reaches the destination agents. To facilitate the generation of appropriate translators, the CMU team developed the Object Modeling Language OML, which the teams utilized to express and communicate the concepts relevant to the task for which they were responsible.[2] A local design representation of an ACL agent can then be mapped into OML objects (see below) and state changes to these OML objects forwarded to the facilitator via a message server of the kind described in the next section; see Figure 4.

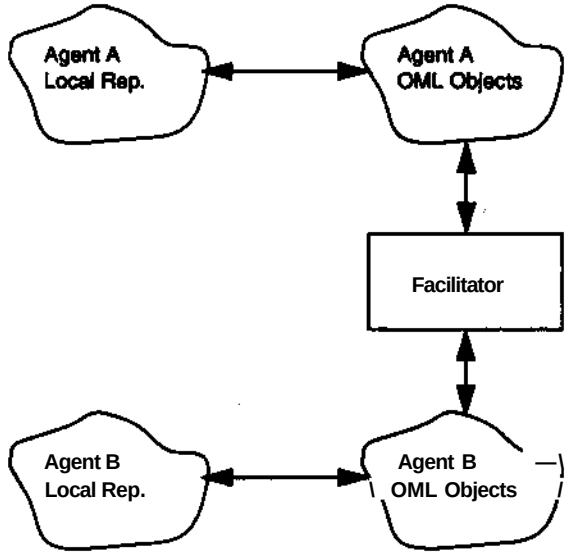


Figure 4. OML-Based Design Representation Mapping

For closely related agents, it may be possible to develop an OML schema that is shared by these agents, which would make the concept mapping mechanism provided by the facilitator unnecessary (see Figure 5).

2. **Snyder, J. and Flemming, U. (1994) *ACL Object Model Specification Language.* Technical Report at http://seed.edrc.cmu.edu/ACL/ObjModelAPI/objapi.book.html**

Figure 5. OML-Based Communication Without Mapping

OML is schema-based and independent of programming languages. It is designed to provide semantic consistency for inter-agent information exchange. Based on an agent's internal representation, an OML schema builds an OML model external to the agent, which describes the agent's model to the outside world. Given such an OML schema, each agent developer provides a language binding specification that is used to map the local representation to and from the OML schema.

The mapping between different local representations involves a large amount of monotonous binding assignments, which - in addition - must be verified. This verification is error-prone and time-consuming, which is especially true when the mapping routines need several programming language linkages, e.g. C, C++, Fortran, or SQL. The necessary code cannot be reliably generated by hand. This creates the need for the automatic generation of language binding code.

By using a language binding compiler provided in the OML standard distribution, the developers can use the OML schema and the language binding specification to generate automatically language binding code in a host language like C++. This code can be compiled into a *language binding library* and linked to an agent's domain code. The standard OML distribution also includes an OML *run-time library*, which delivers packaging and un-packaging of OML objects along with the entire OML communication functionality to the domain application. This OML run-time library should also be linked into the domain application (see Figure 6).

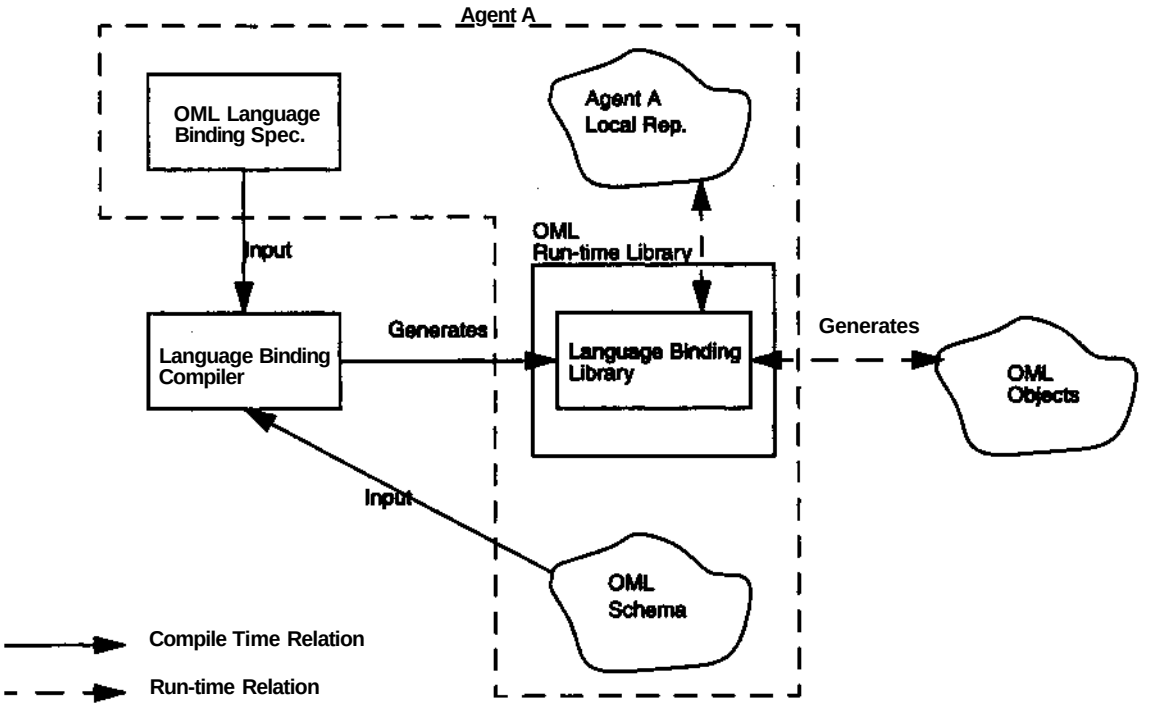


Figure 6. OML-Based Schema Mapping and Language Binding

6 CURRENT IMPLEMENTATION

Overview

Agents in a distributed environment may be structured internally in a hierarchical fashion. But how other "peer" agents are subdivided should be of no concern for an agent. When we use the term "other agents", we refer to the agents in the same agent communication scope. An *agent communication scope* is composed of a parent agent and all its direct child-agents, see Figure 7 (note that a parent agent may only be a conceptual agent representing the entire configuration, but not exist physically). The black box view of an agent introduced above hides its internal structure and the subagents it comprises. For reasons outlined in section 2, we conceived of A2 as an agent composed of subagents as shown in Figure 8.

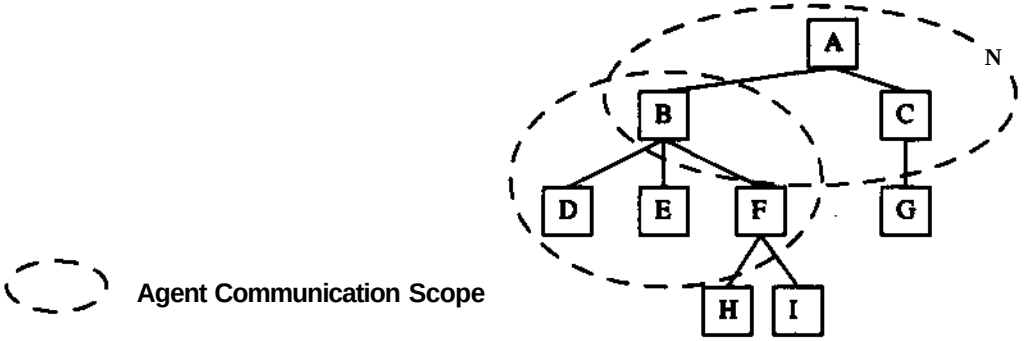


Figure 7. Agent Communication Scope

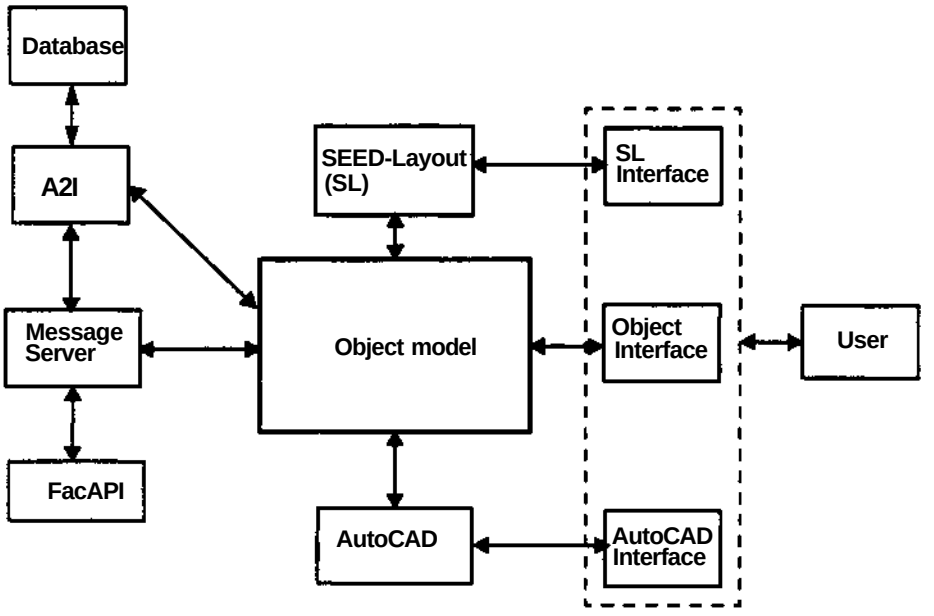


Figure 8. Current A2 Architecture

The central coordinating component is a *message server* able to receive messages from and send them to the facilitator, read from and write into the database, and send and receive object descriptions to or from an object model expressed in OML. This object model consists internally of three parts handling data translations to and from, respectively, SL, the object interface, and AutoCAD.

The role of SL is explained in section 2. AutoCAD is used not as a traditional 'CAD engine' to derive a geometric description of design elements, but solely for 3-dimensional display purposes. The AutoCAD object model provides the generative capabilities for physical, 3-dimensional design components not provided by SL. This part should be replaced by a full-fledged geometric modeler in a next version of A2.

Message Server

The message server is a transaction-based routing server that implements a simple communication protocol. Clients use the communication protocol to request services provided by the message server. Currently, these services consist of

- registering interest,
- broadcasting message transactions, and
- directing message transactions.

A *transaction* is a sequence of OML message frames bound by a special control message in the communication protocol. A transaction is treated as one entity: it is never partially committed or abandoned. The communication between a client program and the message server is completely encapsulated by the OML library, which lessens the burdens on client programming. In fact, the client programs need not be aware of the existence of the communication protocol at all. The overall architecture of the message server is illustrated in Figure 9.

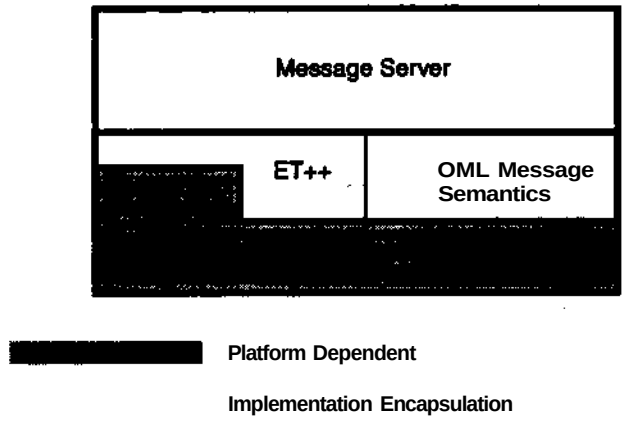


Figure 9. Message Server Architecture Block Diagram

As the figure shows, the window and operating systems are encapsulated by ET++, a C++ application framework.[3] The OML Message Semantics is a part of the OML library that implements the commu-

NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

A2: An Architectural Agent in a Collaborative Engineering Environment

Ulrich Flemming, Zeyno Aygen, James Snyder and Jonah Tsai

EDRC 48-38-96

A2: An Architectural Agent in a Collaborative Engineering Environment

Ulrich Flemming
Zeyno Aygen
James Snyder
Jonah Tsai

November 1996

ABSTRACT

This report describes the functionality of the Architectural Agent (A2) developed by a team at CMU as part of the US ACERL-sponsored ACL (Investigation of an Agent Communication Language) project, which addresses communication issues in a distributed computer-supported building design environment The core functionality of A2 consists of generating schematic layouts of the functional units in a spatial program; this functionality is provided by using SEED-Layout (SL) as a server; SL is a module in the Software Environment to Support the Early Phases in Building Design (SEED) currently under development at the EDRC. A2 is in addition served by the object database UniSQL and AutoCAD, used strictly for display purposes. These diverse components communicate through a message server with each other and a facilitator API provided by another team. The communication protocol and data translations are based on an active object model expressed in the specification language OML developed by the CMU team explicitly for the ACL project The CMU team also suggests an approach toward conflict management and negotiation for an environment like ACL and describes an OML-based protocol that implements the approach. It offers general conclusions about relevant aspects of the ACL project.

ACKNOWLEDGMENTS

The work described here was sponsored by the US Army Corps of Engineers Construction Engineering Research Laboratory (USACERL) under contract DACA-88-93-D-0004.

1 INTRODUCTION

This report describes the contribution of a team at CMU to the USACERL-sponsored ACL (Investigation of an Agent Communication Language) project. The goal of the ACL project was to investigate the use of concurrent engineering (CE) technologies to support collaboration between participants in the facility design and delivery process. The project recognized that the lack of a standardized representational syntax and semantics is a major barrier to implementing CE techniques in this application domain. It intended to determine fundamental functional requirements of a technology that would allow intelligent agents to cooperate in facility design.[1]

In this context, the term *agent* refers to a software configuration at a specific site that

- supports a specific design task, which is not necessarily automated; that is, it may involve human interaction and intervention,
- generates design information in its own internal representation, and
- is able to communicate its design decisions to other agents at different sites.

In order to secure the cooperation of a sufficient number of agents in the project, USACERL organized a team of researchers representing four major universities: the Massachusetts Institute of Technology (MIT), Stanford University (Stanford), the University of Illinois at Urbana-Champaign (UIUC), and Carnegie Mellon University (CMU). The collaboration between the teams was to make use of technologies developed independently by the four universities and USACERL, which included

- the Agent Collaboration Environment (ACE) developed by USACERL,[2] which in turn served as front-end for the energy analysis system BLAST developed by UIUC[3] and two additional agents assuming the tasks, respectively, of the building owner and project manager,
- the structural design agent developed by MIT,[4] and
- the architectural layout system SEED-Layout developed at CMU,[5] which was to execute all architectural tasks in the ACL project.

The sponsor had established at the outset that the cooperation of these systems was to be based on the agent-based Federation Architecture developed by Stanford, which intends to allow independent software agents to exchange information by communicating with *facilitators* through appropriately designed APIs. The facilitators are responsible for message brokering, based on interests registered with a facilitator by the individual agents, and data translations, based on production-type translation rules.[6] Figure 1 depicts the top-level agent connections in the ACL project.

---

1. **The goal statements are taken almost verbatim from Contract DAC88-93-D-0004(010) between the US Army Corps of Engineers Construction Engineering Research Lab and CMU.**
2. **McGraw, K. (1996) *Agent Collaboration Environment (ACE),* http://www.cecer.army.mil/pl/ace/.**
3. **BLAST User's Reference Vol. I and II (1991) BLAST Support Office, Dept. of Mechanical and Industrial Engineering, Univ. of Illinois at Urbana-Champaign**
4. **Chiou, J. and Logcher, R.D.(1996) Testing a Federation Architecture in Collaborative Design Process. Res. Report R96-01. Dept. of Civil and Environmental Engineering, MIT, Cambridge, MA**
5. **Flemming, U. and Chien, S.F (1995) "Schematic Layout Design in SEED Environment" *Journal of Architectural Engineering,* vol. 1 162-169**

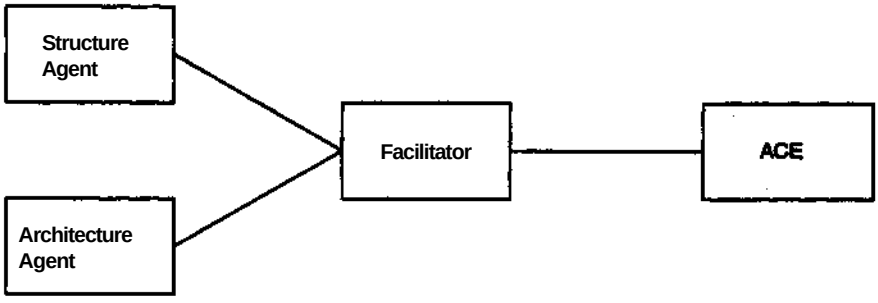


Figure 1. Top-Level ACL Agent Connections

The present report describes the contributions of the CMU team to the ACL project and summarizes its experience with the federated, facilitator-based ACL architecture. The sections of this report fall into three major parts that are by and large self-contained:

Agent Functionality

The first three sections describe the capabilities of the Architectural Agent (A2) developed by the CMU team in domain terms. Section 2 specifies the overall functionality of the agent in terms of the broad tasks it is able to support. Section 3 describes several scenarios meant to convey a more concrete understanding of the tasks and how they interact with tasks performed by other ACL agents in the overall design process as supported by the ACL environment. Section 4 identifies the 'use cases' as they emerge from these scenarios.

Agent Implementation

Section 5 describes data communication issues in the ACL project and introduces the approach taken by the ACL team to handle them, which is based on the Object Modeling Language OML developed by the CMU team specifically for the ACL project. Section 6 describes the current implementation of A2 as a configuration of independently developed subagents that communicate via an OML-based object model.

Conflict Management and Negotiation in the ACL project

Section 8 introduces an approach toward negotiation and conflict management that is suitable for a project like ACL. Section 9 specifies an OML-based negotiation protocol implementing the approach and illustrates the use of this protocol through two examples.

A concluding section summarizes the experience of the CMU team in the ACL project, especially with respect to the Federation Architecture.

---

6. T. Khedro, M. Case, U. Flemming, M. Genesereth, R. Lpgcher, C. Pedersen, J. Snyder, R. Sriram, and P. Teicholz (1995) "Development of Multi-Institutional Testbed for Collaborative Facility Engineering Infrastructure" *Proc. Second Congress on Computing in Civil Engineering* (Atlanta, GA, June 1995) J.P. Mohsen, ed., American Society of Civil Engineers, New York, NY, 1308-1315

2 OVERALL FUNCTIONALITY OF THE ARCHITECTURAL AGENT

This section specifies the overall functionality of the agent developed by the CMU team within the ACL project. We refer to this agent as the *architectural agent* (A2 for short) because its tasks cover the traditional responsibilities of architects during the preliminary building design phase. The core of the capabilities demanded from A2 is to be provided by SEED-Layout[7], one of three modules currently under development at the EDRC to establish SEED, a Software Environment to Support the Early Phases in Building Design[8].

The functionality of A2 consists of support for the tasks listed below, where *support' does not necessarily mean that A2 is able or expected to execute a task automatically. Tasks that involve complex evaluations and judgment are principally left to the user; A2 only has to record and store the user's decisions accurately. Other tasks may be executed automatically, interactively, or through a combination of these two 'generation modes[9]. For example, A2 may generate a building component like a door automatically and place it based on rules of thumb; the user may then inspect the shape and location of the component and modify any one of its attributes interactively. Only mundane routine tasks, like the display of a building component, are completely automated in A2.

Generation

High-level tasks to be supported:

- support the generation of an initial spatial configuration of the required functional units of a building, possibly on several floors, based on an overall project specification received from the owner or project manager agent. The core of this functionality is provided by the existing SEED-Layout (SL) module. However, an intermediate step is needed in which the project specification is translated into a spatial program consisting of a *context and functional units* to be allocated in this context; this is the general input expected by SL. In SEED, this input is generated by another module, SEED-Pro[9], which was not available for use in the ACL project. But we can work around this problem in the context of ACL because the building projects sent to A2 consist largely of standard building types, e.g. one of the four common firestation types (one-company satellite or headquarter or two-company satellite or headquarter found on Army bases). In this case, the user can retrieve a standard program from the database and expand or otherwise modify this program from within SL, using the operations provided by its problem specification component.
- support the addition of structural components received from the structural agent to the spatial configuration; this involves a translation from the format in which A2 receives a description of these components into its own internal representation. The purpose is to check if these components interfere with the architectural configuration or contradict in other ways the architect's objectives.

---

7. Hemming, U. and Chien, S.F. (1995) "Schematic Layout Design in SEED Environment" *Journal of Architectural Engineering,* vol. 1 pp. 162-169
8. Flemming, U. and Woodbury, R. (1995) "Software Environment to Support Early Phases in Building Design (SEED): Overview" *Journal of Architectural Engineering,* vol. 1 pp. 147-152
9. Akin, Ö., Sen, R., Donia, M. and Zhang, Y. (1995) "SEED-Pro: Computer-Assisted Architectural Programming in SEED" *Journal of Architectural Engineering,* vol. 1 pp. 153-16

- support the generation of non-loadbearing components such as windows, doors, partitions, and exterior enclosures. This is a functionality not provided by SL; that is, this task demands the development of additional generative capabilities for A2.
- display all components, be they spatial or physical, as 3-D objects to allow for an inspection by the user.

Since SL supports only the first of these tasks, we developed supplemental agents to provide the other generation tasks. In this way, we can leave SL essentially unchanged. However, this internal division of labor will be invisible to the other ACL agents. All communication with the outside (that is, SEED-external) world will go through A2, which will be perceived by the other ACL agents and facilitator as a single, homogeneous agent (or black box) that executes all of the required architectural tasks. In the following, the term A2 will refer to the entire architectural agent as it is perceived by the outside; that is, it includes the capabilities provided by SL.

Communication

Tasks to be supported by A2 when communicating design information:

- broadcast design information generated by A2 (including information generated by SL; see the preceding section)
- receive design information generated and broadcast elsewhere
- display the information received in an intelligible form; this covers not only the graphical display of building components as specified above, but also the display of analysis and evaluation results, queries, requests etc. generated by other agents
- support negotiation between agents and the users of agents about changes in both requirements and design.

3 SCENARIOS

The following scenarios describe the interactions between the tasks identified in the preceding section from a high-level, user perspective. Note that in all of these scenarios, the individual agents are treated as operational units that include their users. The decisions and comments that the agents broadcast may have been generated internally manually by the user or automatically by some piece of software. These distinctions are invisible to the outside world in the ACL environment; that is, the agents appear to the outside world as a black boxes that are completely defined by the inputs they receive and outputs they generate (a view introduced in the preceding section for SL). We adopt this perspective for the development of the following scenarios because it allows us to specify the contributions of other agents without knowing how they work internally.

However, we indicate occasionally the logic that leads to a specific output, especially when it comes to the architectural agent. The use cases in the next section describe in greater detail how the user and the A2 software interact internally in the performance of tasks; that is, the section provides an insider's view of A2.

Cabin Example

The sponsor provided the ACL teams with the design of a simple, one-room cabin to be used as a first case to test the interactions between agents. The first scenario outlines a simple linear process without iterations and negotiations that reproduces the given cabin design. It is the base scenario from which no step can be removed if we want to generate a complete design. That is, a complete demonstration of A2 in its simplest form has to support at least the steps outlined below.

Step I

A2 receives general project information, like the type and size of the building to be designed (and displays it to its user in an appropriate format).

Step 2

A2 generates an appropriate problem specification from the project specification (using the capabilities provided by the problem specification component of SL). The specification contains

- a top functional unit of class *building* named 'cabin'
- a context.

The building has a single constituent of class *massing-elemenu* which has two constituents, a *storey* and a *roof}*® The functional requirements on these are acquired from the owner agent, retrieved from a database of functional units, or set by the user.

The context contains the basic context specification, like the size of the buildable area available (as broadcast by the owner agent).

10. See Flemming and Chien (1995) for a detailed description of the functional unit classes handled by SL.

(In an extended scenario allowing for iterations and negotiation, A2 may broadcast the generated program and may receive comments from the owner agent.)

Step 3

A2 generates a layout of the storey inside the massing element with the given dimensions (a trivial task that is fully supported by SL; but the 2D layout generated by SL has to be extruded before it is broadcast).

A2 broadcasts a description of the generated spatial configuration.

Step 4

A2 receives descriptions of structural components (generated by the structural agent), translates them into its internal representation, and displays them to the user.

Step 5

A2 adds non-load-bearing enclosure elements, partitions, doors, windows, and finishing elements, if required, to the design.

A2 broadcasts descriptions of these elements.

Step 6

A2 receives evaluations and analyses, and displays them in the appropriate formats.

Firestations on Army Bases

The real test and demonstration domain agreed upon by the ACL teams is the design of firestations on Army bases. From the perspective of the CMU team, this task does not require a new scenario because the capabilities of SL are generic, that is, not building-type specific: the same operators can be used to design a cabin or a fire station. The only difference is in the functional units that have to be allocated and in the requirements associated with these functional units.

Conflict Detection and Negotiation

Different agents at remote sites may be interested in the same objects, but may also disagree about some properties of the objects of common interest. A standard example in building design is the spatial conflict that occurs when a mechanical engineer routes pipes through regions occupied by structural elements placed by the architect or structural engineer. The mechanical engineer may not know about the existence of the interfering part, but the architect or structural engineer, when discovering the spatial conflict, may register disagreement with the mechanical engineer's decision, who - in turn - may disagree with the element's current placement. The involved parties then have to negotiate about ways to resolve the conflict, which may involve additional participants like the project manager or owner.

Such conflicts cannot be ruled out in the ACL project - indeed, must be expected - because of the distributed design representation and the limited or specialized views with which each agent approaches the task at hand. For example, conflicts can be expected based on independent decisions made by the architectural and structural agent. The negotiation protocol specified in part 3 to handle these situations will be illustrated by using the following two scenarios.

Scenario 1: Disagreement about the type of a structural member

Step 1

The structural agent (SA) generates a structural system for the cabin configuration broadcast by A2, which includes a roof structure made up of solid roof joists supported by a ridge beam at one end and the external walls at the other end.

SA broadcasts descriptions of the structural members.

Step 2

A2 receives the descriptions and displays them as 3D objects.

It broadcasts a disagreement with SA*s decision about the roof structure (because the A2 user, upon inspecting the roof system, decided that it looks boring) and requests trusses.

Step 3

SA sees no problem with this request. It broadcasts a message that the elements of the roof structure be retracted.

SA generates a new roof design consisting of trusses and broadcasts their description.

Step 4

A2 receives and agrees with the new design. (Internally, it replaces the old roof structure with the new one in its representation of the current design and stores it persistently. It may save the old design persistently in the database before-hand.)

Scenario 2: Disagreement about the placement of a structural member

Step 1

A2 generates an internal partition to divide the cabin into two distinct areas and broadcasts its description.

The A2 user discovers that the partition does not fit neatly underneath a truss, thus making its extension to the roof awkward. He broadcasts, through A2, a disagreement with the placement of the truss and requests a location on top of the partition.

Step 2

SA receives this proposal, but disagrees with it. It broadcasts its disagreement and a comment stating that the design and spacing of the trusses are optimal from a cost perspective.

The owner agent broadcasts agreement with SA's comment and requests compliance from A2.

Step 3

The user of A2 gives in and generates a new location for the partition underneath an existing truss. A2 broadcasts a message describing the change in the partition's location.

At this point, SA and the owner agent must update their internal design representations. They may or may not save the old version.

4 USE CASES

This section divides the overall functionality of A2 described in Section 1 into *use cases,* a concept introduced by Jacobson et.al. to engineer the phases of an object-oriented software development process.[1] Use cases describe the functionality of the software system under development in terms of the specific tasks the users of the system are able to be perform in interaction with the system. The description uses the terms of the application domain and is implementation-independent.

The term *user* refers in the following to a user of A2 performing tasks normally associated with architects. The specifications below treat A2 as a single-user system. This does not mean that multiple users cannot use it. It only means that distinctions in use cases that depend on which user is involved are not made; more importantly, it means that we do not develop use cases dealing specifically with the sharing of responsibilities between users playing different "roles", with the "ownership" of information, and with negotiations between responsibilities and ownerships. These more complex issues would overburden us at the present time.

Initialize Project

The A2 administrator establishes a first-time connection with the facilitator and broadcasts A2's interests and capabilities, or its inputs and outputs, respectively.

Select Filtered Object Information

The user identifies to A2 the class of objects she is interested in. A2 displays the description of the class attributes, and the user selects the attributes of interest.

Display Filtered Object Information

A2 informs the user if and to what extent it has received filtered object information and displays an overview of that information. The user may request the display of all or only of parts of this information. A2 displays the selected information in an appropriate form (to be determined on a case-by-case basis).

Receive Project Specification

The user selects project information (see use case "select filtered object information'*) and requests its display. A2 displays the information received (if any; see use case "display filtered object information").

1. **Jacobson, I., Christerson, M.f Jonsson, Pf and Overgaard, G. (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach.* New York: Addison-Wesley**

Generate Problem Specification

In interaction with SL, the user generates a problem specification for the type of project received through A2, starting with an initial specification that is retrieved from a database. SL provides interactive editing capabilities, which the user may use to modify the retrieved specification if it does not fit entirely the received project description.

When the user is satisfied, she requests from SL to send the specification to A2, which receives and broadcasts it.

Modify Problem Specification

The user receives a change request for the current problem specification (either through A2 or an independent communication, for example, by fax or telephone).

The user attempts to incorporates the change into the current problem specification. Simple modifications may be done directly in A2. For more elaborate ones, SL may have to be used, in which case A2 has to transmit the problem specification to SL and employ SL to change it. After the change, the modified specification is sent back to A2 and broadcast from there.

SL offers initially the possibility to make the following types of changes:

1. Add a functional unit
2. Delete a functional unit
3. Change a constituent relation; that is, a functional unit becomes the constituent of a different functional unit
4. Modify a requirement parameter
5. Add a requirement, including settings for the initial parameters
6. Delete a functional requirement
7. Modify a context parameter

The changes can be broadcast as modifications to an existing specification or as an alternative to an existing specification (see use case "modify a layout" below).

Generate a Layout

The problem specification generated with the help of SL or received back from A2 after some negotiations allows SL to support actively the generation of a layout that satisfies the requirements in the specification and reflects other concerns of the user (architect) not explicitly stated (that is, their management rests solely wit the user). The use cases developed for SL describe the range of supporting capabilities offered by SL (see the SEED-Layout Requirements Specification[2]).

After the user is satisfied with a layout, he asks SL to send the layout to A2, which displays the components three-dimensionally on its screen and broadcasts the layout.

---

2. Flemming, U., Coyne, R., and Chien, S.F. (1994) *SEED-Layout Requirements Analysis,* http://seed.edrc.cmu.edu/SL/SL-start.book.html

Modify a Layout

When A2 (or the user) receives a request for a layout modification, this layout has to be send back to SL, except in trivial cases that allow for modifications directly in A2. Modifications that are possible right now range from simple ones like changing the dimensions of a design unit or changing its location to more complex ones requiring, for example, changing the entire organizational scheme of a floor, if not an entire building. SL offers a broad range of generation capabilities to realize these changes.

After the user is satisfied, she requests from SL to send the modified layout to A2, and A2 broadcasts it.

NOTE: The user may initiate this broadcast in two basic forms: (1) as a modification of an existing layout, which may nevertheless include new objects; or (2) as a new solution alternative. The essential difference between these two options is that in case 1, objects that existed before will be broadcast under their old identifiers, while in case 2, new identifiers will be created and broadcast for all objects in the layout.

Receive Structural Component Descriptions

The user selects structural component information using the new object filter (see use case "select filtered object information")* A2 displays the new components received (see use case "display filtered object information")

Modify a Structural Component

The user modifies interactively the location or shape of a structural component. When the user commits to the change, A2 broadcasts the change. This may generate a conflict and subsequent negotiations.

Generate All Physical Non-Load-Bearing Components of a Specific Type

The user requests A2 to generate all instances of a specific physical component that are needed, e. g. all doors. A2 adheres to the following script in general (see the following specific use cases for type-specific details):

After the user's request has been received, A2 displays the currently available technologies or predefined elements for the selected component type. The user selects one of these.

Guided by the selected technology, A2 generates all needed instances of the requested component type, using reasoning that is specific to the type and explained below, and displays them in a preview display. A2 maintains all relations between components that are important for subsequent queries or editing; for example, it records internally in which wall each door has been placed so that, when the user later moves that door to the right and left, it will always stay within the surfaces of the wall to which it belongs.

This sequence can be executed in two distinct ways:

1. All instances are created automatically ("place all'*)
2. A2 first shows a location for a new instance and places an element there only if this is explicitly requested by the user ("place and find next")

When the user commits to the generated components, possibly after some modifications (see the change use cases below), A2 stores their descriptions persistently and broadcasts them.

Generate All Internal Partitions

A2 generates all internal partitions of a selected type that enclose all rooms and avoid structural elements like columns.

Generate All External Enclosures

A2 generates all external walls or enclosures of a selected type that are non-structural.

Generate All Internal Doors

A2 generates all internal doors of a selected type between rooms with required adjacencies or public access requirements. It computes the width from the functional type of the rooms and selects a specific location based on rules-of-thumb, like "if space permits, leave at least one foot on the hinge side of the door".

Generate All Windows

A2 generates all windows of the selected type for rooms with an explicit natural lighting requirement or for rooms of a type that implies such a requirement. It computes the window size from the room area.

Generate All Floor Finishes

A2 generates all floor finishes of selected types for all rooms.

Generate All Wall Finishes

A2 generates all wall finishes of selected types for all rooms.

Generate All Ceiling Finishes

A2 generates all ceiling finishes of selected types for all rooms.

Generate All Roof Coverings

A2 generates all roof coverings of a selected type.

Generate a Door

The user requests A2 to generate a door in a selected enclosure or partition.

A2 displays the currently available door types depending on whether an exterior or interior door is required. The user selects a type.

A2 generates a door of the selected type and places it in the selected wall. A2 displays the door.

When the user commits to the door, A2 stores its description persistently and broadcasts its description.

Generate a Window

The user requests A2 to generate a window in a selected enclosure.

A2 displays the currently available window types. The user selects a type.

A2 generates a window of the selected type and places it in the selected wall. A2 displays the window.

When the user commits to the window, A2 stores its description persistently and broadcasts its description.

Change the Location of a Component

The user selects a component and moves it interactively to a different location without changing its shape.

When the user commits to the change, A2 updates the description of the component and broadcasts this description.

Change the Shape of a Component

The user selects a component and changes its shape in the preview display using 'pulling' or 'pushing[9] operations on corners, edges, or faces.

When the user commits to the change, A2 updates the persistent description of the component and broadcasts this description.

Change a Finish

The user requests changing the finish of a selected component.

A2 shows the available finishes, and the user selects one. A2 updates the display of the finish; this will involve at least using a color appropriate to the finish material, if not some form of texture mapping, when this is appropriate.

When the user commits to the change, A2 updates the persistent description of the component's finish (probably an attribute value) and broadcasts this description.

Receive Energy Analysis

This use case works analogously to the use case "receive structural component description."

5 CONCEPT MAPPING BASED ON THE OBJECT MODELING LANGUAGE OML

Concept Mapping

A basic issue in developing collaborative engineering systems is the representation of the product and process information that must be communicated between agents. This product information should include the geometry of the physical parts of the product and their relationships, non-geometric information (e.g., details on functionality of the parts, constraints, and design intent), and multiple levels of abstraction. The software used in the ACL project by the participating agents is so diverse that for pragmatic reasons, a common design representation shared by all agents could not be found. The group as a whole decided therefore that the individual agents in the ACL project maintain their own internal representations and generate from these the views most appropriate to the tasks they execute. That is, the ACL project takes a de-centralized approach to agent coupling (and, by implication, product modeling). This also eliminates the possibility of having a central agent controlling the *scheduling* of agent activities.

Some of the participating agents 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. It is the responsibility of the facilitator to translate between the different representations in a way that maintains consistency with the other agents' representations.

The scenarios in Section 3 illustrate concretely some obvious and important implications of choosing a distributed product model. The communication between agents is primarily about objects representing building elements or components, to which comments in the broadest sense of the term are added during negotiations. A component in which several agents are interested will have an independent representation in each agent's internal design representation. In fact, the information clustered around one component in one agent's representation may be distributed over several components in the representation of another agent. As a result, objects are not communicated directly or 'shared'; rather, the agents exchange *descriptions* of objects, which we were careful to demonstrate in the scenarios. In this situation, it is of utmost **importance that *each agent keeps its internal representation of the current design components up-to-date,*** based on the descriptions it receives about newly created objects or about changes made to existing objects by other agents.

It is likely that an agent may create additional versions of a design component that differ from the version that it broadcasts to other agents. An agent is free to save (i.e. store persistently) different versions of the objects of interest, which may or may not originate in the agent itself, depending on how it structures and manages its design tasks. But we strongly suggest that each agent store persistently at least all versions of the objects *it* creates and broadcasts so that it will be able to answer queries about prior design states; this may become important, for example, if one wants to unravel the reasons behind certain conflicts and how they were resolved. A related issue is who decides when a design is finished and which versions of the objects it contains are part of the final design. Again, this is not automatically clear in a distributed product model that is updated in unpredictable ways by the asynchronous decisions of distributed agents. The nego-

tiation protocol introduced in part 3 is designed to be able to handle this issue, (the ACL team decided as a whole not to address this issue initially).

The lack of a common design representation that could be shared by all agents in the ACL project implies that *the system as a whole needs a mapping mechanism.* Concepts and objects in different local representations must be mapped, or even re-synthesized, when they are communicated among agents. On the other hand, not every agent has access to the whole design; in fact, an agent sees only those portions of a design that it understands and is interested in.

Concept mapping and object re-synthesis may require attribute transformations, which - in turn - may require attribute extraction from related concepts other than the concept being mapped. We illustrate this by a simple example. Figure 2 and Figure 3 show two different wall concepts, one of which may be used by an architectural agent and the other one by a structural design agent. The figure uses the widely used OMT notation for the specification of objects models.[1]

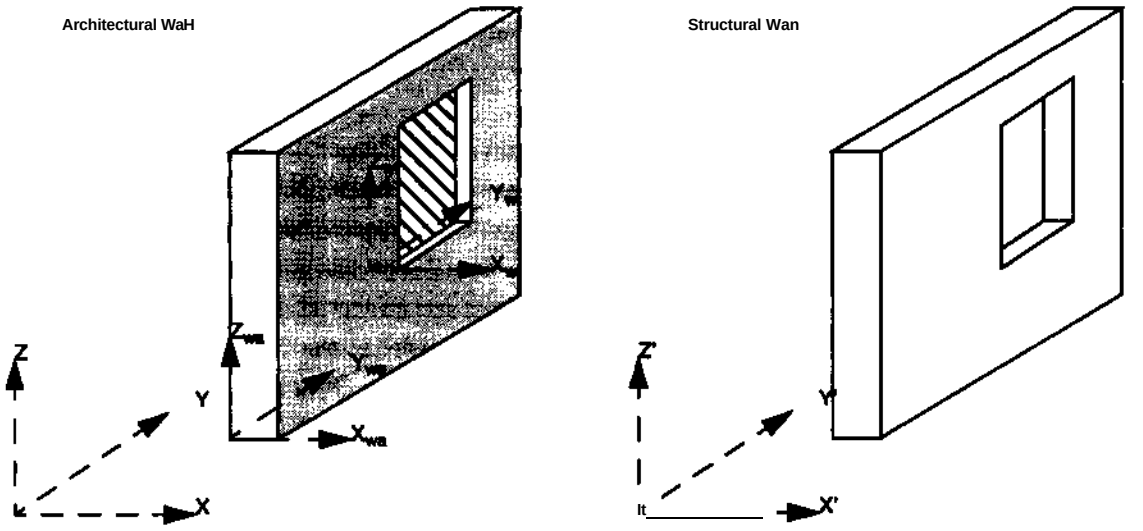


Figure 2. Architectural Wall and Structural Wall

1. **Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F, & Lorensen, W. (1991). Object-Oriented Modeling and Design. Englewood Cliffs: Prentice-Hall.**

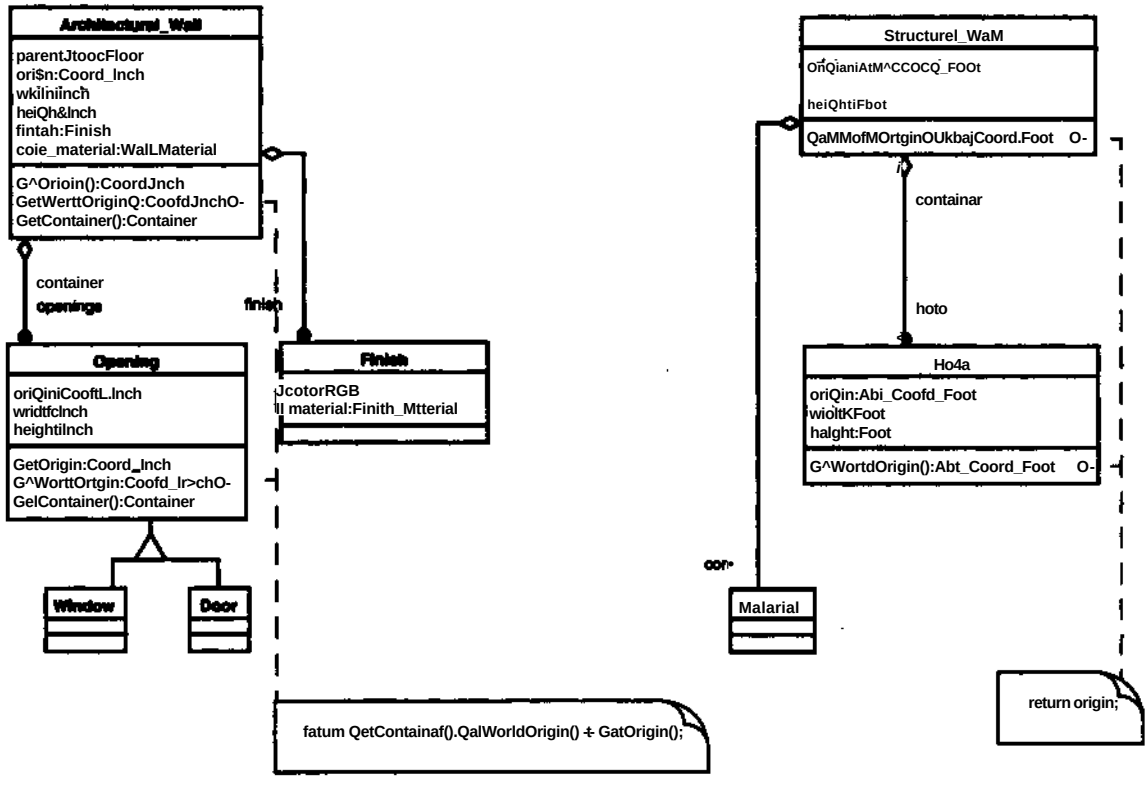


Figure 3. OMT Model of Architectural Wall and Structural Wall

In the architectural wall and window objects, each object is defined in its own coordinate system, where the window coordinate system is relative to the wall coordinate system. The coordinate system of the architectural wall objects is, in turn, relative to a single world-coordinate system. In contrast, the structural objects are defined in one single world coordinate system, where all coordinates are relative to a single origin.

A quick glance at Figure 3 reveals that the mapping from Architectural_Wall to Structural_Wall requires

1. coordinate transformations for origin, width and height
2. synthesis of object StructuraLWall::core based on the attributes of Architectural_Wall::core_material
3. synthesis of object StructuraLWall::hole based on the attributes of ArchitecturaLWall::openings.

The mapping from Structural_Wall to Architectural_Wall requires

1. coordinate transformations for origins, width and height
2. synthesis of object Architectural_Wall::core_material based on the attributes of StructuraLWall::core.

Note that this example is by no means completely worked out; it is only intended to illustrate the notion of concept mapping. The complexity of concept mapping increases significantly when the mapping

process requires recursive concept mappings. The knowledge required for correct concept mappings has to come from the developers who design and implement the concepts.

The Object Modeling Language OML

The facilitator is responsible for concept mapping in the ACL project; thus, communication between top level ACL agents must go through the facilitator before the information communicated reaches the destination agents. To facilitate the generation of appropriate translators, the CMU team developed the Object Modeling Language OML, which the teams utilized to express and communicate the concepts relevant to the task for which they were responsible.[2] A local design representation of an ACL agent can then be mapped into OML objects (see below) and state changes to these OML objects forwarded to the facilitator via a message server of the kind described in the next section; see Figure 4.

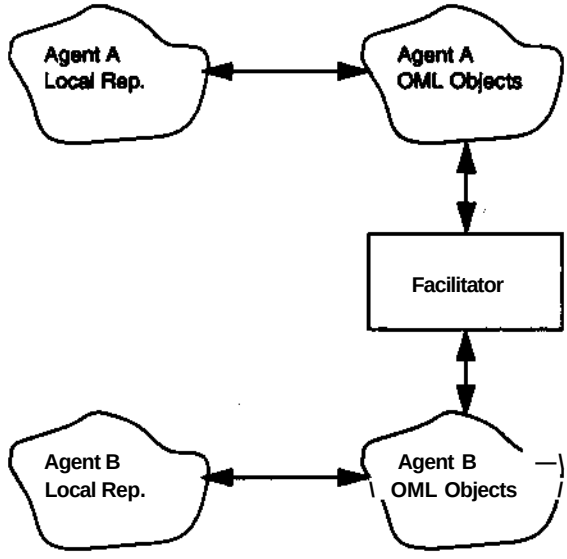


Figure 4. OML-Based Design Representation Mapping

For closely related agents, it may be possible to develop an OML schema that is shared by these agents, which would make the concept mapping mechanism provided by the facilitator unnecessary (see Figure 5).

2. **Snyder, J. and Flemming, U. (1994) *ACL Object Model Specification Language.* Technical Report at http://seed.edrc.cmu.edu/ACL/ObjModelAPI/objapi.book.html**

Figure 5. OML-Based Communication Without Mapping

OML is schema-based and independent of programming languages. It is designed to provide semantic consistency for inter-agent information exchange. Based on an agent's internal representation, an OML schema builds an OML model external to the agent, which describes the agent's model to the outside world. Given such an OML schema, each agent developer provides a language binding specification that is used to map the local representation to and from the OML schema.

The mapping between different local representations involves a large amount of monotonous binding assignments, which - in addition - must be verified. This verification is error-prone and time-consuming, which is especially true when the mapping routines need several programming language linkages, e.g. C, C++, Fortran, or SQL. The necessary code cannot be reliably generated by hand. This creates the need for the automatic generation of language binding code.

By using a language binding compiler provided in the OML standard distribution, the developers can use the OML schema and the language binding specification to generate automatically language binding code in a host language like C++. This code can be compiled into a *language binding library* and linked to an agent's domain code. The standard OML distribution also includes an OML *run-time library*, which delivers packaging and un-packaging of OML objects along with the entire OML communication functionality to the domain application. This OML run-time library should also be linked into the domain application (see Figure 6).

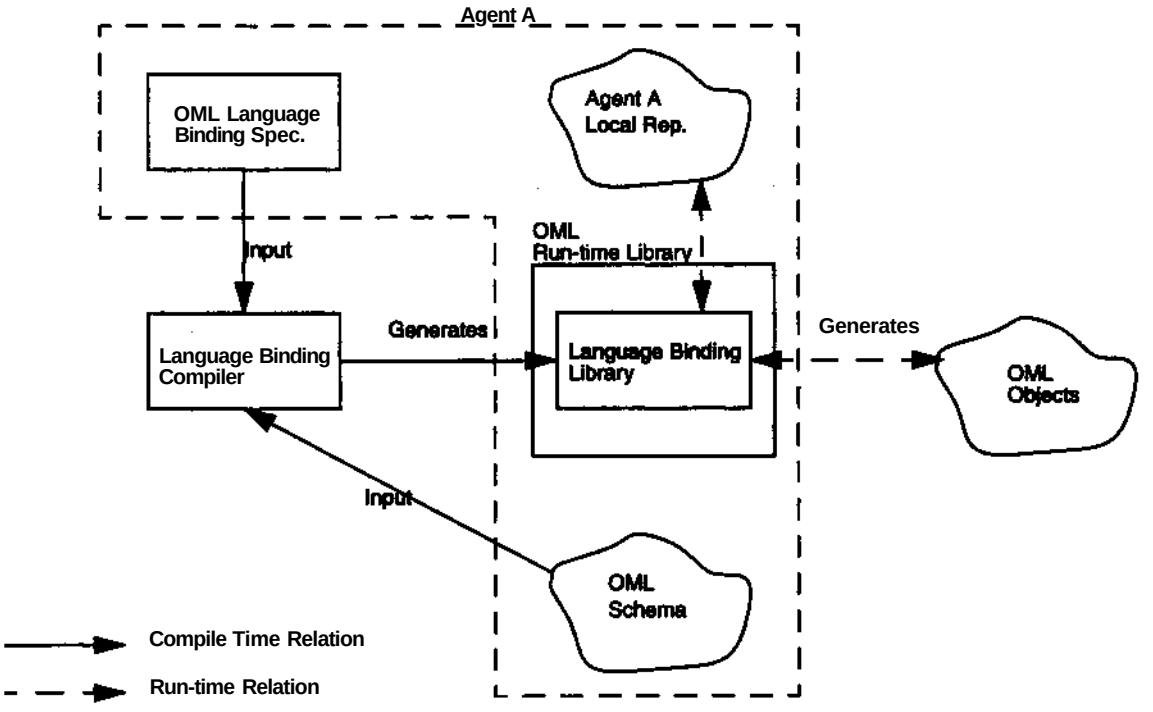


Figure 6. OML-Based Schema Mapping and Language Binding

6 CURRENT IMPLEMENTATION

Overview

Agents in a distributed environment may be structured internally in a hierarchical fashion. But how other "peer" agents are subdivided should be of no concern for an agent. When we use the term "other agents", we refer to the agents in the same agent communication scope. An *agent communication scope* is composed of a parent agent and all its direct child-agents, see Figure 7 (note that a parent agent may only be a conceptual agent representing the entire configuration, but not exist physically). The black box view of an agent introduced above hides its internal structure and the subagents it comprises. For reasons outlined in section 2, we conceived of A2 as an agent composed of subagents as shown in Figure 8.

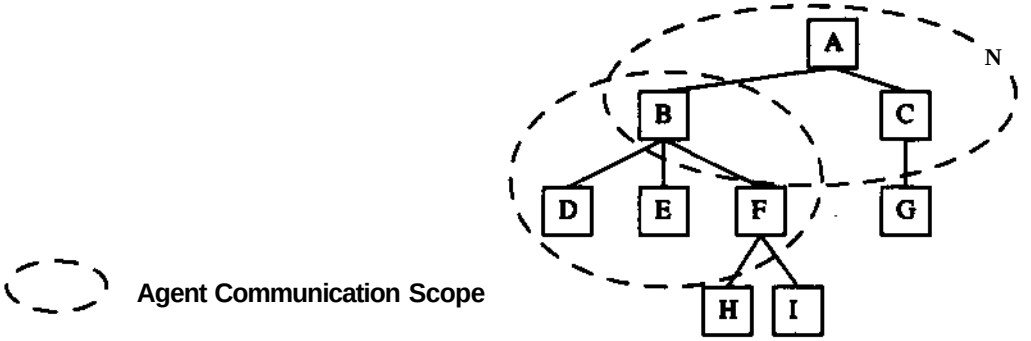


Figure 7. Agent Communication Scope

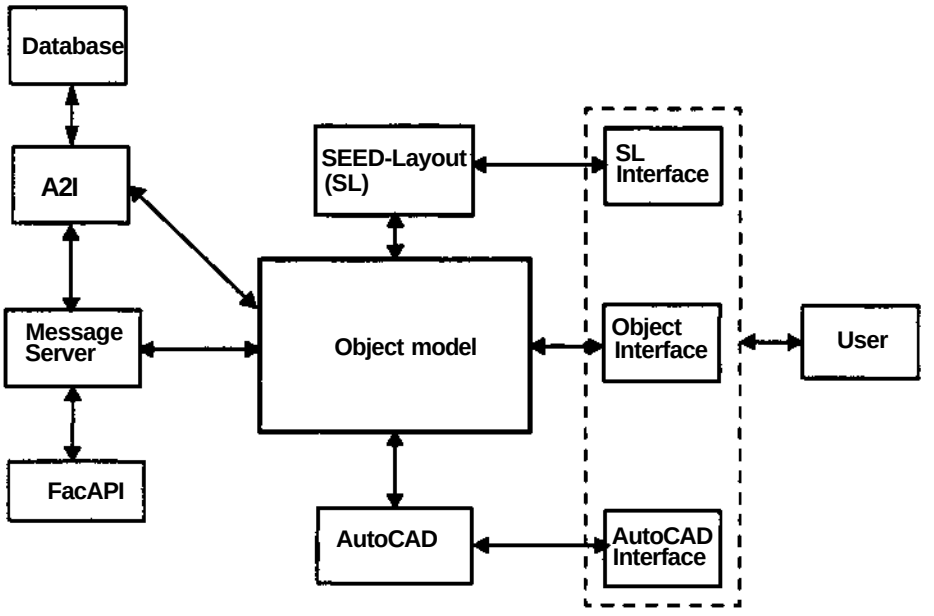


Figure 8. Current A2 Architecture

The central coordinating component is a *message server* able to receive messages from and send them to the facilitator, read from and write into the database, and send and receive object descriptions to or from an object model expressed in OML. This object model consists internally of three parts handling data translations to and from, respectively, SL, the object interface, and AutoCAD.

The role of SL is explained in section 2. AutoCAD is used not as a traditional 'CAD engine' to derive a geometric description of design elements, but solely for 3-dimensional display purposes. The AutoCAD object model provides the generative capabilities for physical, 3-dimensional design components not provided by SL. This part should be replaced by a full-fledged geometric modeler in a next version of A2.

Message Server

The message server is a transaction-based routing server that implements a simple communication protocol. Clients use the communication protocol to request services provided by the message server. Currently, these services consist of

- registering interest,
- broadcasting message transactions, and
- directing message transactions.

A *transaction* is a sequence of OML message frames bound by a special control message in the communication protocol. A transaction is treated as one entity: it is never partially committed or abandoned. The communication between a client program and the message server is completely encapsulated by the OML library, which lessens the burdens on client programming. In fact, the client programs need not be aware of the existence of the communication protocol at all. The overall architecture of the message server is illustrated in Figure 9.

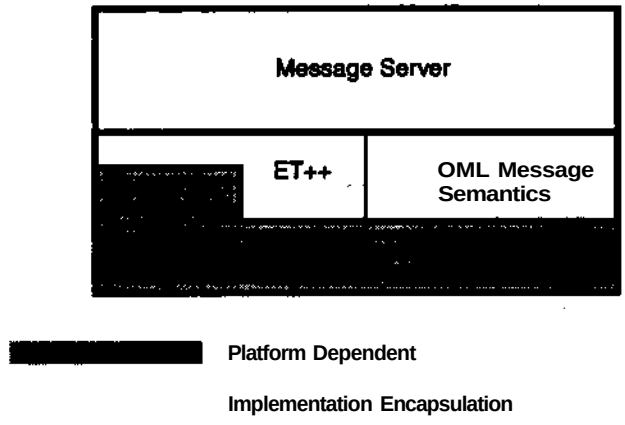


Figure 9. Message Server Architecture Block Diagram

As the figure shows, the window and operating systems are encapsulated by ET++, a C++ application framework.[3] The OML Message Semantics is a part of the OML library that implements the commu-

nication protocol. The OMT design diagram of the Message Server can be found in the Appendix and will not be discussed here. The next paragraphs describe briefly how the message server works.

Each client connected to the message server is explicitly represented as a ClientConnection in the message server. Each ClientConnection can be in one of three *basic states* and two *minor states* at any given time.

Basic States:

- **Connected:** ClientConnections are initially not identified. The only action a client can do in this state is to identify itself. There are two kinds of identifications: normal and privileged. A client with *privileged* identification does not have to register any interest at all; it is, by default, interested in everything.
- **Registered:** This is the state in which the client has identified itself. The only action supported in this state is registering the interests of the client and activating the ClientConnection. The registered interests of a ClientConnection are kept throughout the life-time of the ClientConnection. All interests are registered in terms of OML object classes.
- **Activated:** A ClientConnection is fully qualified only in this state, which allows for routing of transactions to this ClientConnection. All transactions a client submits to the message server in this state are broadcast and routed to all interested clients. Routing occurs at the message frame level instead of the transaction level. The message server inspects each frame message and routs the message to each interested client. When a client is activated, the message server automatically retrieves all instances of the currently registered interest classes and routes them back to the activating client program. Automatic instance retrieval allows a client program to return to a previous state after the client program experiences exceptions, e.g. a core dump. In addition to routing frames to interested clients, the message server routes one copy of every frame to privileged clients. Since the number of frames routed by the message server is large, registering as a privileged client is strongly discouraged.

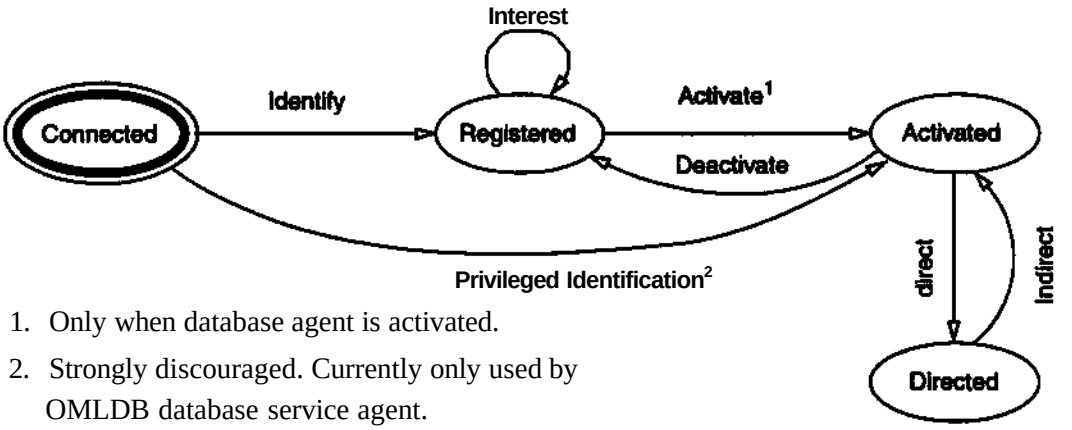
Minor States:

- **Directed:** The Directed state can only be entered through an Activated state. A client can request entering a Directed state to direct transactions only to an activated client instead of broadcasting the transactions to all interested clients. A directed transaction is fully routed to the destination client and disregards the interests of this client. Directed transaction should be used with caution because the destination client may not be interested and thus not understand the message at all.
- **Query:** The Query state can be entered only through an Activated state. A client can query any other activated client in the system, where the query transaction is only delivered to the client being queried. The answers of the query is sent back to the querying client through a directed transaction. This state is currently used extensively for querying a database agent for session management.

The state transitions of a ClientConnection are shown in Figure 10.

---

3. **Wcinand, A. and G. Gamma. (1995) ET++ - *a portable, homogenous class library and application framework.* Taligent Inc. Cupertino, CA**



1. Only when database agent is activated.
2. Strongly discouraged. Currently only used by OMLDB database service agent.

Figure 10. Client Connection State Transition Diagram

We provide a user interface for monitoring the status of each of the message server's ClientConnections as shown in Figure 11..

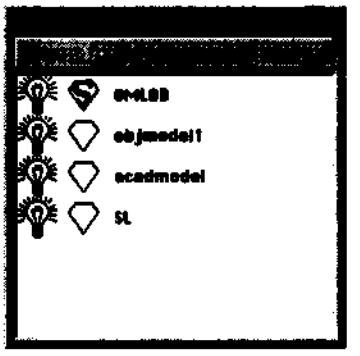


Figure 11. Message Monitoring Interface

Architecture Agent Interface (A2I)

A2I is a software agent serving two independent purposes, data browsing for A2 and software encapsulation for the UniSQL database. Data browsing is used to browse class definitions and object instances through the OML library and UniSQL encapsulation, where the UniSQL encapsulation is not visible to the users of A2I; its function is to provide object instance storage services to other agents in the system. The overall system architecture of A2I is depicted in Figure 12; this diagram should be seen as a detailed view of the corresponding rectangle in Figure 8.

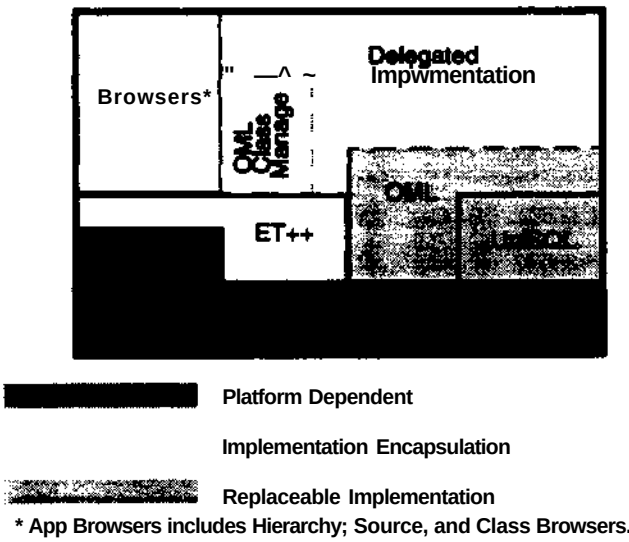


Figure 12. A2I Architecture

Platform-Dependent Portion

The platform-dependent portion of A2I consists of the operating (OS) and window systems (WinSys). An application encapsulates these under the ET++ framework. Access to OS and WinSys occurs through calls in this encapsulation. However, the OML library and UniSQL have not been rewritten with ET++ and therefore access OS and WinSys directly.

The application domain code is thus OS/WinSys independent such that portability of the application domain code is maximized.

Replaceable Implementation

A2I uses two external implementation libraries, the OML library and UniSQL library. These two libraries are linked into A2I and are encapsulated again by the Delegated Implementation part such that the application domain code does not access OML and UniSQL directly. These libraries are, in turn, encapsulated by the implementation encapsulation portion providing services to the application domain code.The services are thus replaceable without affecting any other part of A2.

Implementation Encapsulation

The implementation encapsulation provides a simpler, more consistent interface to the application domain code such that changes to the OML and UniSQL libraries are hidden; it also caches database objects.

The implementation encapsulation is composed of two subsystems, the Delegated Implementation and the OML Class Manager. The Delegated Implementation does most of the library encapsulation and del-

egation. Instead of accessing OML and UniSQL libraries directly, the application domain code accesses the OML and UniSQL services through the Delegated Implementation. The Class Manager provides an ET++-viewable class hierarchy and makes use of the Delegated Implementation. ET++ provides services for traversing the OML class hierarchy and produce multiple views of it efficiently and appropriately for our application domain.

ApplicationBrowsers

The application browsers are the user interface for OML class and instance browsing. A2I currently offers three browsers, a Hierarchy Browser, Source Browser, and Class Browser.

The *Hierarchy Browser* provides a graphical representation of the internal class hierarchy maintained by the OML Class Manager, see Figure 13. It offers different ways of viewing a OML class hierarchy as well as some user controls of the class browser. The *Source Browser* is an editing tool for the OML class source code; see Figure 14. The *Class Browser* provides structure and instance browsing for an OML class; see Figure 15.

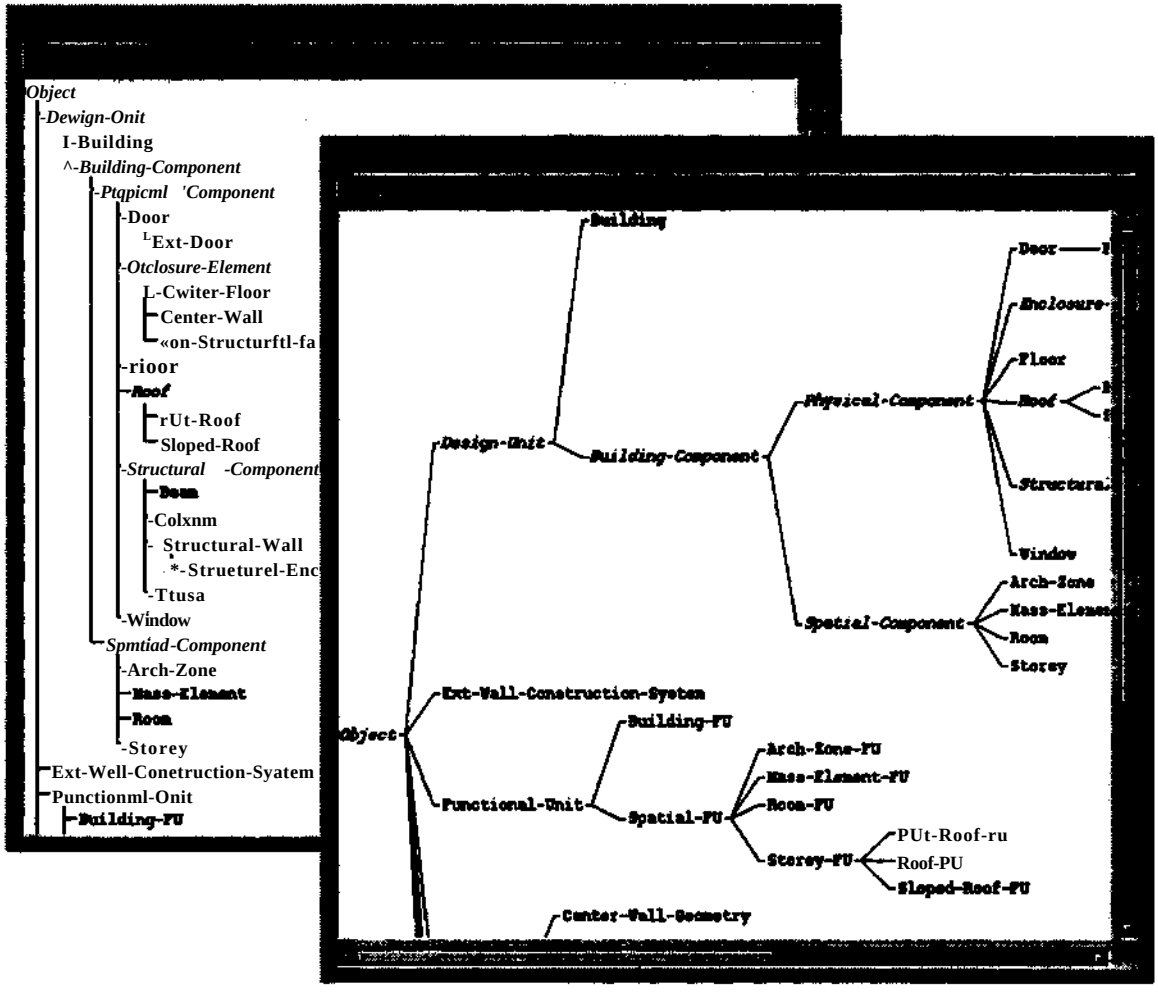


Figure 13. Hierarchy Browser in Different Views

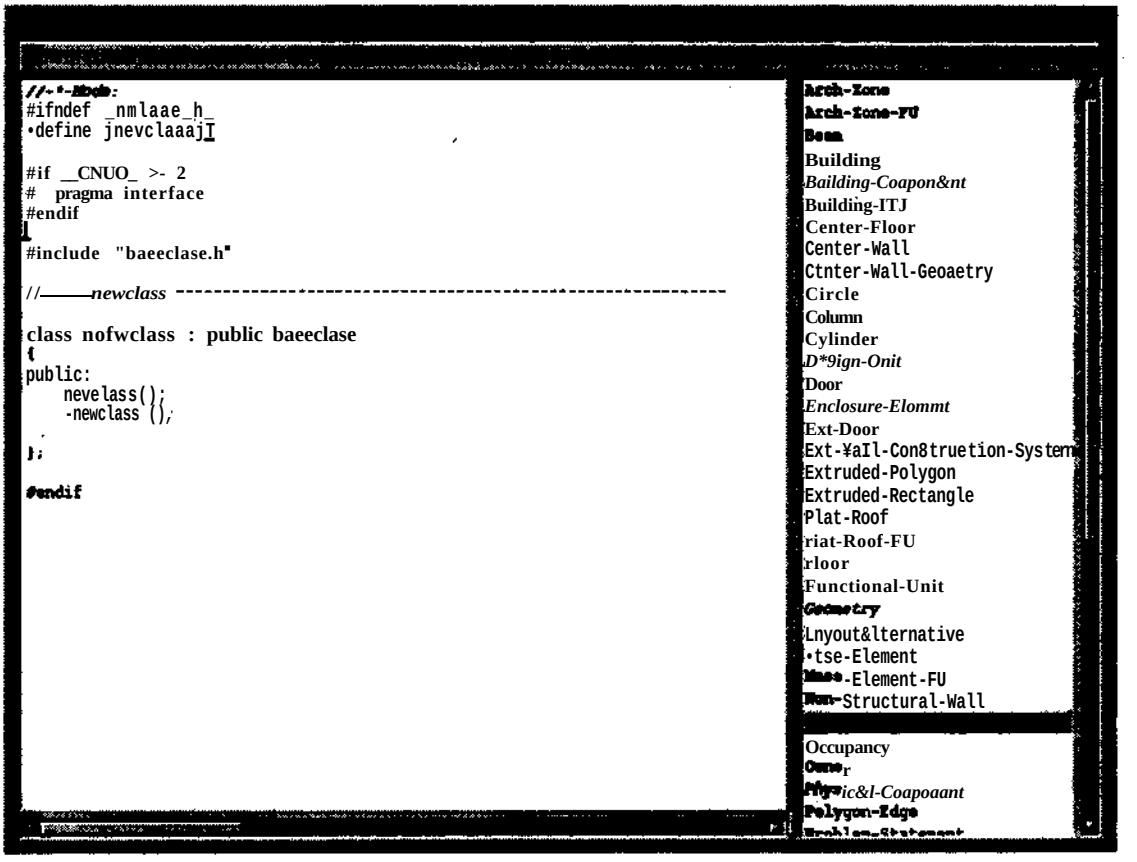


Figure 14. Source Browser

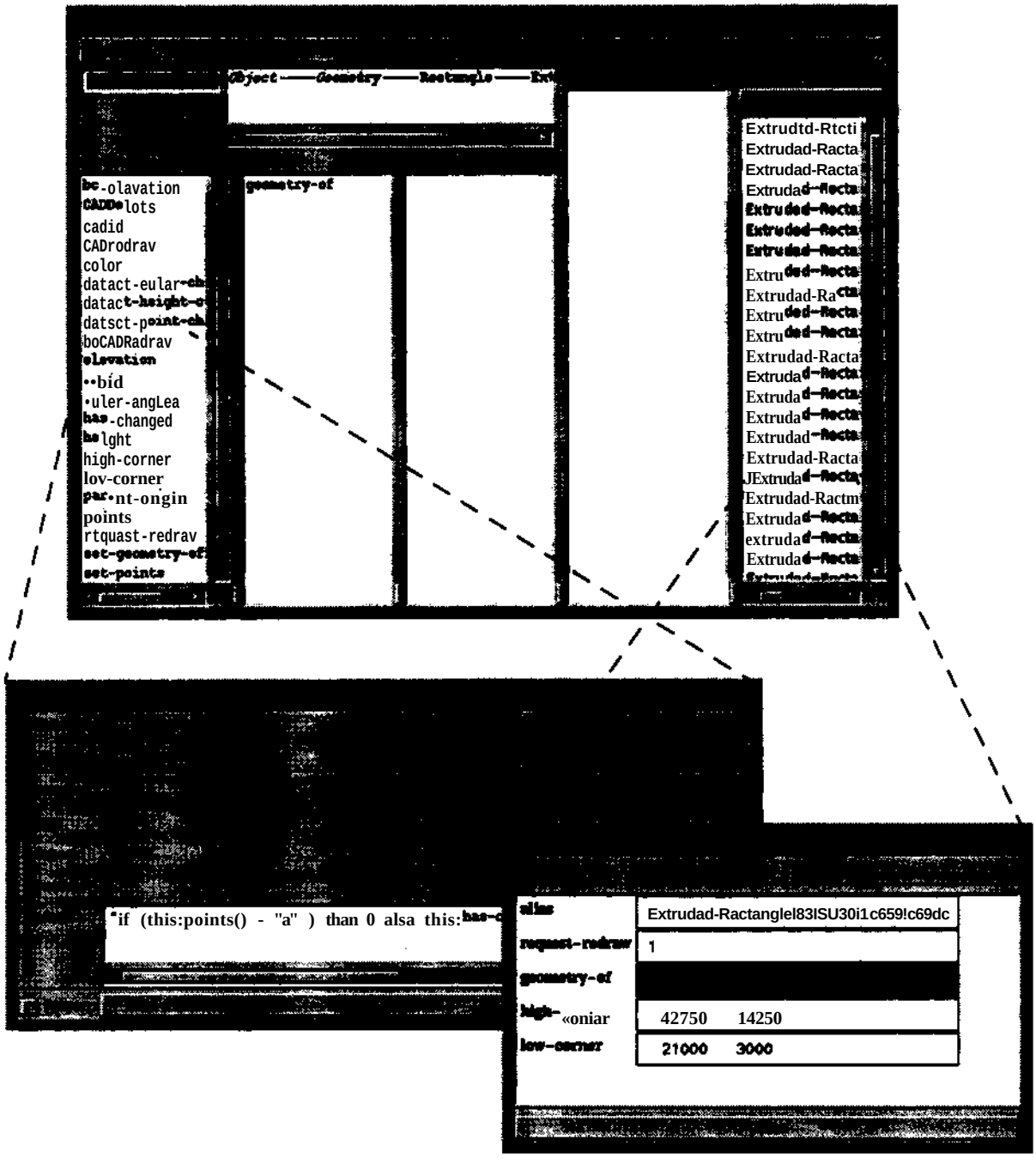


Figure 15. Class Browser

SEED-Layout

The standard SL interface supports the capabilities provided by SL for A2; this interface consists in turn of different display elements (windows, menus etc.) serving different purposes. Figure 16 shows the central "design window" from which the user can invoke most of the generative capabilities of SL; the lay-out displayed is a sublayout in the administrative wing of a firestation.

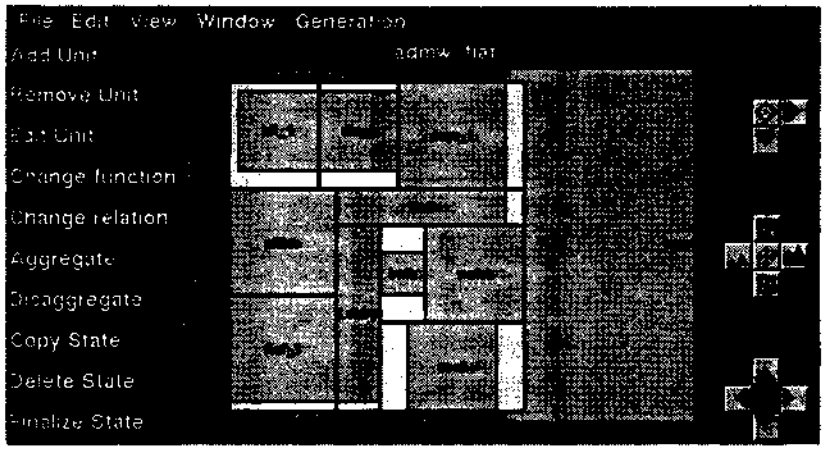


Figure 16. SL Design Window

AutoCAD Agent (ACAD)

The geometric descriptions of design components are provided by the object model expressed in OML and not derived from a traditional 'CAD engine'. As stated earlier, we use AutoCAD merely as a short-cut for 3-dimensional display purposes. Figure 17 shows as an example the display of a firestation lay-out generated by SL and communicated to A2 under the firestation scenario. The firestation contains the major functional units of a "one-company, satellite" firestation on an Army base with a central apparatus room adjacent to an administrative wing on one side and a dormitory wing on the other side. Note that SL does not deal with roof shapes. It only communicates to A2 if a roof is supposed to be sloped or flat. Based on this information, A2 determines initial shapes for sloped roofs (using simple rules-of-thumb) strictly for display purposes. These shapes will be updated once the structural design agent has broadcast its structural roof designs.

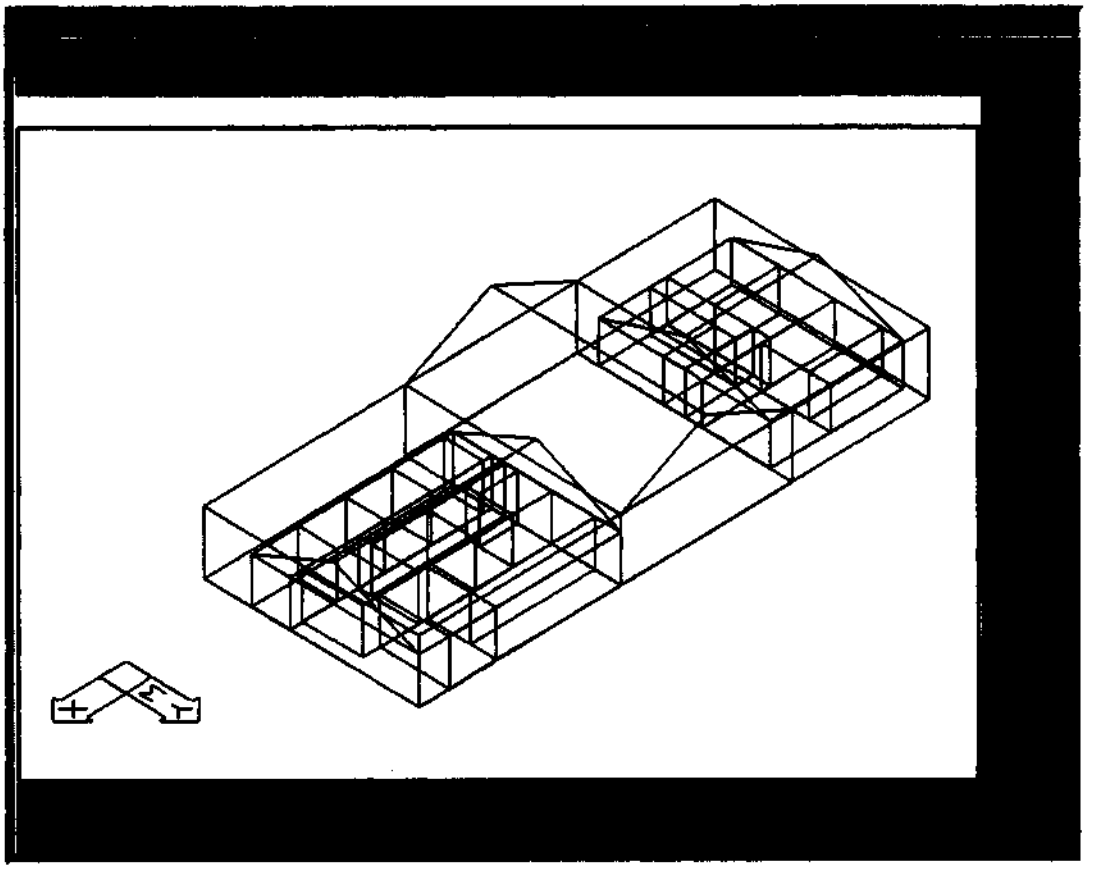


Figure 17.3-Dimensional Display in the AutoCAD Window

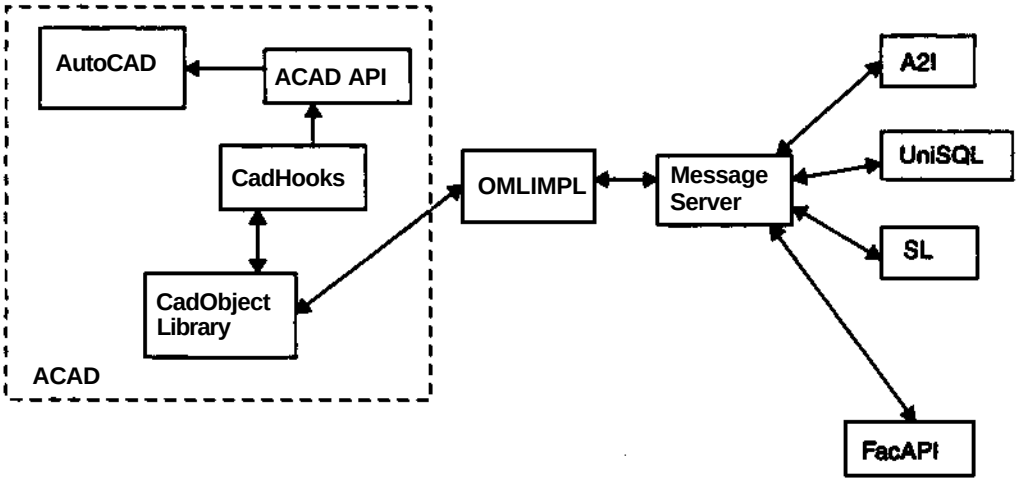


Figure 18. AutoCAD Agent Architecture

Figure 18 illustrates the components and connectors for the AutoCAD agent (ACAD) handling the 3D displays. AutoCAD receives the geometric descriptions of design components from the ACAD API which, in turn, uses AutoCAD's Advanced Modeling Extension AME to update the display of the design components. Each updated geometry is represented by a *cad object* that belongs to the AutoCAD object model and is assigned a unique identifier by ACAD API procedures. CADHooks is a component that maintains a dictionary of these objects and their cad object identifiers, which designate their AutoCAD representations within the CAD Object Library. The data translations to and from SL and AutoCAD, and consequently the mapping of a geometric description of a design component in SL to a *cad object* in AutoCAD, is performed by the object model using language binding translators.

7 CONFLICT MANAGEMENT IN THE ACL PROJECT

Basic Assumptions

A *conflict* is a disagreement between agents about a design decision made by some agent and broadcast to other agents. It can only arise because this decision violates other agents' evaluation criteria (we assume that an agent does not make, or at least does not broadcast, a decision that violates its own evaluation criteria). The disagreement can originate with a piece of evaluation software or with a human designer interacting with an agent; that is, a conflict might be asserted directly by a human designer through the agent.

This view of conflicts, together with the decentralized agent coupling adopted for the ACL project, has the following implications for conflict management and negotiation between ACL agents:

1. Conflict detection is triggered by data updates.
2. Direct negotiation between agents needs a mapping mechanism like concept communication does.
3. Since there is no central agent controlling the scheduling of agent activities, negotiation between agents has to follow a strict protocol.
4. A conflict scheme will be shared by all agents and there will be a centralized conflict management mechanism.

We assume that an OML *Conflict class* is shared by all agents as described in section "Negotiation Protocol". We also assume that the facilitator has been augmented to include the following capabilities:

1. route Conflict instances to appropriate agents according to interest and capability lists registered by the agents,
2. translate OML instances that are referred to in Conflict instances,
3. determine the list of agents interested in a Conflict instance,
4. recognize conflicts caused by OML instances and attributes classified as I/O (see the following section),
5. maintain a representation of those parts of each agent's schema that are communicated (because of 3 and 4).

Conflict management can be divided into three phases: detection, rationalization, and resolution[1]. We discuss these phases below in the context of the ACL project.

Conflict Detection

As stated above, conflicts are detected in the ACL project by individual agents. An agent incorporates externally generated design decisions into its local design representation. With help from any locally available software tools, it may then perform consistency checks, apply analysis or simulation tools, or simply display the design to the user. Any one of these or similar actions may lead to the detection of a conflict like

1. **Case, M. P. (1994) *The Discourse Model for Collaborative Engineering Design: A Distributed and Asynchronous Approach.* Ph.D. Thesis. Graduate College of the University of Illinois at Urbana-Champaign.**

spatial interference, violation of a computable design requirement, or violation of a subjective criterion evaluated intuitively by the user. That is, an agent detects conflicts by whatever method it deems appropriate.

Each attribute in an OML object can be classified into one of the four categories, *INPUT, OUTPUT, I/O* and *PRIVATE.* Through these classifications, the direction of data flow and filtration of data can be controlled at the granularity of attributes. The *PRIVATE* category specifies that an attribute should not be exposed to external agents. The *INPUT, OUTPUT,* and *I/O* categories indicate the direction of the data flow with respect to the agent who owns the OML schema; they also imply the privilege for modifying an attribute. The *OUTPUT* and *I/O* categories indicate that any agent can modify an attribute so classified; therefore, these types of attributes are subject to conflicts caused by modifications done by several agents and can be *automatically detected by the facilitator.*

Once an agent detects a conflict, it generates an instance of the conflict object and broadcasts it. It is unlikely that an agent detecting a conflict knows to whom the conflict should be addressed. This is the reason why we assumed that the facilitator has been augmented to have sufficient knowledge for determining the agents interested in a conflict.

We do not advocate for the ACL project, and building design in general, what we call the "speak now or forever hold your peace" approach toward negotiation. In this approach, a conflict is sent out to all agents together with a time limit for their response; only agents that respond in-time will be invited to the conflict resolution process. We do not take this approach because (i) agents may not have sufficient information to comment on the conflict when it is broadcast, or (ii) not all agents may be available at that time (e. g. because they are temporarily deactivated or have not even joined the team). Both reasons ultimately reflect the fact that design takes place, in principle, in an "open world": what is true at one time may become untrue later and vice-versa because the information available changes dynamically and unpredictably over the duration of a project.

Rationalization & Conflict Resolution

We choose human negotiation as the major conflict resolution strategy based on the following reasons. Not all aspects of design can be expressed in computable form, and not all design evaluations can be done by software. The knowledge needed to detect and resolve conflicts involving such evaluations is simply not available to the computer. Furthermore, a design satisfying all computable evaluation criteria is not necessarily a good design; thus, trade-offs involving non-computable criteria have to involve human judgment. Aside from these technical reasons, there are also ethical and legal reasons for leaving conflict resolution in building design to humans: the trade-offs that have to be made may influence the well-being of the occupants of the building under consideration significantly or may have legal implications for which the designers can be held accountable. They should therefore be aware of the conflicts that arise in the course of a design and take an active part in conflict negotiations.

A conflict is *resolved* if and only if all agents in the resolution process explicitly agree with a design decision that resolves the conflict. Therefore, the conflict resolution process needs an explicit agreement

strategy. If we furthermore assume that a design is satisfactory when no outstanding conflicts exist, each agent in the system should at least keep a list of the outstanding conflicts it is interested in.

Agents obviously should pursue the goal to make design decisions that do not lead to conflicts detected by other agents. However, a design without conflicts still needs to be explicitly approved by all agents to become a candidate for a final design; for example, there may be parts of the design that need further elaboration by some agent that simply has not gotten around to deal with this issue. The negotiation protocol presented below can be extended to handle this situation.

8 NEGOTIATION PROTOCOL

Scope

The proposed protocol captures the information necessary to manage conflicts generated during the course of a design project and facilitates its communication among the agents interested in the conflict. For the reasons stated in the preceding section, the automatic detection and resolution of conflicts are not supported by this protocol. However, by keeping an accurate history of a negotiation and resolution process, the proposals made, and the response of the participants, conflicts can be managed in an orderly manner.

Definitions

This section provides the definitions that are needed to describe the protocol.

Conflict Class

A *Conflict class* collects the information needed to negotiate and resolve conflicts. It consists as the following parts:

- a (potentially empty) *set of conflicting attribute descriptions* (defined in section "Attribute Conflict" below).
- an *agent set* of participating agents and their status (defined in section "Agent Status" below).
- a (potentially empty) *set of counter proposals* (defined in section "Counter Proposal" below).
- a *conflict state* (defined in section "Conflict State" below).

We assume that the facilitator has a persistent storage facility for conflict information, that the agents share the OML definition of Conflict class described below, and that the facilitator performs any translations between schema slot names and values for any conflict information.

Agent Set

The *agent set* is the set of agents capable of adding a comment to a Conflict instance.

Agent Status

The *agent status* is a response to the *definition* of a conflict in a Conflict instance and can be of the following types:

- *nojresponse* (agent has no opinion; this is the default)
- *agree* (agent agrees with the definition of the conflict)
- *disagree* (agent does not agree with the definition of the conflict)
- *dissension* (agent does not want to have the same opinion as the group)

Attribute Conflict

Attribute conflicts identify specific OML instances and slot values in a Conflict instance. Attribute conflicts are specialized into *value conflicts* and *relation conflicts.* Value conflicts target a slot value within a specific OML instance. Relation conflicts target a relationship between specific instances.

Agent Attribute Status

Each agent can identify its status with respect to each attribute conflict. The agent *attribute status* can again be of the following types:

- *nojresponse* (agent has no opinion; this is the default)
- *agree* (agent agrees with the current value of the attribute)
- *disagree* (agent does not agree with the current value, but has no recommendation)
- *proposal* (agent does not agree with the current value and has a recommendation)

Counter Proposal

If consensus cannot be reached, another Conflict instance, called a *counter proposal,* can be initiated in response to the unresolvable conflict; this creates a *tree of Conflict instances.* The parts of the counter proposal need not be the same as the initial proposal.

Counter proposals are particularly useful when the individual parts of a Conflict instance can be agreed upon, but not the whole. For example, an agreement about the height of a window and that of a wall can be reached separately, but taken together, produce a conflict from one or the other agents' perspective because the window is too high to fit into the wall. Counter proposals may create a complex structure as illustrated in Figure 18.

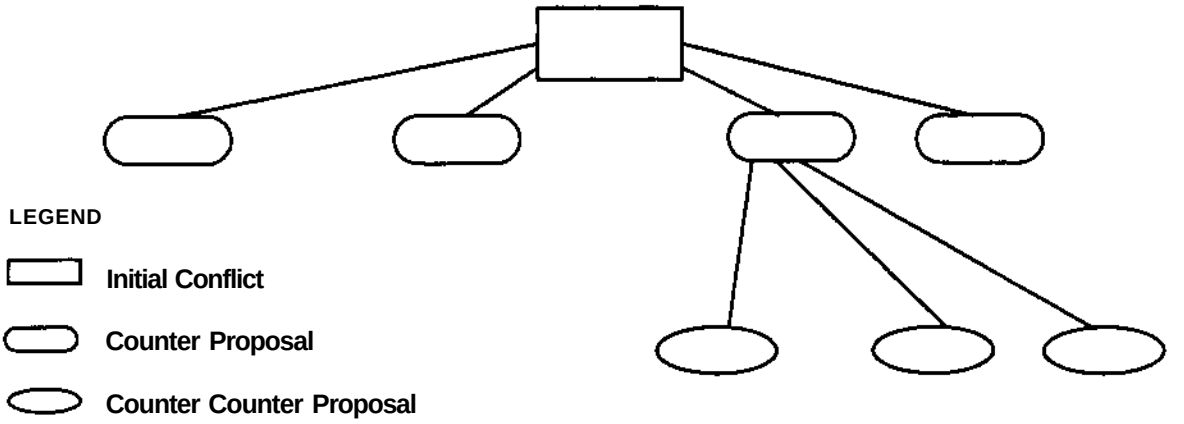


Figure 19. Example Counter Proposal Structure

ConflictState

A Conflict instance can be *OPEN, RESOLVED,* or *CLOSED.* Upon creation, the Conflict is *OPEN,* and the *resolved* attribute is FALSE. During the course of negotiation, a Conflict instance changes state to *RESOLVED* if all agents have the same agent status *(dissension* means agree *and* disagree); resolved Conflict instances can be reopened if an agent status is changed. Resolving one node in a Conflict tree makes the other nodes mute and resolves the whole tree. A resolved Conflict instance (or Conflict tree) can be closed only by the originating agent; once a Conflict instance is closed, it cannot be changed.

We expect that traditional communication methods will be used to reach consensus and closure. The proposed protocol is intended to record necessary historical information, including potential disagreements.

Protocol Specification

An OMT representation of a Conflict class is shown in Figure 19. A Conflict (Object) is an instance of that class. Note that in this representation, Conflict trees are created via the *counterproposal* relationship of the Conflict class.

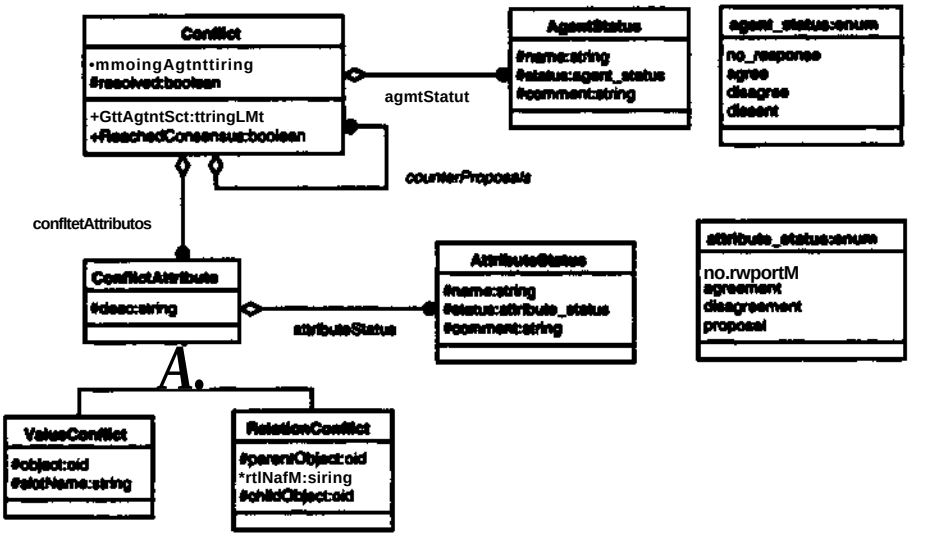


Figure 20. Conflict Composition OMT Diagram

We specify the protocol for conflict management using the state transition diagram shown in Figure 20. The member function *ReachedConsensus* searches the Conflict tree for any node that is resolved. Once a Conflict tree is resolved, the originating agent can request closure.

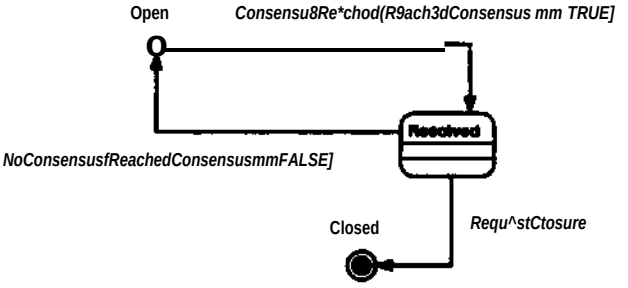


Figure 21. Conflict State Diagram

Negotiation Examples

This section uses the two negotiation scenarios from Section 3 to illustrate the negotiation protocol described above. Assume that a cabin project has been set up and that the cabin design has progressed to the point where SL has generated and committed a schematic layout. Graphical representations of the cabin layout generated by SL are presented in Figure 21 & Figure 22.

In the following examples, instances are shown in italics, slot names are given in upper case, and slot values are enclosed in quotes.

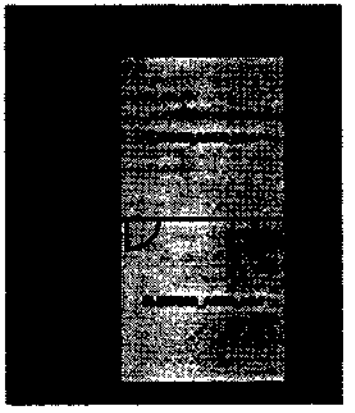


Figure 22. Cabin **Layout Generated** by SL

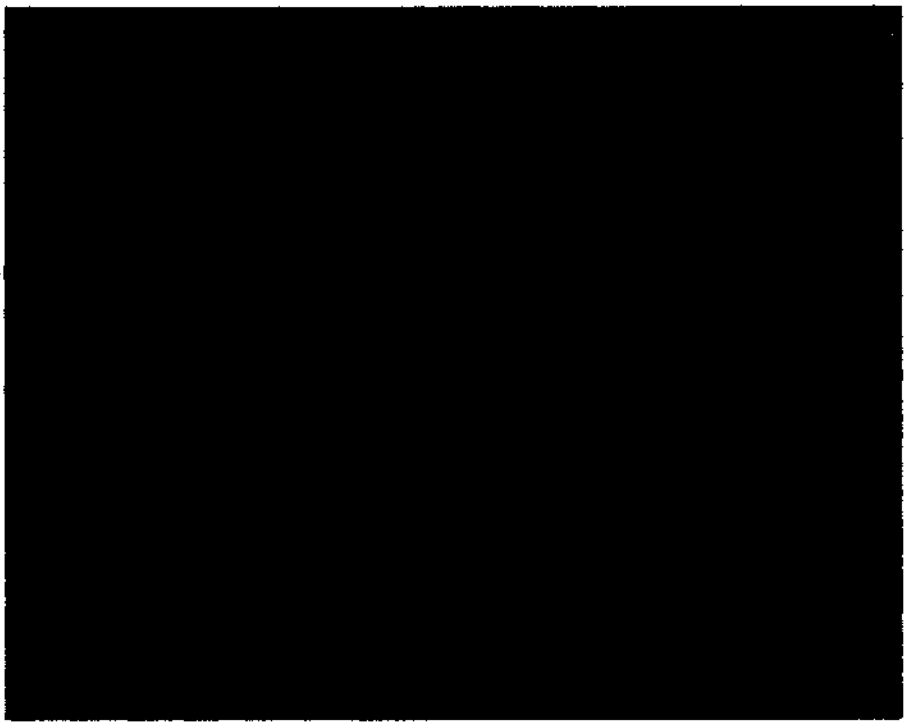


Figure 23. Cabin Displayed by ACad

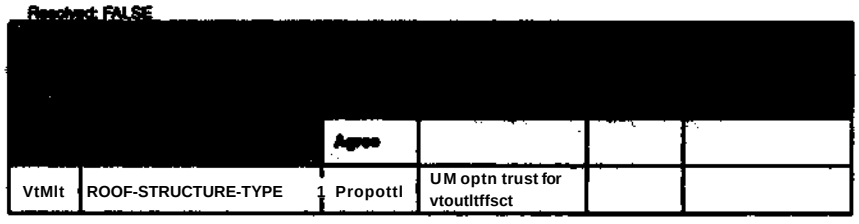
Example 1: Negotiating the type of an element

In this example, two top-level ACL agents are involved in the negotiation, the Structural Design Agent (SA) and A2. In addition, SL is involved, but does not communicate directly with the facilitator.

SA: SA receives OML instances generated by SL describing the cabin configuration.
SA generates *structural-components* based on SUs schematic layout, where the roof is supported by components of RCX)F-STRUCTURE-TYPE "joist".
SA commits the generated *structural-components.*

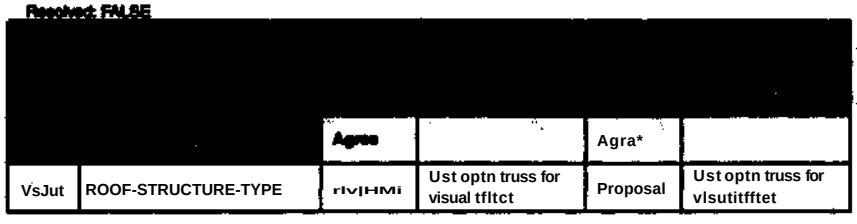
A2: A2 receives *structural-components* generated by SA and displays them.
The user of A2 does not like the look of the roof joists.
She constructs a *conflict* and sets A2's agent status to "agree" and attribute status to "proposal" with the comment, "Use open truss for visual effect". (By definition, A2 agrees with the definition of the conflict.)
The user commits *conflict.*

SA: SA receives *conflict* and notifies its user.

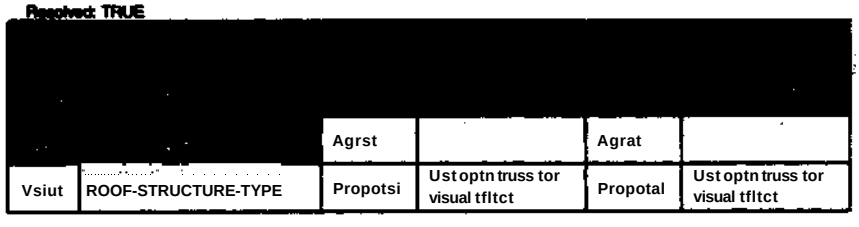


The user of SA agrees with the definition of *conflict.* He sets SA's agent status to "agree" and attribute status to be "proposal" with comment, "Use open truss for visual effect". The user of S A commits *conflict*

A2: **A2 receives *conflict* as modified by SA.**



At this moment, A2 and SA propose the same solution: the conflict is resolved. Since A2 originated the conflict, the user of A2 closes it.



SA: **The user of SA modifies the cabin structure by removing the roof joists and adding roof trusses. He commits the modified design.**

Example 2: Negotiating the placement of objects

This example continues the negotiation example above. Suppose SA generates six trusses as illustrated in Figure 23. Suppose also that an "owner agent" (OW) has joined the project by connecting to the facilitator and has been informed of all generated objects and conflicts.

Figure 24. Trusses **Generated by SA and wall generated** by SL

A2: The user of A2 discovers that *truss 1* and *walll,* a non-loadbearing partition she added to the cabin design, are not aligned.
She constructs a *conflict* with the comment, "Align Truss 1 with Wall 1", and commits it.

OW, SA: Receive *conflict* and notify their users.

Rttototcfc FALSE

		Agree					
Value	origin	Proposal	Align Truss 1 with Wall 1				

SA: The user of SA sets SA's agent status to "agree" and attribute status to "disagree" with the comment, 'Truss location is optimal for cost reasons".
SA commits *conflict.*

A2, OW: Receive *conflict* committed by SA and notify their users.

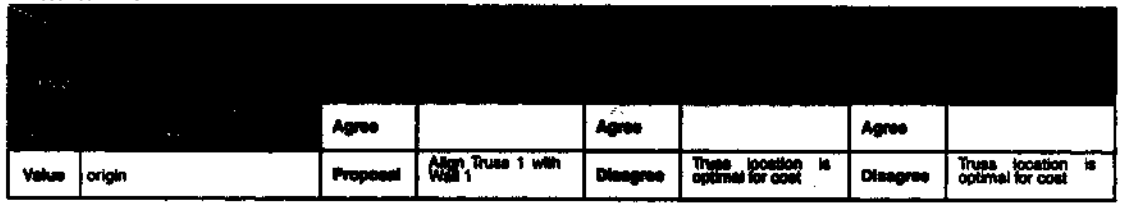
RMOtod: FALSE

		Agree		Agree			
Value	origin	Proposal	Align Truss 1 with Wall 1	Disagree	Truss location is optimal for cost		

OW: The user of OW agrees with the argument of SA (and thus disagrees with A2's proposal).
The user of OW modifies OW's agent status to "agree" and attribute status to "disagree" with comment, 'Truss location is optimal for cost reasons".
OW commits ***conflict.***

A2, SA: Receive *conflict* committed by OW and notify their users.

Resolved: FALSE



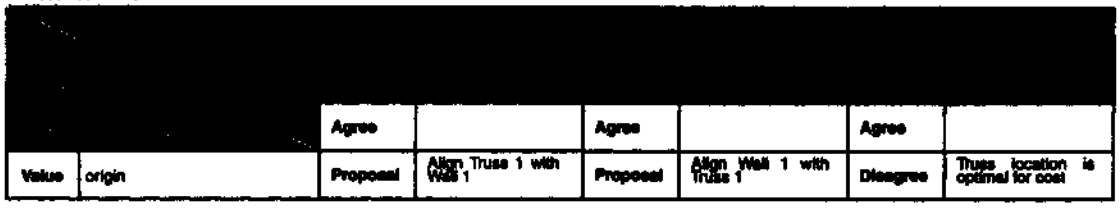
		Agree		Agree		Agree	
Value	origin	Proposal	Align Truss 1 with Wall 1	Disagree	Truss location is optimal for cost	Disagree	Truss location is optimal for cost

SA: **The user of SA modifies the attribute status to "proposal" with comment, "Align Wall 1 with Truss 1".**

SA commits *conflict*

OW, A2: Receive *conflict* and notify their users.

Resolved: FALSE



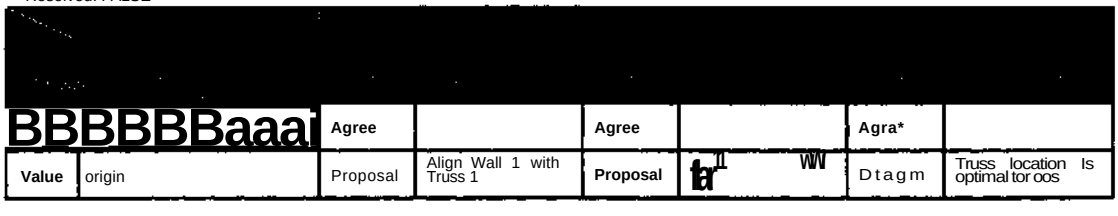
		Agree		Agree		Agree	
Value	origin	Proposal	Align Truss 1 with Wall 1	Proposal	Align Wall 1 with Truss 1	Disagree	Truss location is optimal for cost

A2: **The user of A2 agrees with the proposal and modifies A2*s attribute status to "proposal'[1]' and comment to "Align Wall 1 with Truss 1".**

The user of A2 commits the modified *conflict.*

SA, OW: Receive the updated *conflict* generated by A2 and notify their users.

Resolved: FALSE

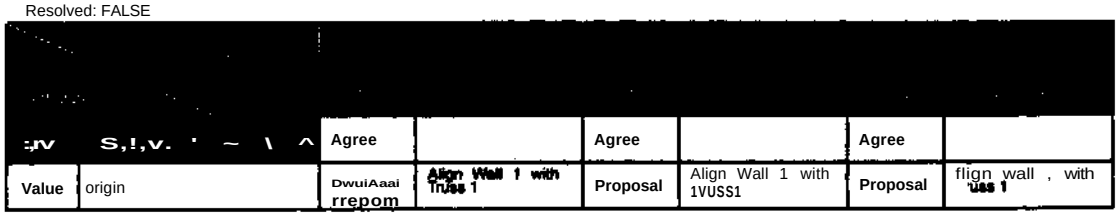


BBBBBBaaai		Agree		Agree		Agra*	
Value	origin	Proposal	Align Wall 1 with Truss 1	Proposal	fa[1] WW	Dtagm	Truss location Is optimal tor oos

OW: **The user of OW modifies the attribute status to "proposal" with comment, "Align Wall 1 with Truss 1".**

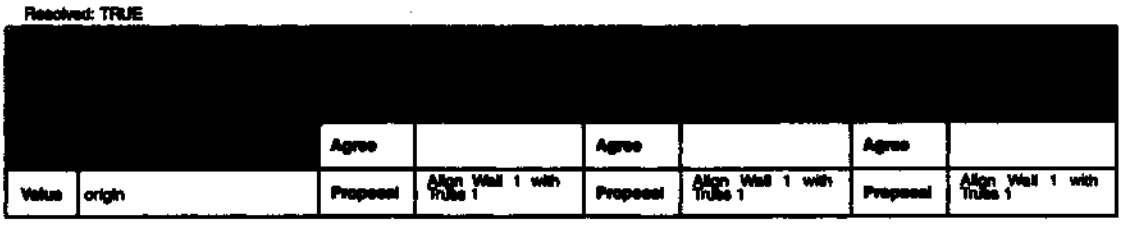
SA, A2: Receive the updated *conflict* and notify their users.

Resolved: FALSE



:,v S,!,v. ' ~ \ ^		Agree		Agree		Agree	
Value	origin	DwuiAaai rrepom	Align Wall 1 with Truss 1	Proposal	Align Wall 1 with 1VUSS1	Proposal	flign wall , with 'uss 1

A2: **The user of A2 announces that *conflict* is closed.**



A2: **The user of A2 aligns *walll* with *trussl* and commits the newly modified design. Since a *door* is attached to *walll,* the *door's* origin is also updated (we assume that the *door's* origin is not relative to *walll).***

SA, OW: Receive the modified design and synchronize their internal design representation with the newly received OML instances.

9 DISCUSSION

The CMU team's observations are grouped below under two headings: project aspects that we deem successful, and lessons learned from less successful aspects.

Successful Project Aspects

We believe that to the project's credit, it chose to address "difficult" issues even though supporting software technologies may not have existed. For example, teams at MIT, UIUC and CMU developed applications with complex internal representations and separate databases, which constitute, as stated earlier, a distributed product model. Software to integrate such applications does not exist, but if distributed collaboration is to succeed in building design, it must be able to handle collaboration between applications of this type. Furthermore, we arrived - as a team - at a clear understanding of what the information exchange and negotiation problems in such a set-up are. In addition, the domain-specific computational models produced by individual teams were innovative and interesting. In particular, MIT's structural agent provides an elegant and comprehensive solution to modeling structural design knowledge.

We also stress benefits derived from the interdisciplinary nature of the project. We were able to identify scenarios for the conceptual phases of building design that result in a less linear design process across the different disciplines. The CMU team was also able to communicate to other teams that architects don't "just draw lines" (as someone put it in an early meeting). We believe that the communication of this kind of information was greatly facilitated by using the World-Wide Web as a document distribution mechanism.

The CMU-developed schema specification tools provided the basis for progress and communication of the team as a whole. As the preceding sections show, we were also able to integrate our own components successfully using the same tools. The language binding compiler enabled us to exchange information reliably between existing applications where other tools would have failed.

Lessons Learned

During the course of the project, we were able to communicate some information between sites, but the system as a whole was never complete or tested. Use cases depending on communication with other agents could therefore not be implemented; this applies to all use cases dealing with design components or evaluations generated by other agents or depending on the existence of such objects. We stress, however, that an implementation would be very easy if the needed information were to become available because the OML model for the objects of interest has been implemented. Parallel experiments between the CMU and MIT teams provided essential verification of the OML approach to schema specification and language binding. However, this experience could not be transferred to the larger project for the reasons stated below.

From an infrastructure perspective, the Internet posed problems: it does not yet appear suitable for the type of serious collaboration aimed at in this project. It is still not as stable as a Wide-Area Network (WAN) or a Local-Area Network (LAN). In addition, teams working in different time zones pose difficult synchronization problems.

The Federation Architecture was not able to support the integration of applications of the given type over the Internet. We offer several reasons for this:

1. **We never knew if information was, in fact, exchanged correctly because we could not verify translation rules or data structure mappings in an application.**
2. **The Federation Architecture did not provide any tools for object-oriented information modeling. Some researchers claim that it *should not* have such tools, but it is clear to us that for the given applications, which represent state-of-the-art object-based modeling techniques, such tools are essential.**
3. **We also observed that the needs of the teams that developed agents (i.e. clients) were not taken into account when services were provided via the facilitator. In particular, we believe the emphasis on object-centered representations was neglected until very late in the project.**
4. **We need rigorous formalisms for schema concept mapping based on which translation code can be generated automatically: manually generated translation rules proved deficient; we need *software* to generate translation rules automatically. This became apparent during a final project meeting at Stanford on Dec. 20,1996. For example, it was very easy to generate manually a rule that can "loop" (i.e. fire multiple times for a given data set), and these kinds of errors were introduced even in very simple translations. This only confirms the experience with several decades of expert systems development: they are brittle and difficult to verify.**

REFERENCES

Akin, <X Sen, R., Donia, M. and Zhang, Y. (1995) "SEED-Pro: Computer-Assisted Architectural Programming in SEED" *Journal ofArchitectural Engineering,* vol. 1 pp. 153-16

Case, M. P. (1994) *The Discourse Model for Collaborative Engineering Design: A Distributed and Asynchronous Approach.* Ph.D. Thesis. Graduate College of the University of Illinois at Urbana-Champaign.

Chiou, J. and Logcher, R. D. (1996) Testing a Federation Architecture in Collaborative Design Process. Res. Report R96-01. Dept. of Civil and Environmental Engineering, MIT, Cambridge, MA

Flemming, U., Coyne, R., and Chien, S.F. (1994) *SEED-Layout Requirements Analysis.* http://seed.edrc.cmu.edu/SL/SL-start.book.html.

Flemming, U. and Chien, S.F (1995) "Schematic Layout Design in SEED Environment" *Journal of Architectural Engineering,* vol. 1 162-169

Flemming, U. and Woodbury, R. (1995) "Software Environment to Support Early Phases in Building Design (SEED): Overview" *Journal of Architectural Engineering,* vol. 1 pp. 147-152

Gamma, E., Helm,R., Johnson, R. & Vlissides, J. (1994) *Design Patterns - Object Oriented Software.* Reading, Massachusetts. Addison-Wesley

Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach.* New York: Addison-Wesley

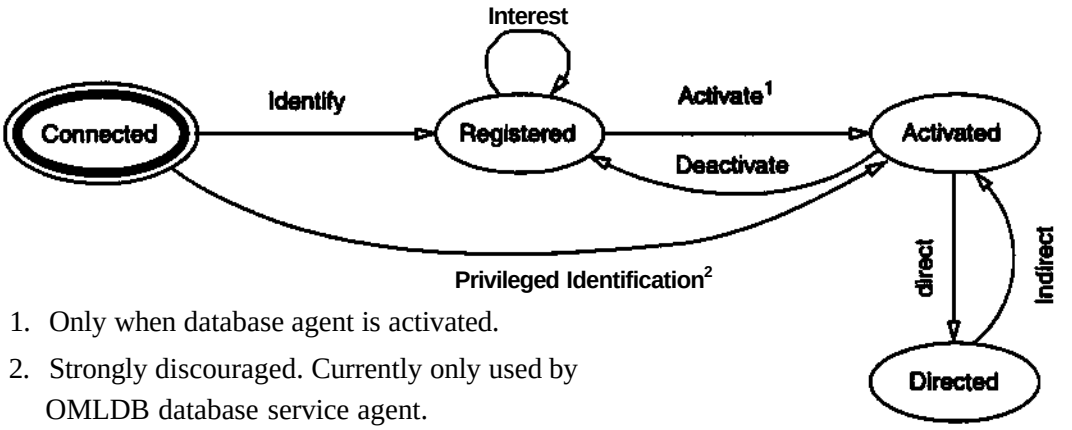
Khedro, T., Case, M. P., Flemming, U., Genesereth, M. R., Logcher, R., Pedersen, C, Snyder, J., Sriram, R. D., Teicholz, P. M. (1995). "Development of a Multi-Institutional Testbed for Collaborative Facility Engineering Infrastructure". In J. P. Moshen (Ed.) *Proceedings of the Second Congress on Computing in Civil Engineering,* pp 1308-1315

McGraw, K. *(1996) Agent Collaboration Environment (ACE).* http://www.cecer.army.mil/pl/ace/.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-Oriented Modeling and Design.* Englewood Cliffs: Prentice-Hall.

Snyder, J. and Flemming, U. (1994) *ACL Object Model Specification Language,* report accessible from http://seed.edrc.cmu.edu/ACL/ObjModelAPI/objapi.book.html.

Wcinand, A. and G. Gamma. (1995) £7++ - *a portable, homogenous class library and application framework.* Taligcnt Inc. Cupertino, CA



1. Only when database agent is activated.
2. Strongly discouraged. Currently only used by OMLDB database service agent.

Figure 10. Client Connection State Transition Diagram

We provide a user interface for monitoring the status of each of the message server's ClientConnections as shown in Figure 11..

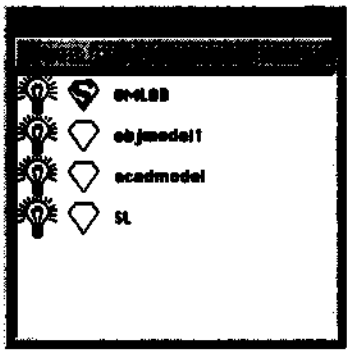


Figure 11. Message Monitoring Interface

Architecture Agent Interface (A2I)

A2I is a software agent serving two independent purposes, data browsing for A2 and software encapsulation for the UniSQL database. Data browsing is used to browse class definitions and object instances through the OML library and UniSQL encapsulation, where the UniSQL encapsulation is not visible to the users of A2I; its function is to provide object instance storage services to other agents in the system. The overall system architecture of A2I is depicted in Figure 12; this diagram should be seen as a detailed view of the corresponding rectangle in Figure 8.

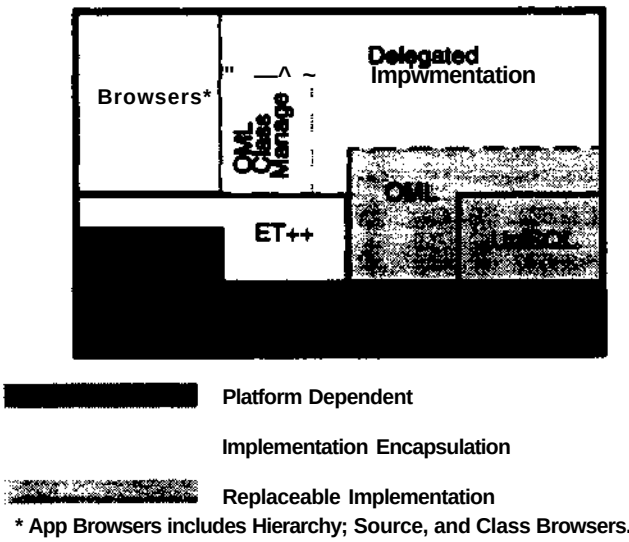


Figure 12. A2I Architecture

Platform-Dependent Portion

The platform-dependent portion of A2I consists of the operating (OS) and window systems (WinSys). An application encapsulates these under the ET++ framework. Access to OS and WinSys occurs through calls in this encapsulation. However, the OML library and UniSQL have not been rewritten with ET++ and therefore access OS and WinSys directly.

The application domain code is thus OS/WinSys independent such that portability of the application domain code is maximized.

Replaceable Implementation

A2I uses two external implementation libraries, the OML library and UniSQL library. These two libraries are linked into A2I and are encapsulated again by the Delegated Implementation part such that the application domain code does not access OML and UniSQL directly. These libraries are, in turn, encapsulated by the implementation encapsulation portion providing services to the application domain code.The services are thus replaceable without affecting any other part of A2.

Implementation Encapsulation

The implementation encapsulation provides a simpler, more consistent interface to the application domain code such that changes to the OML and UniSQL libraries are hidden; it also caches database objects.

The implementation encapsulation is composed of two subsystems, the Delegated Implementation and the OML Class Manager. The Delegated Implementation does most of the library encapsulation and del-

egation. Instead of accessing OML and UniSQL libraries directly, the application domain code accesses the OML and UniSQL services through the Delegated Implementation. The Class Manager provides an ET++-viewable class hierarchy and makes use of the Delegated Implementation. ET++ provides services for traversing the OML class hierarchy and produce multiple views of it efficiently and appropriately for our application domain.

ApplicationBrowsers

The application browsers are the user interface for OML class and instance browsing. A2I currently offers three browsers, a Hierarchy Browser, Source Browser, and Class Browser.

The *Hierarchy Browser* provides a graphical representation of the internal class hierarchy maintained by the OML Class Manager, see Figure 13. It offers different ways of viewing a OML class hierarchy as well as some user controls of the class browser. The *Source Browser* is an editing tool for the OML class source code; see Figure 14. The *Class Browser* provides structure and instance browsing for an OML class; see Figure 15.

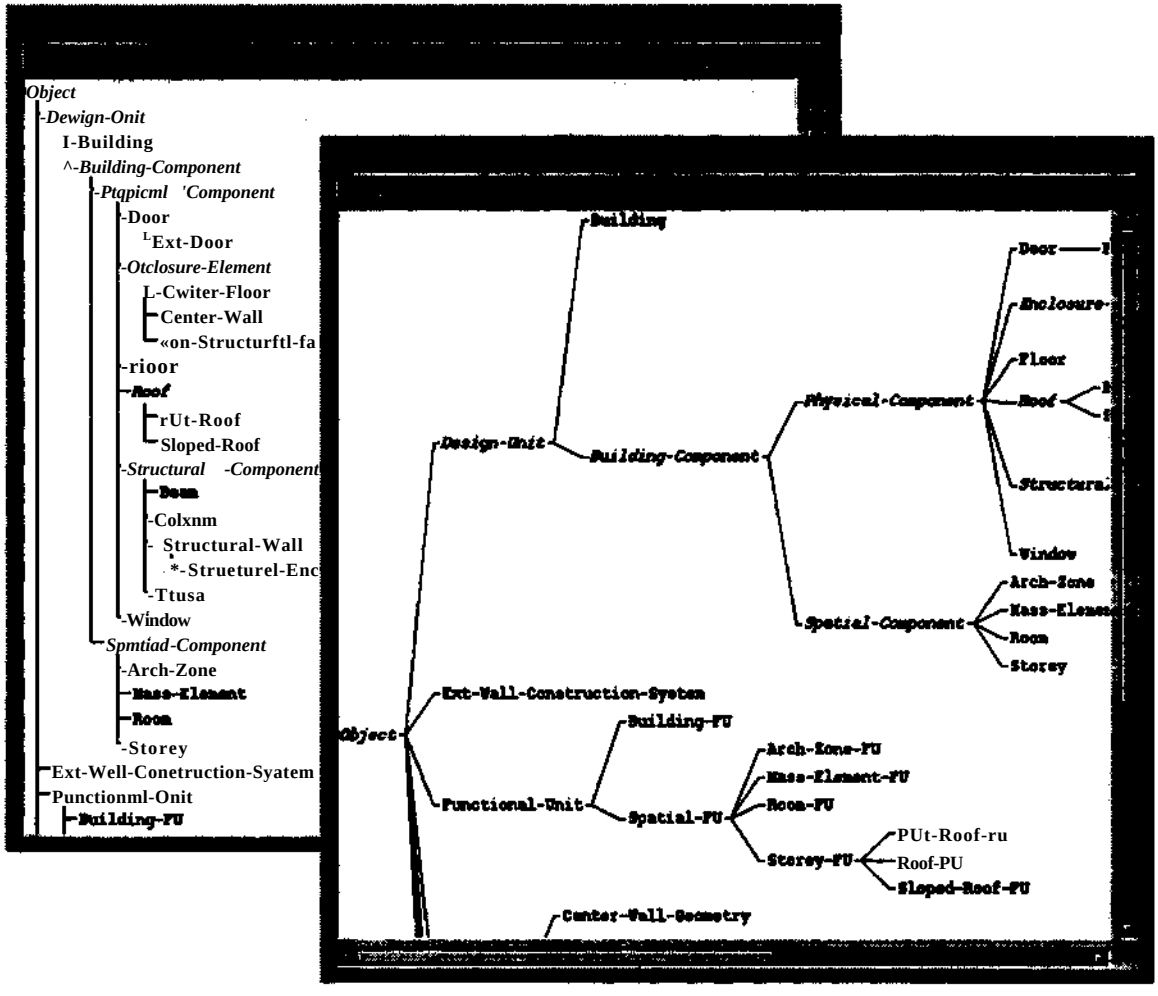


Figure 13. Hierarchy Browser in Different Views

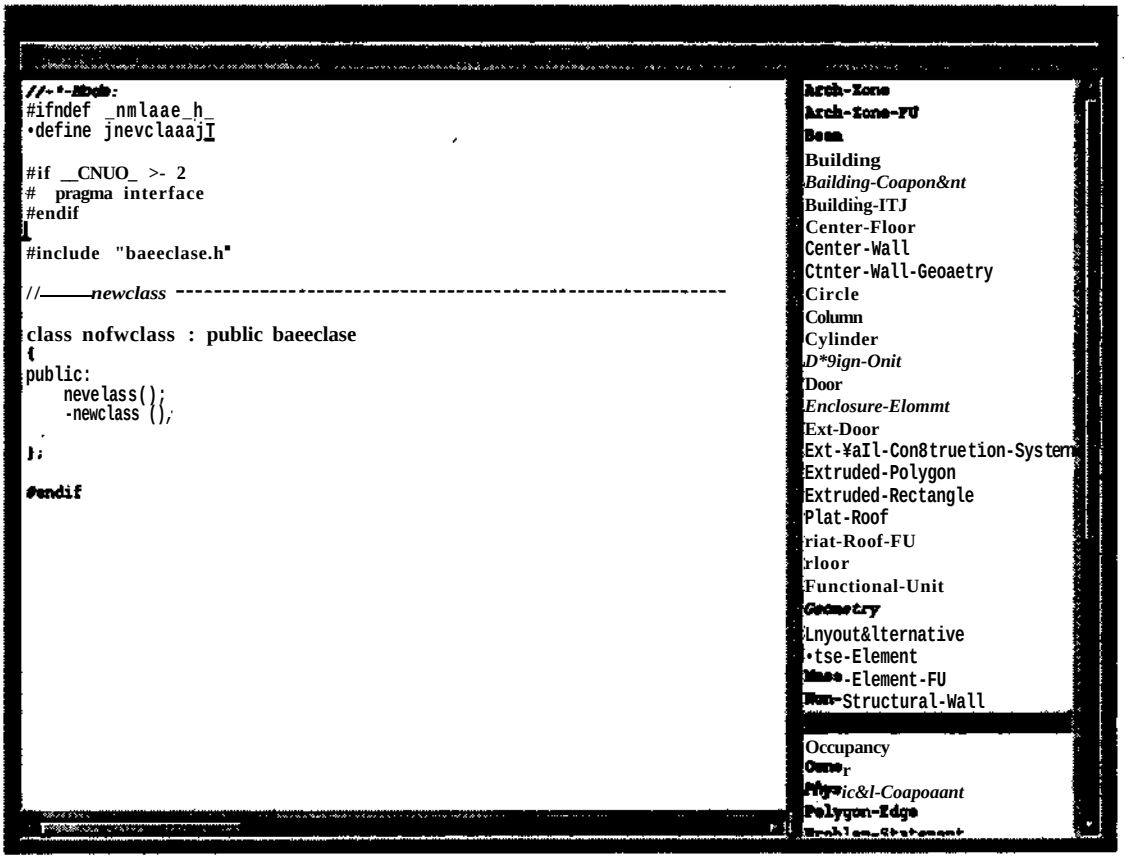


Figure 14. Source Browser

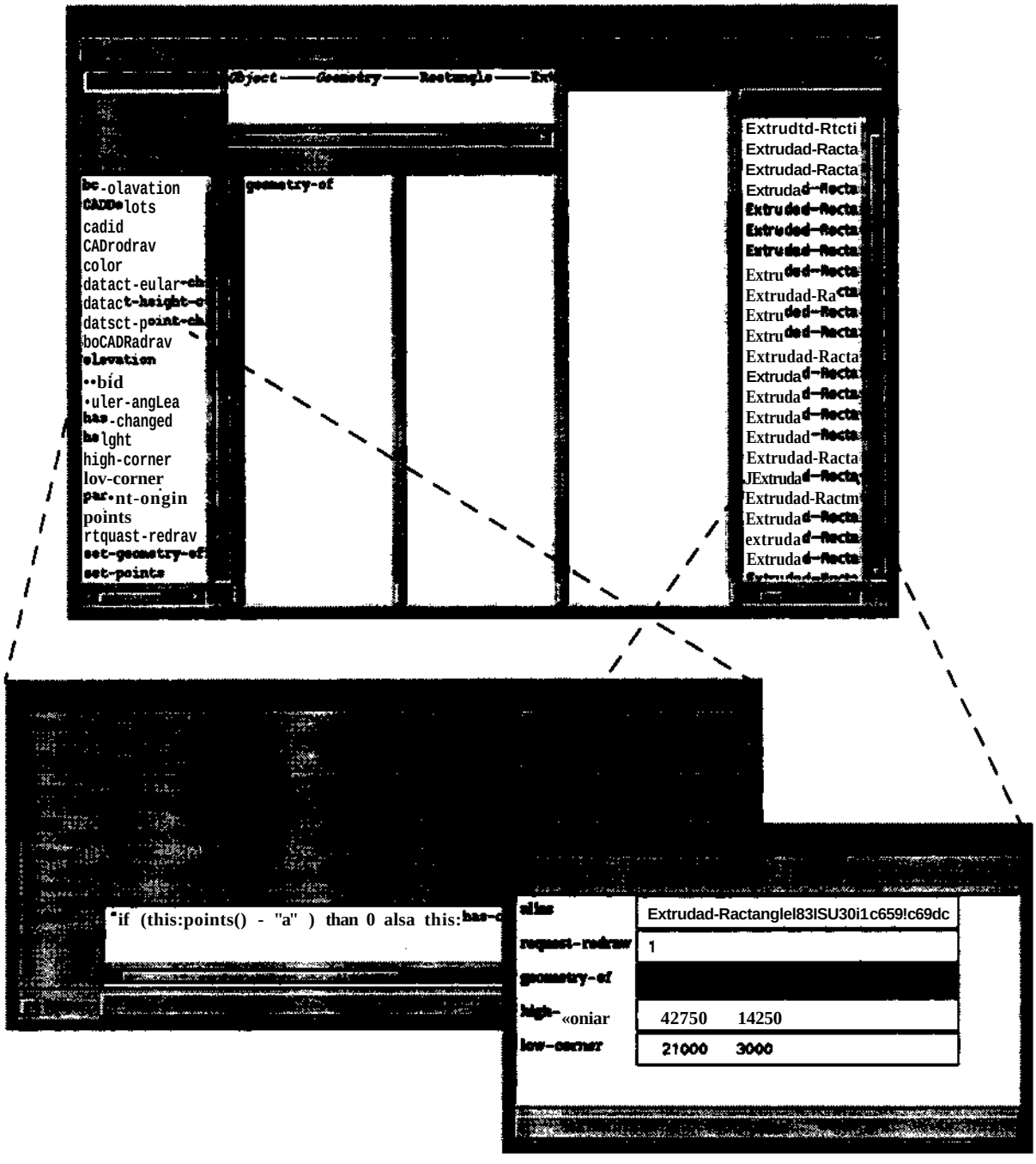


Figure 15. Class Browser

SEED-Layout

The standard SL interface supports the capabilities provided by SL for A2; this interface consists in turn of different display elements (windows, menus etc.) serving different purposes. Figure 16 shows the central "design window" from which the user can invoke most of the generative capabilities of SL; the lay-out displayed is a sublayout in the administrative wing of a firestation.

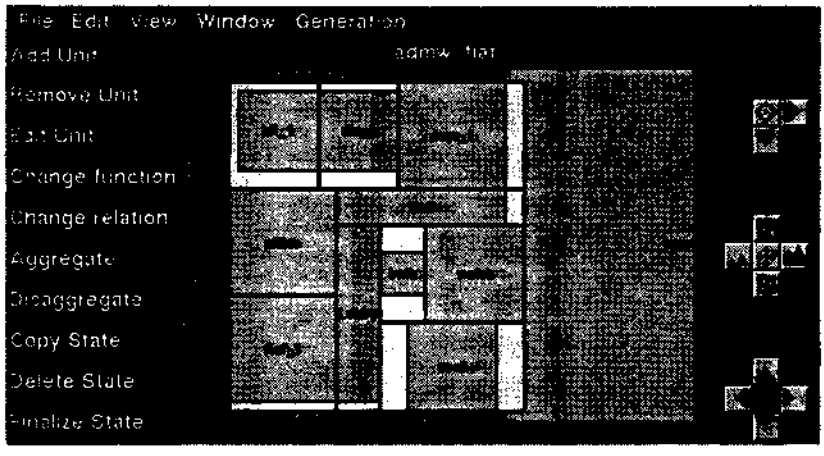


Figure 16. SL Design Window

AutoCAD Agent (ACAD)

The geometric descriptions of design components are provided by the object model expressed in OML and not derived from a traditional 'CAD engine'. As stated earlier, we use AutoCAD merely as a short-cut for 3-dimensional display purposes. Figure 17 shows as an example the display of a firestation lay-out generated by SL and communicated to A2 under the firestation scenario. The firestation contains the major functional units of a "one-company, satellite" firestation on an Army base with a central apparatus room adjacent to an administrative wing on one side and a dormitory wing on the other side. Note that SL does not deal with roof shapes. It only communicates to A2 if a roof is supposed to be sloped or flat. Based on this information, A2 determines initial shapes for sloped roofs (using simple rules-of-thumb) strictly for display purposes. These shapes will be updated once the structural design agent has broadcast its structural roof designs.

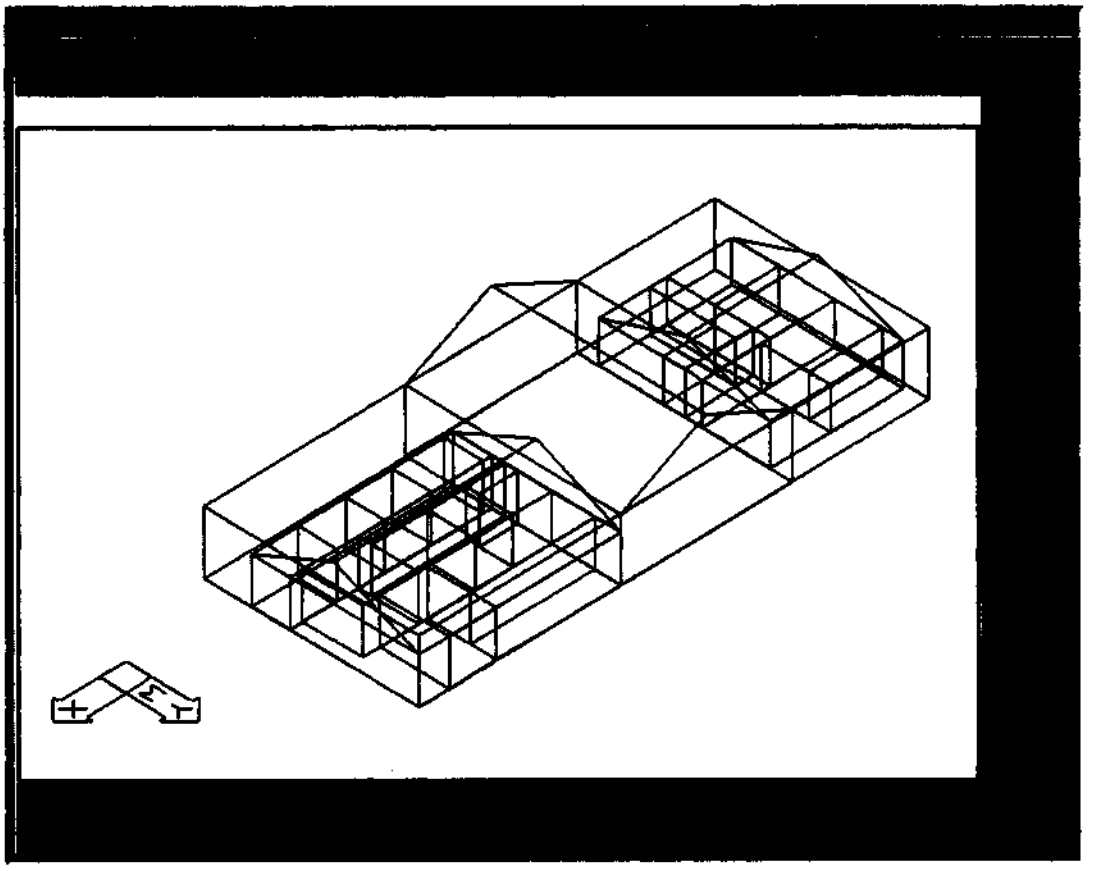


Figure 17.3-Dimensional Display in the AutoCAD Window

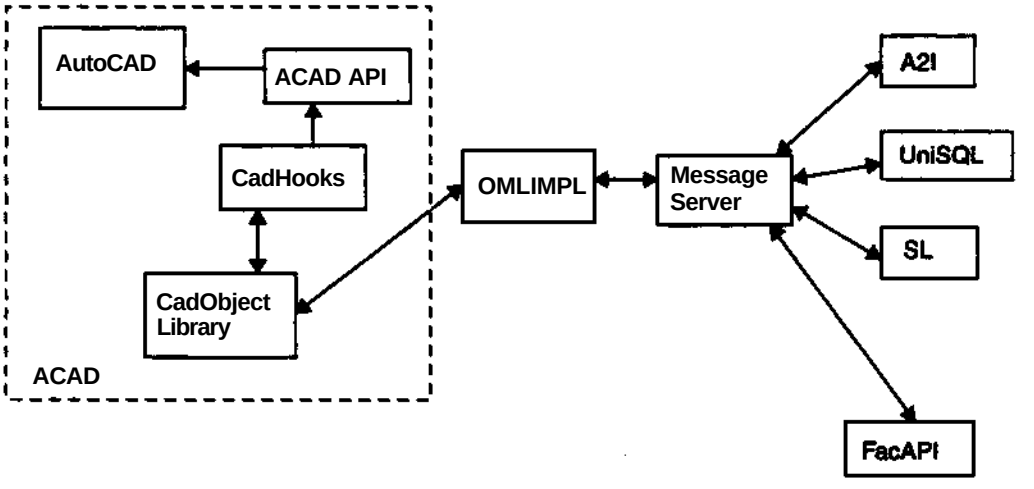


Figure 18. AutoCAD Agent Architecture

Figure 18 illustrates the components and connectors for the AutoCAD agent (ACAD) handling the 3D displays. AutoCAD receives the geometric descriptions of design components from the ACAD API which, in turn, uses AutoCAD's Advanced Modeling Extension AME to update the display of the design components. Each updated geometry is represented by a *cad object* that belongs to the AutoCAD object model and is assigned a unique identifier by ACAD API procedures. CADHooks is a component that maintains a dictionary of these objects and their cad object identifiers, which designate their AutoCAD representations within the CAD Object Library. The data translations to and from SL and AutoCAD, and consequently the mapping of a geometric description of a design component in SL to a *cad object* in AutoCAD, is performed by the object model using language binding translators.

7 CONFLICT MANAGEMENT IN THE ACL PROJECT

Basic Assumptions

A *conflict* is a disagreement between agents about a design decision made by some agent and broadcast to other agents. It can only arise because this decision violates other agents' evaluation criteria (we assume that an agent does not make, or at least does not broadcast, a decision that violates its own evaluation criteria). The disagreement can originate with a piece of evaluation software or with a human designer interacting with an agent; that is, a conflict might be asserted directly by a human designer through the agent.

This view of conflicts, together with the decentralized agent coupling adopted for the ACL project, has the following implications for conflict management and negotiation between ACL agents:

1. Conflict detection is triggered by data updates.
2. Direct negotiation between agents needs a mapping mechanism like concept communication does.
3. Since there is no central agent controlling the scheduling of agent activities, negotiation between agents has to follow a strict protocol.
4. A conflict scheme will be shared by all agents and there will be a centralized conflict management mechanism.

We assume that an OML *Conflict class* is shared by all agents as described in section "Negotiation Protocol". We also assume that the facilitator has been augmented to include the following capabilities:

1. route Conflict instances to appropriate agents according to interest and capability lists registered by the agents,
2. translate OML instances that are referred to in Conflict instances,
3. determine the list of agents interested in a Conflict instance,
4. recognize conflicts caused by OML instances and attributes classified as I/O (see the following section),
5. maintain a representation of those parts of each agent's schema that are communicated (because of 3 and 4).

Conflict management can be divided into three phases: detection, rationalization, and resolution[1]. We discuss these phases below in the context of the ACL project.

Conflict Detection

As stated above, conflicts are detected in the ACL project by individual agents. An agent incorporates externally generated design decisions into its local design representation. With help from any locally available software tools, it may then perform consistency checks, apply analysis or simulation tools, or simply display the design to the user. Any one of these or similar actions may lead to the detection of a conflict like

1. **Case, M. P. (1994) *The Discourse Model for Collaborative Engineering Design: A Distributed and Asynchronous Approach.* Ph.D. Thesis. Graduate College of the University of Illinois at Urbana-Champaign.**

spatial interference, violation of a computable design requirement, or violation of a subjective criterion evaluated intuitively by the user. That is, an agent detects conflicts by whatever method it deems appropriate.

Each attribute in an OML object can be classified into one of the four categories, *INPUT, OUTPUT, I/O* and *PRIVATE.* Through these classifications, the direction of data flow and filtration of data can be controlled at the granularity of attributes. The *PRIVATE* category specifies that an attribute should not be exposed to external agents. The *INPUT, OUTPUT,* and *I/O* categories indicate the direction of the data flow with respect to the agent who owns the OML schema; they also imply the privilege for modifying an attribute. The *OUTPUT* and *I/O* categories indicate that any agent can modify an attribute so classified; therefore, these types of attributes are subject to conflicts caused by modifications done by several agents and can be *automatically detected by the facilitator.*

Once an agent detects a conflict, it generates an instance of the conflict object and broadcasts it. It is unlikely that an agent detecting a conflict knows to whom the conflict should be addressed. This is the reason why we assumed that the facilitator has been augmented to have sufficient knowledge for determining the agents interested in a conflict.

We do not advocate for the ACL project, and building design in general, what we call the "speak now or forever hold your peace" approach toward negotiation. In this approach, a conflict is sent out to all agents together with a time limit for their response; only agents that respond in-time will be invited to the conflict resolution process. We do not take this approach because (i) agents may not have sufficient information to comment on the conflict when it is broadcast, or (ii) not all agents may be available at that time (e. g. because they are temporarily deactivated or have not even joined the team). Both reasons ultimately reflect the fact that design takes place, in principle, in an "open world": what is true at one time may become untrue later and vice-versa because the information available changes dynamically and unpredictably over the duration of a project.

Rationalization & Conflict Resolution

We choose human negotiation as the major conflict resolution strategy based on the following reasons. Not all aspects of design can be expressed in computable form, and not all design evaluations can be done by software. The knowledge needed to detect and resolve conflicts involving such evaluations is simply not available to the computer. Furthermore, a design satisfying all computable evaluation criteria is not necessarily a good design; thus, trade-offs involving non-computable criteria have to involve human judgment. Aside from these technical reasons, there are also ethical and legal reasons for leaving conflict resolution in building design to humans: the trade-offs that have to be made may influence the well-being of the occupants of the building under consideration significantly or may have legal implications for which the designers can be held accountable. They should therefore be aware of the conflicts that arise in the course of a design and take an active part in conflict negotiations.

A conflict is *resolved* if and only if all agents in the resolution process explicitly agree with a design decision that resolves the conflict. Therefore, the conflict resolution process needs an explicit agreement

strategy. If we furthermore assume that a design is satisfactory when no outstanding conflicts exist, each agent in the system should at least keep a list of the outstanding conflicts it is interested in.

Agents obviously should pursue the goal to make design decisions that do not lead to conflicts detected by other agents. However, a design without conflicts still needs to be explicitly approved by all agents to become a candidate for a final design; for example, there may be parts of the design that need further elaboration by some agent that simply has not gotten around to deal with this issue. The negotiation protocol presented below can be extended to handle this situation.

8 NEGOTIATION PROTOCOL

Scope

The proposed protocol captures the information necessary to manage conflicts generated during the course of a design project and facilitates its communication among the agents interested in the conflict. For the reasons stated in the preceding section, the automatic detection and resolution of conflicts are not supported by this protocol. However, by keeping an accurate history of a negotiation and resolution process, the proposals made, and the response of the participants, conflicts can be managed in an orderly manner.

Definitions

This section provides the definitions that are needed to describe the protocol.

Conflict Class

A *Conflict class* collects the information needed to negotiate and resolve conflicts. It consists as the following parts:

- a (potentially empty) *set of conflicting attribute descriptions* (defined in section "Attribute Conflict" below).
- an *agent set* of participating agents and their status (defined in section "Agent Status" below).
- a (potentially empty) *set of counter proposals* (defined in section "Counter Proposal" below).
- a *conflict state* (defined in section "Conflict State" below).

We assume that the facilitator has a persistent storage facility for conflict information, that the agents share the OML definition of Conflict class described below, and that the facilitator performs any translations between schema slot names and values for any conflict information.

Agent Set

The *agent set* is the set of agents capable of adding a comment to a Conflict instance.

Agent Status

The *agent status* is a response to the *definition* of a conflict in a Conflict instance and can be of the following types:

- *nojresponse* (agent has no opinion; this is the default)
- *agree* (agent agrees with the definition of the conflict)
- *disagree* (agent does not agree with the definition of the conflict)
- *dissension* (agent does not want to have the same opinion as the group)

Attribute Conflict

Attribute conflicts identify specific OML instances and slot values in a Conflict instance. Attribute conflicts are specialized into *value conflicts* and *relation conflicts.* Value conflicts target a slot value within a specific OML instance. Relation conflicts target a relationship between specific instances.

Agent Attribute Status

Each agent can identify its status with respect to each attribute conflict. The agent *attribute status* can again be of the following types:

- *nojresponse* (agent has no opinion; this is the default)
- *agree* (agent agrees with the current value of the attribute)
- *disagree* (agent does not agree with the current value, but has no recommendation)
- *proposal* (agent does not agree with the current value and has a recommendation)

Counter Proposal

If consensus cannot be reached, another Conflict instance, called a *counter proposal,* can be initiated in response to the unresolvable conflict; this creates a *tree of Conflict instances.* The parts of the counter proposal need not be the same as the initial proposal.

Counter proposals are particularly useful when the individual parts of a Conflict instance can be agreed upon, but not the whole. For example, an agreement about the height of a window and that of a wall can be reached separately, but taken together, produce a conflict from one or the other agents' perspective because the window is too high to fit into the wall. Counter proposals may create a complex structure as illustrated in Figure 18.

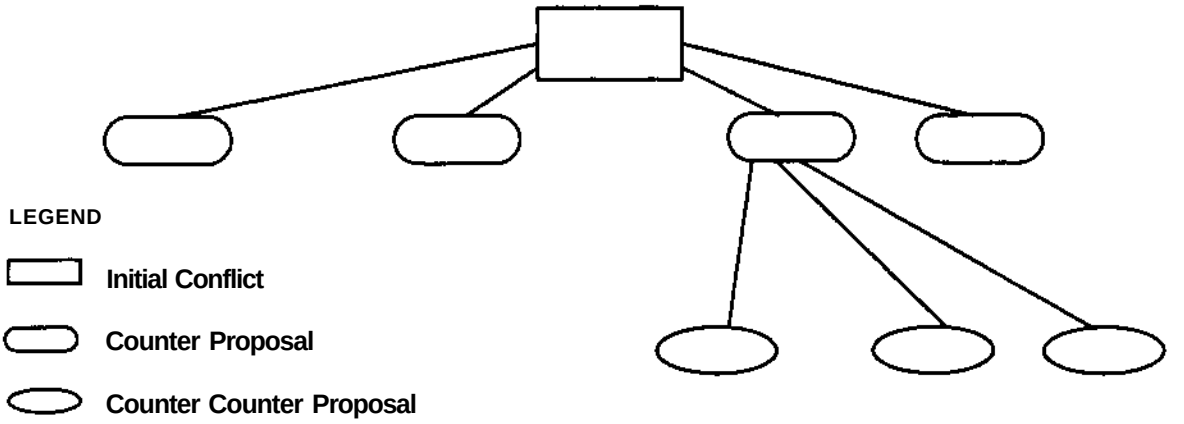


Figure 19. Example Counter Proposal Structure

ConflictState

A Conflict instance can be *OPEN, RESOLVED,* or *CLOSED.* Upon creation, the Conflict is *OPEN,* and the *resolved* attribute is FALSE. During the course of negotiation, a Conflict instance changes state to *RESOLVED* if all agents have the same agent status *(dissension* means agree *and* disagree); resolved Conflict instances can be reopened if an agent status is changed. Resolving one node in a Conflict tree makes the other nodes mute and resolves the whole tree. A resolved Conflict instance (or Conflict tree) can be closed only by the originating agent; once a Conflict instance is closed, it cannot be changed.

We expect that traditional communication methods will be used to reach consensus and closure. The proposed protocol is intended to record necessary historical information, including potential disagreements.

Protocol Specification

An OMT representation of a Conflict class is shown in Figure 19. A Conflict (Object) is an instance of that class. Note that in this representation, Conflict trees are created via the *counterproposal* relationship of the Conflict class.

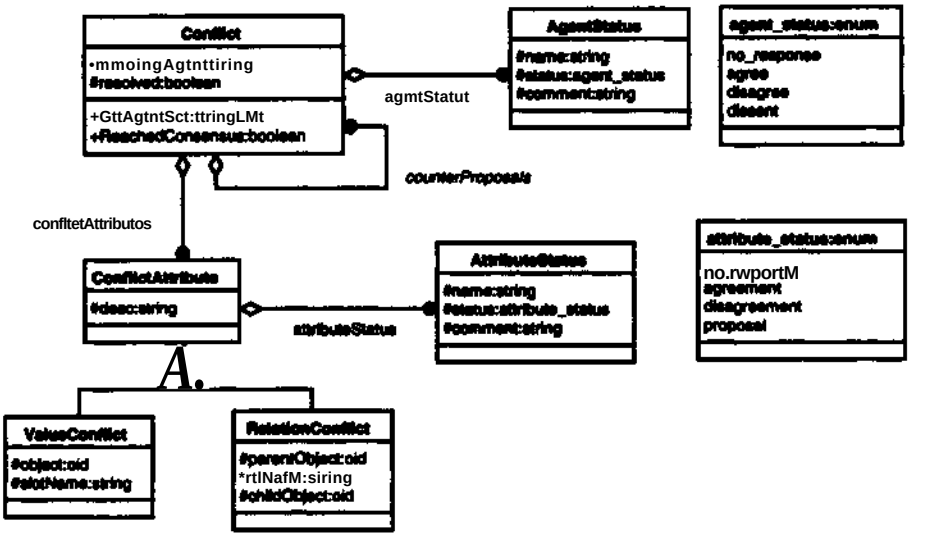


Figure 20. Conflict Composition OMT Diagram

We specify the protocol for conflict management using the state transition diagram shown in Figure 20. The member function *ReachedConsensus* searches the Conflict tree for any node that is resolved. Once a Conflict tree is resolved, the originating agent can request closure.

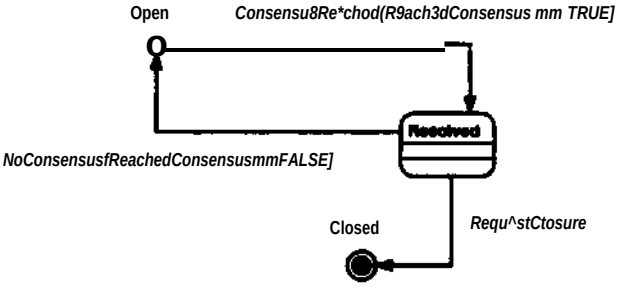


Figure 21. Conflict State Diagram

Negotiation Examples

This section uses the two negotiation scenarios from Section 3 to illustrate the negotiation protocol described above. Assume that a cabin project has been set up and that the cabin design has progressed to the point where SL has generated and committed a schematic layout. Graphical representations of the cabin layout generated by SL are presented in Figure 21 & Figure 22.

In the following examples, instances are shown in italics, slot names are given in upper case, and slot values are enclosed in quotes.

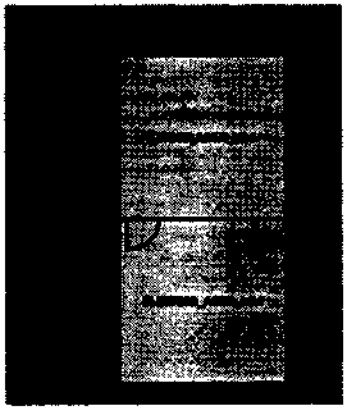


Figure 22. Cabin **Layout Generated** by SL

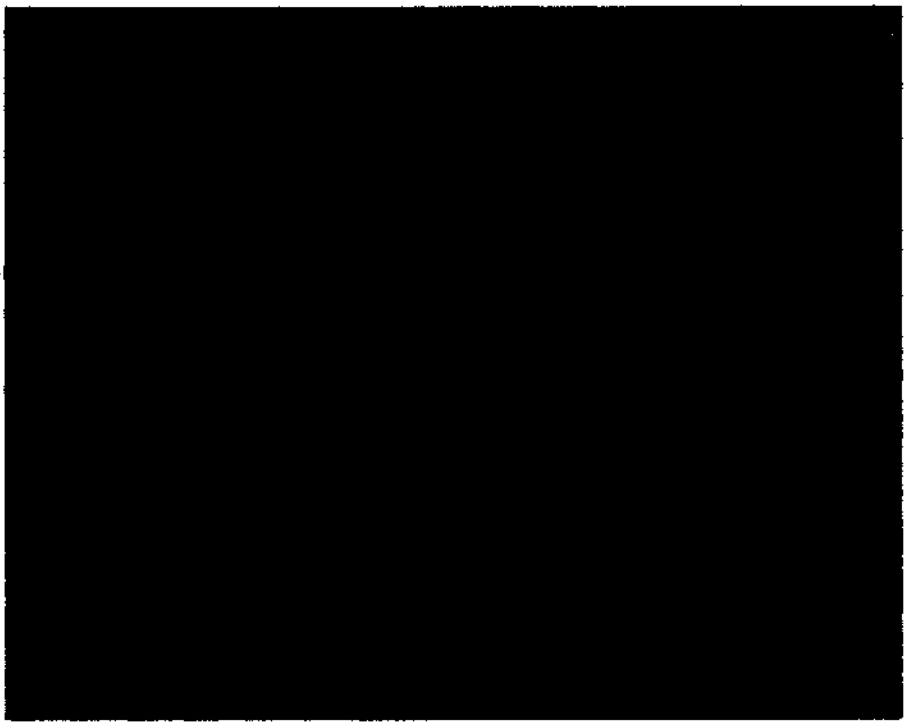


Figure 23. Cabin Displayed by ACad

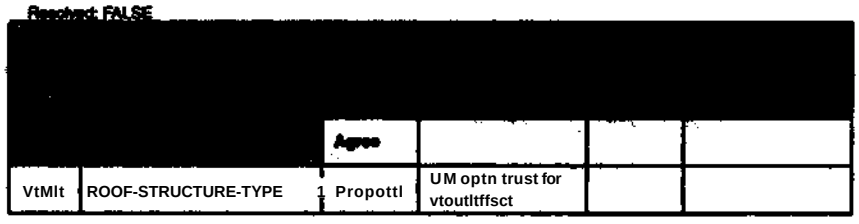
Example 1: Negotiating the type of an element

In this example, two top-level ACL agents are involved in the negotiation, the Structural Design Agent (SA) and A2. In addition, SL is involved, but does not communicate directly with the facilitator.

SA: SA receives OML instances generated by SL describing the cabin configuration.
SA generates *structural-components* based on SUs schematic layout, where the roof is supported by components of RCX)F-STRUCTURE-TYPE "joist".
SA commits the generated *structural-components.*

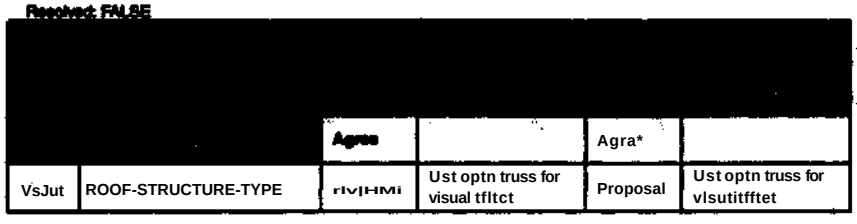
A2: A2 receives *structural-components* generated by SA and displays them.
The user of A2 does not like the look of the roof joists.
She constructs a *conflict* and sets A2's agent status to "agree" and attribute status to "proposal" with the comment, "Use open truss for visual effect". (By definition, A2 agrees with the definition of the conflict.)
The user commits *conflict.*

SA: SA receives *conflict* and notifies its user.

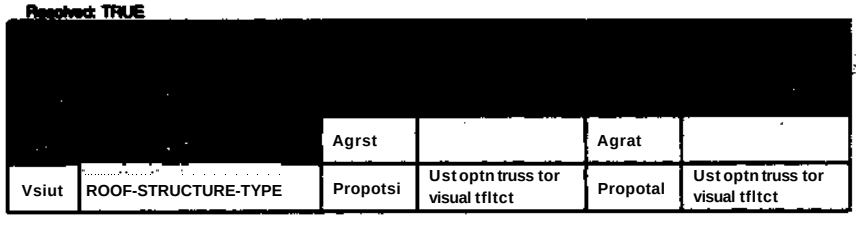


The user of SA agrees with the definition of *conflict.* He sets SA's agent status to "agree" and attribute status to be "proposal" with comment, "Use open truss for visual effect". The user of S A commits *conflict*

A2: **A2 receives *conflict* as modified by SA.**



At this moment, A2 and SA propose the same solution: the conflict is resolved. Since A2 originated the conflict, the user of A2 closes it.



SA: **The user of SA modifies the cabin structure by removing the roof joists and adding roof trusses. He commits the modified design.**

Example 2: Negotiating the placement of objects

This example continues the negotiation example above. Suppose SA generates six trusses as illustrated in Figure 23. Suppose also that an "owner agent" (OW) has joined the project by connecting to the facilitator and has been informed of all generated objects and conflicts.

Figure 24. Trusses **Generated by SA and wall generated** by SL

A2: The user of A2 discovers that *truss 1* and *walll,* a non-loadbearing partition she added to the cabin design, are not aligned.
She constructs a *conflict* with the comment, "Align Truss 1 with Wall 1", and commits it.

OW, SA: Receive *conflict* and notify their users.

Rttototcfc FALSE

		Agree					
Value	origin	Proposal	Align Truss 1 with Wall 1				

SA: The user of SA sets SA's agent status to "agree" and attribute status to "disagree" with the comment, 'Truss location is optimal for cost reasons".
SA commits *conflict.*

A2, OW: Receive *conflict* committed by SA and notify their users.

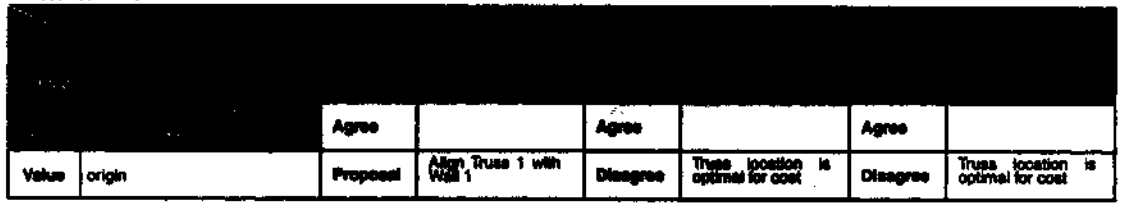
RMOtod: FALSE

		Agree		Agree			
Value	origin	Proposal	Align Truss 1 with Wall 1	Disagree	Truss location is optimal for cost		

OW: The user of OW agrees with the argument of SA (and thus disagrees with A2's proposal).
The user of OW modifies OW's agent status to "agree" and attribute status to "disagree" with comment, 'Truss location is optimal for cost reasons".
OW commits ***conflict.***

A2, SA: Receive *conflict* committed by OW and notify their users.

Resolved: FALSE



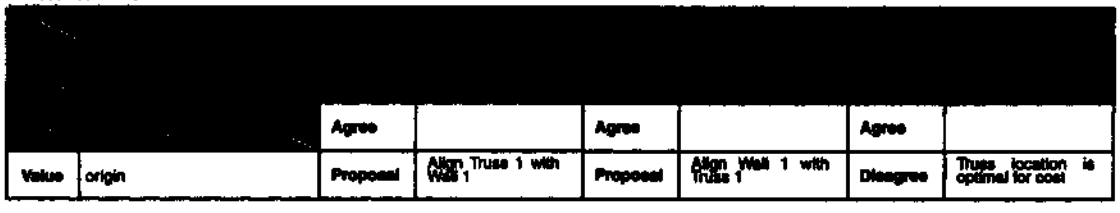
		Agree		Agree		Agree	
Value	origin	Proposal	Align Truss 1 with Wall 1	Disagree	Truss location is optimal for cost	Disagree	Truss location is optimal for cost

SA: **The user of SA modifies the attribute status to "proposal" with comment, "Align Wall 1 with Truss 1".**

SA commits *conflict*

OW, A2: Receive *conflict* and notify their users.

Resolved: FALSE



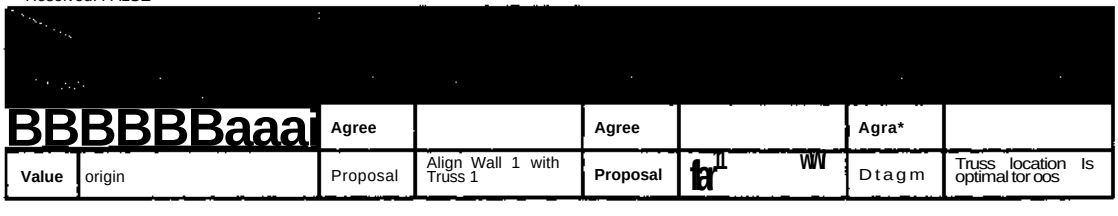
		Agree		Agree		Agree	
Value	origin	Proposal	Align Truss 1 with Wall 1	Proposal	Align Wall 1 with Truss 1	Disagree	Truss location is optimal for cost

A2: **The user of A2 agrees with the proposal and modifies A2*s attribute status to "proposal'[1]' and comment to "Align Wall 1 with Truss 1".**

The user of A2 commits the modified *conflict.*

SA, OW: Receive the updated *conflict* generated by A2 and notify their users.

Resolved: FALSE

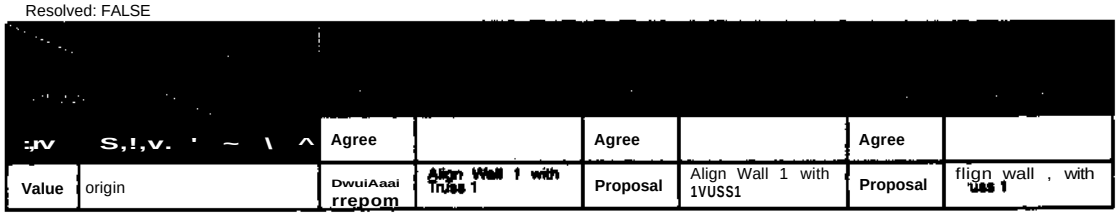


BBBBBBaaai		Agree		Agree		Agra*	
Value	origin	Proposal	Align Wall 1 with Truss 1	Proposal	fa[1] WW	Dtagm	Truss location Is optimal tor oos

OW: **The user of OW modifies the attribute status to "proposal" with comment, "Align Wall 1 with Truss 1".**

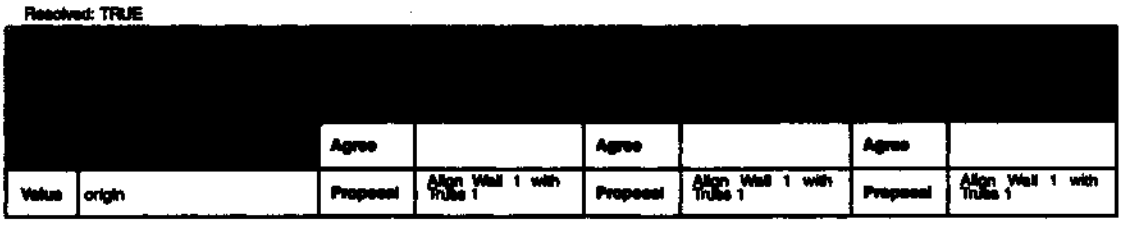
SA, A2: Receive the updated *conflict* and notify their users.

Resolved: FALSE



:,v S,!,v. ' ~ \ ^		Agree		Agree		Agree	
Value	origin	DwuiAaai rrepom	Align Wall 1 with Truss 1	Proposal	Align Wall 1 with 1VUSS1	Proposal	flign wall , with 'uss 1

A2: **The user of A2 announces that *conflict* is closed.**



A2: **The user of A2 aligns *walll* with *trussl* and commits the newly modified design. Since a *door* is attached to *walll,* the *door's* origin is also updated (we assume that the *door's* origin is not relative to *walll).***

SA, OW: Receive the modified design and synchronize their internal design representation with the newly received OML instances.

9 DISCUSSION

The CMU team's observations are grouped below under two headings: project aspects that we deem successful, and lessons learned from less successful aspects.

Successful Project Aspects

We believe that to the project's credit, it chose to address "difficult" issues even though supporting software technologies may not have existed. For example, teams at MIT, UIUC and CMU developed applications with complex internal representations and separate databases, which constitute, as stated earlier, a distributed product model. Software to integrate such applications does not exist, but if distributed collaboration is to succeed in building design, it must be able to handle collaboration between applications of this type. Furthermore, we arrived - as a team - at a clear understanding of what the information exchange and negotiation problems in such a set-up are. In addition, the domain-specific computational models produced by individual teams were innovative and interesting. In particular, MIT's structural agent provides an elegant and comprehensive solution to modeling structural design knowledge.

We also stress benefits derived from the interdisciplinary nature of the project. We were able to identify scenarios for the conceptual phases of building design that result in a less linear design process across the different disciplines. The CMU team was also able to communicate to other teams that architects don't "just draw lines" (as someone put it in an early meeting). We believe that the communication of this kind of information was greatly facilitated by using the World-Wide Web as a document distribution mechanism.

The CMU-developed schema specification tools provided the basis for progress and communication of the team as a whole. As the preceding sections show, we were also able to integrate our own components successfully using the same tools. The language binding compiler enabled us to exchange information reliably between existing applications where other tools would have failed.

Lessons Learned

During the course of the project, we were able to communicate some information between sites, but the system as a whole was never complete or tested. Use cases depending on communication with other agents could therefore not be implemented; this applies to all use cases dealing with design components or evaluations generated by other agents or depending on the existence of such objects. We stress, however, that an implementation would be very easy if the needed information were to become available because the OML model for the objects of interest has been implemented. Parallel experiments between the CMU and MIT teams provided essential verification of the OML approach to schema specification and language binding. However, this experience could not be transferred to the larger project for the reasons stated below.

From an infrastructure perspective, the Internet posed problems: it does not yet appear suitable for the type of serious collaboration aimed at in this project. It is still not as stable as a Wide-Area Network (WAN) or a Local-Area Network (LAN). In addition, teams working in different time zones pose difficult synchronization problems.

The Federation Architecture was not able to support the integration of applications of the given type over the Internet. We offer several reasons for this:

1. **We never knew if information was, in fact, exchanged correctly because we could not verify translation rules or data structure mappings in an application.**
2. **The Federation Architecture did not provide any tools for object-oriented information modeling. Some researchers claim that it *should not* have such tools, but it is clear to us that for the given applications, which represent state-of-the-art object-based modeling techniques, such tools are essential.**
3. **We also observed that the needs of the teams that developed agents (i.e. clients) were not taken into account when services were provided via the facilitator. In particular, we believe the emphasis on object-centered representations was neglected until very late in the project.**
4. **We need rigorous formalisms for schema concept mapping based on which translation code can be generated automatically: manually generated translation rules proved deficient; we need *software* to generate translation rules automatically. This became apparent during a final project meeting at Stanford on Dec. 20,1996. For example, it was very easy to generate manually a rule that can "loop" (i.e. fire multiple times for a given data set), and these kinds of errors were introduced even in very simple translations. This only confirms the experience with several decades of expert systems development: they are brittle and difficult to verify.**

REFERENCES

Akin, <X Sen, R., Donia, M. and Zhang, Y. (1995) "SEED-Pro: Computer-Assisted Architectural Programming in SEED" *Journal ofArchitectural Engineering,* vol. 1 pp. 153-16

Case, M. P. (1994) *The Discourse Model for Collaborative Engineering Design: A Distributed and Asynchronous Approach.* Ph.D. Thesis. Graduate College of the University of Illinois at Urbana-Champaign.

Chiou, J. and Logcher, R. D. (1996) Testing a Federation Architecture in Collaborative Design Process. Res. Report R96-01. Dept. of Civil and Environmental Engineering, MIT, Cambridge, MA

Flemming, U., Coyne, R., and Chien, S.F. (1994) *SEED-Layout Requirements Analysis.* http://seed.edrc.cmu.edu/SL/SL-start.book.html.

Flemming, U. and Chien, S.F (1995) "Schematic Layout Design in SEED Environment" *Journal of Architectural Engineering,* vol. 1 162-169

Flemming, U. and Woodbury, R. (1995) "Software Environment to Support Early Phases in Building Design (SEED): Overview" *Journal of Architectural Engineering,* vol. 1 pp. 147-152

Gamma, E., Helm,R., Johnson, R. & Vlissides, J. (1994) *Design Patterns - Object Oriented Software.* Reading, Massachusetts. Addison-Wesley

Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach.* New York: Addison-Wesley

Khedro, T., Case, M. P., Flemming, U., Genesereth, M. R., Logcher, R., Pedersen, C, Snyder, J., Sriram, R. D., Teicholz, P. M. (1995). "Development of a Multi-Institutional Testbed for Collaborative Facility Engineering Infrastructure". In J. P. Moshen (Ed.) *Proceedings of the Second Congress on Computing in Civil Engineering,* pp 1308-1315

McGraw, K. *(1996) Agent Collaboration Environment (ACE).* http://www.cecer.army.mil/pl/ace/.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-Oriented Modeling and Design.* Englewood Cliffs: Prentice-Hall.

Snyder, J. and Flemming, U. (1994) *ACL Object Model Specification Language,* report accessible from http://seed.edrc.cmu.edu/ACL/ObjModelAPI/objapi.book.html.

Wcinand, A. and G. Gamma. (1995) £7++ - *a portable, homogenous class library and application framework.* Taligcnt Inc. Cupertino, CA