Knowledge-Based Computer-Aided Architectural Design
G. Carrara and Y.E. Kalay (Editors)
1994 Elsevier Science B.V.

7

A DISTRIBUTED COOPERATIVE MODEL FOR ARCHITECTURAL DESIGN

Jens Pohl
Architecture Department
California Polytechnic State University
San Luis Obispo, California

Leonard Myers
Computer Science Department
CAD Research Center
California Polytechnic State University
San Luis Obispo, California

Meaningful support of the architectural design activity in a computer-based environment has proven to be a much more difficult and elusive undertaking than first anticipated. The reasons are related not only to the ill-defined nature of the activity [Rittel and Webber 1984, Simon 1984], but also to the inadequacies of the representational and operational models that have been used as the framework for computerization.

Simon [1981] has defined design as an intellectual activity with the objective of changing an existing situation into a preferred one. In the fields of architecture and engineering this comprehensive definition can be narrowed to apply to the production of an artifact. Since much has been written in the past two decades about how architects design [Cross 1984, Schon 1983, Mackinder and Marvin 1982, Akin 1978, Wade 1977], it is not the intention of the authors to reiterate and debate the often subtle differences among these descriptions.

Rather, our purpose is to identify the more obvious features and derive from these an appropriate computer-based model. We start by characterizing the design activity in terms of five functional elements: information, representation, visualization, reasoning, and intuition.

INFORMATION

Design is a cooperative activity involving many sources of information that are often widely dispersed. Typically, the problem specifications evolve with the problem solution as the designer interacts with the environment. Accordingly, the information requirements of the designer are not predictable since the information needed to solve the problem depends largely on the solution strategy adopted [Fischer and Nakakoji 1991]. In this respect design is a learning process in which the designer progressively develops a clearer understanding of the problems that are required to be solved.

Much of the information that designers use in the development of a design solution is gleaned from experience with past projects. In fact, it has been argued that design solutions commonly evolve out of the adaptation, refinement and combination of prototypes [Gero et al. 1988]. Prototypical knowledge can be categorized into several types [Pohl et al. 1992].

> Vertical prototype knowledgebases that contain typical object descriptions and relationships for a complete artifact type. Such a knowledgebase may include all of the types that exist in a particular problem setting: for example, a building type such as a library, bank branch, courthouse or supermarket (Figure 7.1).
>
> Horizontal prototype knowledgebases that contain typical solutions for sub-problems, such as waterproofing considerations and conventions for windows or other constructional practices (Figure 7.2). Such knowledge certainly transcends artifact types within the same design discipline, and often bridges design disciplines. For example, the waterproofing techniques developed in the automobile industry for car windows may apply equally to building windows.
>
> Domain prototype knowledgebases that contain guidelines for developing solutions within contributing narrow domains. For example, the range of structural solutions appropriate for a 20-story office building in Los Angeles is greatly influenced by the seismic character of that region. Posed with this design problem structural engineers will immediately draw upon a set of rules that guide the design activity.

Exemplary prototype knowledgebases that describe a specific instance of an artifact type or solution to a sub-problem. Exemplary prototypes can be instances of vertical or horizontal prototypes, such as a particular library building or a specific entrance in an existing building. Architects often refer to exemplary prototypes in exploring solution alternatives to sub-problems.

Experiential knowledgebases that represent the factual prescriptions, strategies and solution conventions employed by the designer in solving similar kinds of design problems. Such knowledgebases are typically rich in methods and procedures.

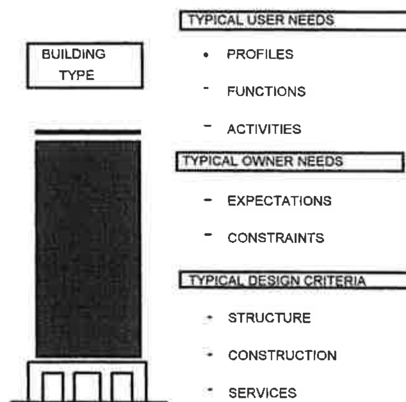

Figure 7.1. Building type knowledge

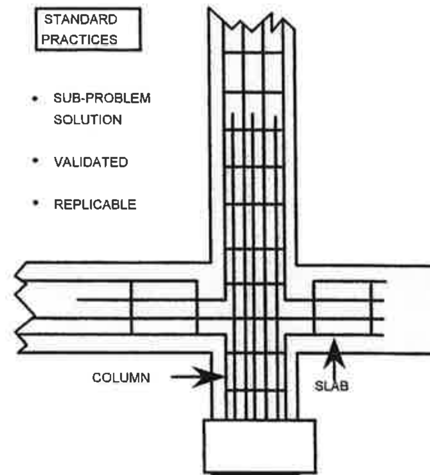

Figure 7.2. Standard practices knowledge

The volume of this information is potentially overwhelming. However, particularly the more astute and experienced designers will insist on taking their time to assimilate as much information as possible into the problem setting before committing to a solution theme. Much of the information cannot be specifically structured and prepared for ready access, because the needs of the designer cannot be anticipated. Every step toward a solution generates new problems and information needs [Simon 1981].

REPRESENTATION

The methods and procedures that designers utilize to solve design problems rely heavily on their ability to identify, understand and manipulate objects. In this

respect, objects are complex symbols that convey meaning by virtue of the explicit and implicit information that they encapsulate within their domain. For example, architects develop design solutions by reasoning about neighborhoods, sites, buildings, floors, spaces, walls, windows, doors, and so on. Each of these objects encapsulates knowledge about its own nature, its relationships with other objects, its behavior within a given environment, what it requires to meet its own performance objectives, and how it might be manipulated by the designer within a given design problem scenario. This knowledge is contained in the various representational forms of the object as factual data, algorithms, rules, exemplar solutions, and prototypes.

The reliance on object representations in reasoning endeavors is deeply rooted in the innately associative nature of the human cognitive system. Information is stored in long term memory through an indexing system that relies heavily on the forging of association paths. These paths relate not only information that collectively describes the meaning of symbols such as 'house', 'fish' and 'car', but also connect one symbol to another. Symbols are not restricted to the representation of physical objects, but also serve as concept builders. They provide a means for grouping and associating large bodies of information under a single conceptual metaphor. In fact, Lakoff and Johnson [1980] argue that "...our ordinary conceptual system, in terms of which we both think and act, is fundamentally metaphorical in nature". They refer to the influence of various types of metaphorical concepts, such as 'desirable is up' (spatial metaphors) and 'fight inflation' (ontological or human experience metaphors), on the way human beings select and communicate strategies for dealing with every day events.

Designers typically intertwine the factually based aspects of objects with the less precise, but implicitly richer language of metaphorical concepts. This leads to the spontaneous linkage of essentially different objects through the process of analogy. In other words, the designer recognizes similarities between two or more sub-components of apparently unrelated objects and embarks upon an exploration of the discovered object seeking analogies where they may or may not exist. At times these seemingly frivolous pursuits lead to surprising and useful solutions of the design problem at hand.

VISUALIZATION

Designers use various visualization media, such as visual imagination, drawings and physical models, to communicate the current state of the evolving design solution to themselves and others. Drawings, in particular, have become intrinsically associated with the design activity. In this sense the drawing is

somewhat analogous to a game board on which the designer is able to freely experiment with spatial shapes and relationships (Figure 7.3).

The historical preoccupation of architects with visual images in physical media can easily lead to the erroneous conclusion that the act of drawing is the principal element of the design activity, and that this activity cannot proceed without the drawing medium. In fact, drawings are information poor. They are largely limited to the representation of geometric, textural and spatial information. The addition of labels and text provides only a modicum of semantic information. Yet, the reasons that underlie the artifact that is represented by the drawing are at least as important as the image in the drawing. The drawing by itself cannot adequately convey the intent of the designer when subsequent changes have to be made in the design solution by other members of the team. Under those circumstances the unavailability of the designer often results in misinterpretation and inappropriate conclusions that lead to costly, and sometimes dangerous, mistakes.

Although the designer can reason about design problems solely through mental processes, drawings and related physical images are useful and convenient for extending those processes. The failings of the drawing as a vehicle for communicating design intent do not apply to the creator of the drawing. To the designer the drawing serves not only as an extension of long term memory, but also as a visual bridge to its associative indexing structure. In this way, every meaningful part of the drawing is linked to related data and deliberation sequences that together provide an effectively integrated and comprehensive representation of the artifact.

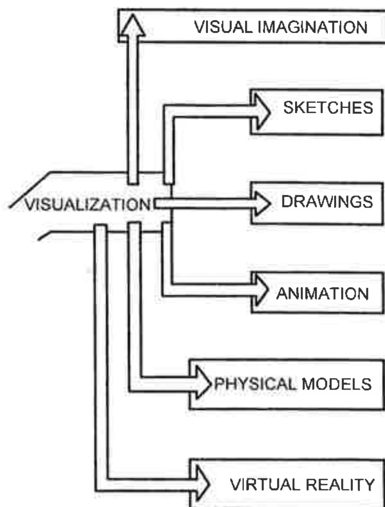

Figure 7.3. Design Activity: Visualization

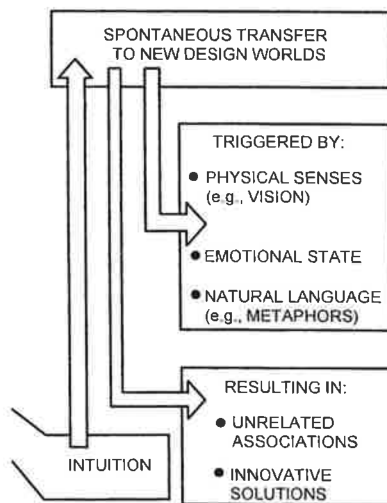

Figure 7.4. Design Activity: Intuition

REASONING

Reasoning is central to the design activity. It is the ability to draw deductions and inferences from information within a problem solving context. The ability of the designer to reason effectively depends as much on the availability of information, as it does on an appropriately high level form of object representation.

Designers typically define the design problem in terms of issues that are known to impact the desired outcome. The relative importance of these issues and their relationships to each other change dynamically during the design process. So also do the boundaries of the design space and the goals and objectives of the desired outcome. In other words, design is an altogether dynamic process in which both the rules that govern the process and the required properties of the end result are subject to continuous review, refinement and amendment [Reitman 1964 and 1965, Rittel and Weber 1984].

The complexity of design does not appear to be due to a high level of difficulty in any one area but the multiple relationships that exist among the many issues that impact the desired outcome. Since a decision in one area will tend to influence several other areas there is a critical need for concurrence. However, the reasoning capabilities of the designer are sequential in nature. Accordingly, designers find it exceedingly difficult to consider more than three or four issues at any one time. In an attempt to deal with the concurrence requirement they employ several strategies to reduce the complexity of the reasoning process to a manageable level:

CONSTRAINTS IDENTIFICATION: By sifting through the available information the designer hopes to find overriding restrictions and limitations that will eliminate knowledge areas from consideration during the design process.

WEIGHTED DESIGN FACTORS: By comparing and evaluating design factors in logical groupings, relative to a set of predetermined solution objectives, the designer hopes to identify a smaller number of factors that have greater impact on the final design solution. Again, the strategy is to reduce the size of the information base by early elimination of apparently unimportant considerations.

SOLUTION CONCEPTUALIZATION: By adopting, early in the design process, a conceptual solution the designer is able to pursue a selective evaluation of the design information. Typically, the designer proceeds to subdivide the design factors into two groups; those that are compatible with the conceptual solution and those that are in conflict. By a process of trial

and error, often at a superficial level, the designer evolves, adapts, modifies, rejects, reconceives and, often, forces the preconceived concept into a final solution.

Although reasoning is an essential functional component of the design activity, it would be misleading to suggest that design is performed in an entirely methodical manner as a set of sequential transformations.

Schon [1988] has drawn attention to four fundamental tensions in design. First, designers typically have difficulty articulating the knowledge and methods that they apply to the design function. While the design solutions that they produce are often ingenious, the subsequent explanations of the designer are less convincing. If the knowledge the designer holds cannot be made explicit then what kind of knowledge is it, how is it retained, and how can it be accessed when needed? Second, there is the apparent paradox between the designer's quest for a unique solution and the requirement of general rules to reason out that solution. Third, designers accumulate knowledge from one project to the next. If they apply this prototype knowledge derived from past projects to produce design solutions, how can they ever generate new prototypes? Fourth, architecture and engineering design is a team effort. The team members have pluralistic backgrounds, interests, and agendas. Yet, they normally agree on a common design solution.

While designers will employ sequential reasoning in short bursts and over somewhat longer periods to explore and evaluate solution alternatives, the generation of the alternatives themselves is often neither sequential nor logical. The latter is characterized by the spontaneous introduction of apparently unrelated thoughts, associations that transcend purely logical relationships, whimsy, and tacit understandings that defy explanation. This characteristic of the human cognitive system is often referred to as intuition.

INTUITION

In Schon's [1988] view designers enter into 'design worlds' in which they find the objects, rules and prototype knowledge that they apply to the design problem under consideration. The implication is that the designer continuously moves in and out of design worlds that are triggered by internal and external stimuli. While the reasoning process employed by the designer in any particular design world is typically sequential and explicitly logical, the transitions from state to state are governed by deeper physiological and psychological causes. Some of these causes can be explained in terms of associations that the designer perceives between an aspect or element of the current state of the design solution and prototype knowledge that the designer has accumulated. Others

may be related to emotional states or environmental stimuli, or interactions of both (Figure 7.4).

For example, while engaged in the act of drawing a possible building outline on a site plan the designer may spontaneously perceive an alternative interpretation of the building footprint as a crocodile. In this case the trigger stimulus is a spatial gestalt that moves the designer momentarily into a new design world, complete with its own types and rules. As another example, the noise and vibration from a passing truck may invoke an emotional response which focuses the designer's attention on the exposure of a contemplated building entrance. This in turn could trigger any number of reference types or experiential archetypes flashing through the conscious thought process of the designer.

Obviously, the number, combinations and permutations of design worlds that the designer may enter into is infinite and largely unpredictable. Of more significance is the fact that these design world transitions enable the designer to leapfrog during the design activity, and establish associations among multiple seemingly unrelated knowledge domains. While the knowledge content of the design space dynamically changes as the designer moves in and out of design worlds, it can be assumed that these changes are not pervasive. As the designer approaches an acceptable solution increasingly larger areas of the design space will become more and more static.

This view of the designer moving through multiple design worlds provides a plausible explanation of a number of apparently conflicting characteristics of the design activity that have been identified in the literature [Lawson 1988, Archea 1987, Schon 1983 and 1988, Rowe 1987, Akin 1986, Kirk and Spreckelmeyer 1988, Broadbent 1979]. For example:

> Designers tend to redefine both the problem specifications and the solution objectives during the design activity, because the design space dynamically changes as the designer moves from one design world to another. Each design world provides the designer with a different view of the design space and an opportunity to modify both the problem definitions and the solution expectations.

> There are no optimal solutions to design problems because the designer has no static benchmark for evaluating the worthiness of a design solution beyond satisfying a set of performance requirements. As he moves from one design world to another his interpretation of these requirements may change significantly, and this in turn will also affect his understanding of the relative importance of each requirement.

> As a corollary it is of interest to note that optimization is not relevant to design solutions. This is entirely compatible with Schon's characterization of design as a "...kind of making" [Schon 1988]. While the designer may apply optimization procedures to a particular sub-problem, the fitting of the pieces together into a coherent design solution is the personal contribution of the designer. A contribution that is based on associations made during design world transitions rather than rational reasoning.

It follows that mathematical techniques and formal reasoning processes are useful tools within design worlds where they assist the designer in exploring, experimenting and evaluating narrow domains. However, they are unlikely to play any significant role in the transfer from one design world to another. Those transitions are governed by physiological and psychological responses to a wide range of stimuli that often enter the consciousness of the designer only after the fact.

EXISTING CAD SYSTEMS

Commercial CAD software has evolved very gradually over the past 25 years from an initial singular focus on drawing automation. Accordingly, even today, in the majority of architectural offices CAD stations are still employed as largely stand-alone units to serve the specific purpose of drawing production and manipulation. Conceived in this limited application realm, CAD-drawing packages have been implemented as large programs that typically encompass all of their functionality within a single process (Figure 7.5).

The first significant departure from this centralized approach came with the realization that the value of drawings can be greatly increased if the objects that they contain are directly linked to the information that they represent (i.e., functions, relationships to other objects, materials, cost, ownership, availability, etc.,). While this led initially to the addition of attribute files (e.g., as symbol libraries) and spreadsheets within the same CAD program, it was soon recognized that the need for a great deal of data could not be accommodated efficiently with this centralized approach. Particularly the asset management requirements of large companies and government departments created a need for more substantial and better organized database systems (Figure 7.6).

The provision of two-way linkages between relational database management systems and drawing and solid modelling packages has not been entirely successful in addressing these needs. Unfortunately, the internal structure of the CAD program did not lend itself well to external connections. Since the CAD package had been designed as a large self-contained program it

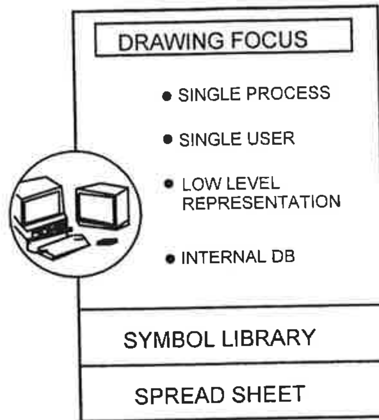

Figure 7.5. 1st Generation CAD Systems

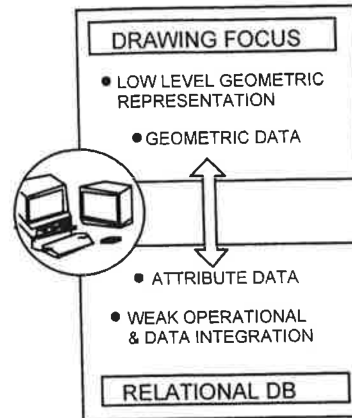

Figure 7.6. 2nd generation CAD Systems

was difficult to find convenient data entry and exit points without significant consequences to the operational flow of the package. As a result, even today most of these CAD-database linkages are time consuming, cumbersome, and functionally constrained.

Attempts to integrate the geometric representations of the CAD program with the data structures employed by the external databases were only partially successful. Attempts to store the geometry of the drawing together with the non-geometric attributes in the relational database failed. Relational databases are optimized to process large numbers of short transactions and are therefore unable to accommodate the much longer graphic transactions required in the CAD environment, within reasonable response time limits. The alternative solution, to store the geometry separately from the attributes and link the two with embedded pointers, has imposed another layer of complexity on an already contrived connection.

The addition of virtually unlimited amounts of attribute information to drawing objects has provided an opportunity to incorporate decision making assistance in the CAD environment. However, for several reasons progress in this promising area has been disappointingly slow. First, the geometric descriptions created by current CAD systems are essentially limited to points, lines, faces and shapes. A description that is useful for reasoning purposes should represent the geometry of the artifact in terms of geometric objects that have meaning in the real world in which the artifact will be constructed and used. In other words, for a CAD system to support the design activity it must contain high level knowledge about the evolving design solution and its context (Figure 7.7).

While it is quite difficult to generate higher level information from lower level information, the reverse is generally fairly straight forward. Given the high level knowledge that an object is a wall, it is relatively easy to generate the coordinates of any point on the wall based on the higher level description and a few default assumptions. To determine on the basis of points, lines and surface parameters that the object is a wall with particular characteristics (e.g., lining thicknesses and structural frame configuration), is a much more difficult undertaking.

Second, it has been difficult to link reasoning agents to CAD programs that are designed for single-tasking environments and are highly demanding of the available resources in their execution environment. Accordingly, initial commercial attempts have been limited to the incorporation of single expert systems in mostly the in-house CAD systems of large engineering, aerospace, and construction firms. Although such reasoning agents have been successful their usefulness has been restricted to sub-problems, such as staircase construction, welding, lighting, and thermal analysis [Lenart and Ketteler 1991, Kim and Degelman 1990].

Third, when these early explorations led to the conclusion that effective design decision support will require, at the very least, multiple agents interacting within an integrated CAD environment, no suitable cooperative model was found to be readily available. Only a few appeared to exist, mostly in university computer science units, as engines for theoretical research endeavors.

In this seemingly dismal state of affairs there are, however, some rays of light. The fierce competition that has prevailed in the CAD industry during the

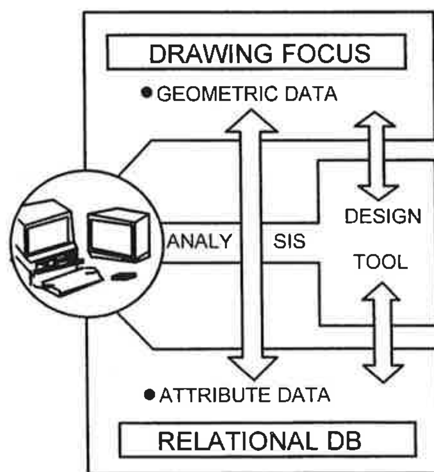

Figure 7.7. Single Agent Design Analysis

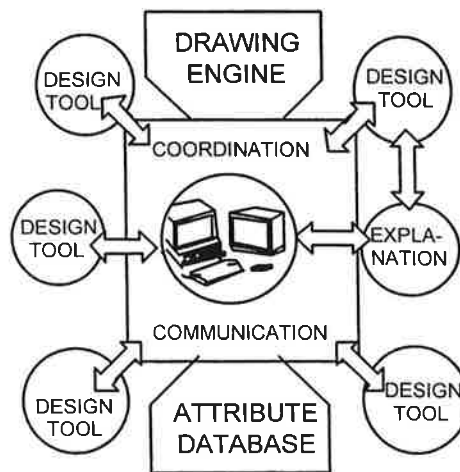

Figure 7.8. Multiple Agent Design Analysis

early 1990s is forcing an increasing number of CAD vendors to branch into new application fields. By necessity, this has resulted in the dismantling of some large single-tasking CAD programs into many small modules that can be used as toolkits for customizing new applications. In particular, this modularization has made it much easier to decentralize drawing and modelling functions within a multi-agent CAD environment, where the CAD program is subservient to other processes (Figure 7.8).

At the same time, a new emphasis on interdisciplinary projects within research units at universities in several countries is leading to closer collaboration among the architecture, engineering and computer science disciplines. This is accelerating the transfer of methodologies in artificial intelligence, distributed computing, and cooperative systems to the computer-based design domain. As a result, new system architectures that provide a framework for distributed networks of semi-autonomous agents that cooperatively interact to solve problems, are poised to replace centralized CAD systems.

A CONCEPTUAL COMPUTER-BASED DESIGN ENVIRONMENT

Over the past four years the ICADS project has been successful in developing a particular approach for the implementation of a computer-based design environment, and identifying the requirements for an integrated multi-agent CAD-design software system. These findings have been reported progressively in a sequence of CAD Research Center Technical Reports [Pohl et al. 1988,1989,1991,1992, Myers et al. 1993].

The ability to demonstrate the approach in several working models (i.e., ICADS-DEMO1, ICADS-DEMO2, and AEDOT) has provided a useful testbed for the development of a body of knowledge relating to theoretical concepts and technical implementation issues. Foremost among these findings is the confirmation that little is gained by a CAD environment that excludes the meaningful participation of the human designer through excessive automation. The design activity presumes an element of the unknown, a problem that has to be solved through a decision making process that cannot be completely predefined because some information is missing. No existing computer system, nor one that is envisioned in the near future, can perform better than an experienced human being under problem conditions that require intuitive rather than logical solution procedures.

It seems logical that a computer-based design environment requires access to a great deal of information; such as user needs and expectations, construction systems or manufacturing conditions and materials, environmental requirements, codes and regulations, standard design practices, and so on. This

information is typically of a textual nature, and in orders of magnitude more plentiful than the data that describe the geometry of the artifacts in the drawing. The management of this information and the assistance that can be provided to the designer in the decision making process that leads to a solution, are critical factors that need to be addressed in the development of a computer-based design system. The areas in which this assistance is in highest demand would appear to include: access to knowledge of past similar projects; automatic analysis and evaluation of the current state of the design solution; conflict identification and generation of conflict resolution alternatives; information search facilities; and, the graphical assembly and manipulation of design objects. The success of the partnership between the human designer and the computer-based assistance facilities depends on the ability of the system to understand and anticipate the requirements of the designer.

A prerequisite for this 'understanding' and effective interaction between designer and computer is the representation of the evolving design solution model, and the context within which this model exists, in terms of high level objects. Each object must be richly described in terms of its physical appearance or conceptual meaning, its context, and its relationships to other objects. The more closely that description resembles the designer's real world view of the artifact, the more intelligent the designer will perceive the computer-based assistance to be.

The availability of knowledge to extend the design space, coupled with agents that cooperatively search the design space for alternative solutions, forms the basis of our proposal for a computer-based design system architecture (Figure 7.9). As an initial, limited implementation of this proposal the ICADS model provides a cooperative decision making framework for the coordination of multiple design assistance agents in an integrated knowledge-based CAD environment. Each agent, typically operating in a narrow domain, provides two kinds of expert design support: intermittent foreground responsiveness to requests for information and assistance initiated directly by the designer; and, continuous background monitoring and evaluation of the evolving design solution. In past implementations of the ICADS model the domain expert agents have been coordinated by a central controlling mechanism that guides the evolving design solution within the context of the design problem space.

Whether centralized or distributed, such a coordinator should be capable of resolving at least routine conflicts among the domain experts. A very common conflict condition will occur whenever two or more domain experts make different proposals for the same solution element. Based on its own multi-domain knowledge of the design space, the coordinator will invoke a procedure for resolving this conflict. The procedure should be consultative rather than dictatorial. In other words, while the coordinator's conflict identification/ resolution expert may suggest a compromise solution, all domain experts should

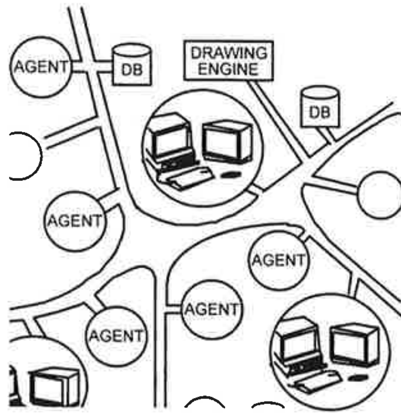

Figure 7.9. Distributed Cooperative Design

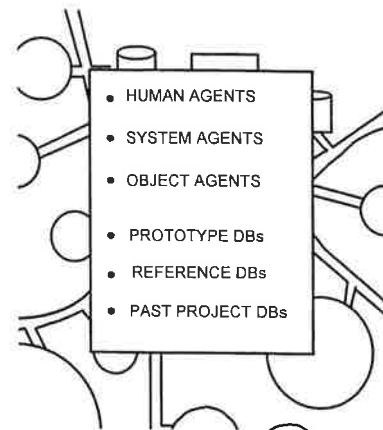

Figure 7.10. Agents and Information Resources

be given an opportunity to consider the impact of this solution on their own domains and suggest a new solution which may be different from the proposed compromise. The need for this kind of iterative consultative approach is indicative of the cooperative decision making behavior of designers that has been observed in manual practice [Schon 1983, Mackinder and Marvin 1982, Mallen and Goumain 1973].

Not only the existence of a substantial number of information resources, but also the ability of agents, coordinator(s) and designer(s) to easily access this body of knowledge, is fundamental to our view of a computer-based design environment (Figure 7.10). Available information resources are expected to include: the programmatic objectives and performance specifications of the design problem; information about prototype solutions of similar design problems; standard design practices; specific domain expertise; environmental context knowledge; and factual reference data.

OBJECTS AS AGENTS

So far, in the various versions of the ICADS model, we have used agents as evaluators and solution proposers that monitor the evolving design solution. They perform their work by analyzing the high level objects that describe the current design state, within the context of their own domain knowledge and dynamically formulated queries to other information sources (i.e., reference databases and prototype knowledgebases). In this sense they function as 'system agents' that have no ability to promote their own interests. They simply exist to express opinions about the current state of the design solution.

We now believe that this approach places limitations on the distributed nature of the ICADS model. From its inception one of the principal objectives has been to take full advantage of the distributed parallel processing potential of networked workstations. Accordingly, the ICADS model was conceived as a cooperative decision making system comprising multiple agents executing on separate machines and communicating with each other through a message passing facility. While it is true that the semantic network which contains the description of the current state of the system is also distributed among the global portions of the fact lists of the agents, a major section of the semantic network is centralized. This is the core section that contains the fundamental descriptions of the objects that the user manipulates graphically in the CAD environment. Only the evaluations that are performed by the agents (i.e., adjunct descriptions) are distributed in RELATION frames with the agents.

From a data management point of view the complexity of a given design project typically increases as the number of objects increases. In architectural design the size of the design space is very much a function of the number of different types of rooms, floors and buildings that have to be considered in the solution process. This is quite logical when one considers that the relationships among objects increase exponentially with the number of objects. Therefore, the centralization of a major portion of the object descriptions constitutes an impediment to the scalability of the current ICADS model into the arena of real world design problems.

A second problem is related to the computationally intensive nature of the graphic-object to design-object mapping that is required in CAD-design environments. If objects, such as the rooms in a building, are treated purely as information entities then a complete set of spatial relationships has to be available at all times whether or not such a complete description is required by any given agent at any particular time. This requires a great deal of computation to be performed, based on presumed need rather than actual request. Typically, the computation is likely to be performed by a single specialized agent that cannot be distributed on several machines.

Finally, if the fundamental descriptions of all objects are held in one place then all agents that require some part of this information have to compete for access to the same information depository. This is likely to create a communication bottleneck.

Some thought has been given to how these three problems could be addressed in the next version of the ICADS model [Myers et al. 1993]. We propose to treat at least some data-objects as agents that are capable of taking actions on their own behalf (Figure 7.11). It may not be desirable to elevate all objects to agent status. Objects that are subservient to other objects in most respects are unlikely to gain much from the capabilities of an active agent. For example, in architectural design, SPACE objects play a fundamental role from

the earliest stages of the design process. Windows, on the other hand, are largely subservient both in terms of function and impact to the SPACE in which they exist.

Exploring this proposal further in architectural design it would seem desirable to implement SPACE objects as agents. How would this ameliorate the problems identified above? First, any particular 'space' in its agent capacity is capable of sending and receiving communications. Therefore, the 'space' would be able to request assistance from other agents such as the system agents in the existing ICADS model. If the 'space' would like to know where it is located in respect to its nearest neighbors it could broadcast a request for specific information relating to its spatial context. A system agent whose domain encompasses the computation of spatial locations would receive the request, perform the necessary calculations, and transmit the results back to the 'space' agent. Two immediate advantages have accrued: only the most necessary computation has been performed; and, the results of the computation which form part of the fundamental description of the object can be held anywhere in the system (as long as they are available to any other authorized agent).

Second, by distributing the requesters, the requestees, and the information that is generated as a result of the servicing of the requests, the communications involved with both the current request transactions and any future use of the information have been likewise distributed. Accordingly, the potential for the occurrence of a communication bottleneck has been effectively reduced.

To explain how an object agent might behave in the kind of distributed operative environment provided by the ICADS model we will take the example of a 'space' agent in a building design application. Early during the conceptual design phase the architect is concerned with the relationships among spaces, particularly subsets of spaces that are likely to be grouped in proximity to each other on the same level of the building. These space groupings eventually evolve into floor plans leading to more detailed analyses of structural and environmental (e.g., lighting, heating, cooling, noise control, etc.) alternatives. However, during the early stages of this process the designer is concerned mostly with locational considerations based on the activities and functional requirements of the users of the spaces. Typically, designers will use circles or some other equally amorphous symbol to represent the particular spaces that they wish to group together and explore many spatial arrangements based on semantic and geometric considerations.

Typically, in the ICADS-DEMO2 environment, the designer would be able to select a particular space from a pool of spaces known by the system to be appropriate to the type of building being designed. The pool of spaces might be represented as labeled circles on one side of the CAD screen and the designer would simply click on the selected space and drag it into the central CAD

working area of the screen, to join a number of other previously selected spaces. Transparent to the actions of the designer, the system could concurrently commence execution of a process endowed with the capabilities of a 'space' agent. The agent process could originate in several different ways. For example, it might be spawned by a generic agent process and then initialized with the knowledge and capabilities appropriate to a 'space' agent (Figure 7.12). Alternatively, it might be simply awakened from a sleep state that was induced immediately after system initialization and maintained while the space remained in the pool. In fact, it is even possible for the space to commence its presence in the design session as a passive object and become an active agent at some future time based on system conditions or designer wishes.

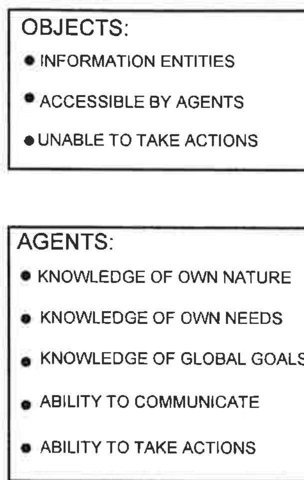

Figure 7.11. Comparison of Objects and Agents

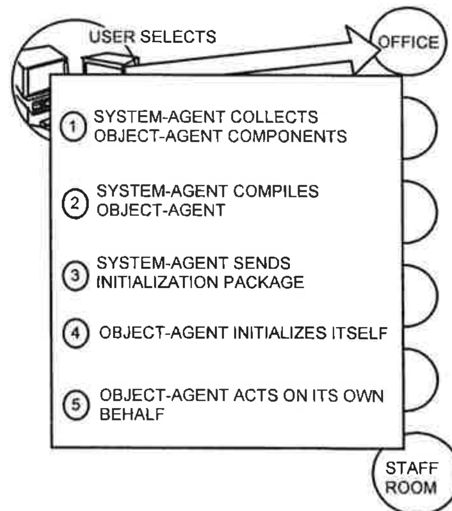

Figure 7.12. Creation of an Object-Agent

The first action of the 'space' agent, on joining the design session, would be to announce its arrival and any other information about its needs, interests, and capabilities that might be appropriate. Part or all of this information could be shared selectively with specific agents, such as a 'system manager' agent, or it could be placed openly on the bulletin board. In either case the 'space' agent would be expected to receive an address and certain communication privileges to allow it to function as an active member of the system community.

Next, the 'space' agent would most likely wish to orient itself within the group of spaces that it has just joined. It could broadcast a message of the type "Where are my neighbors?" This message would be heard by a system agent that is capable of computing the spatial relationships between the 'space' agent and its nearest (space) neighbors. If the system agent is currently idle it will

perform the required calculations and send a reply to the 'space' agent. A number of interesting possibilities exist in this scenario that warrant further discussion. First, the system agent might be busy working on a prior request from another agent. In this case, the new request could be placed in a queue, or the system agent could spawn a duplicate process to satisfy the request. The duplicate system agent would remain active as long as it is needed and then be terminated or put to sleep. Second, the information generated in reply to the 'space' agent's request can be stored in any convenient location in the system. This maximizes the potential for distributing not only the agents, but also the data structures representing the current state of the design solution, throughout the system. Third, based on the request from the 'space' agent the system agent can send messages to other agents to initiate one or more concurrent or sequential actions.

Many additional capabilities could be given to 'space' agents. For example, the ability to: reproduce itself; acquire a 'slave' space; subjugate itself to another space or higher level object agent (e.g., zone or floor); conduct search operations in databases; relocate itself automatically (e.g., closer to or away from another space; attract another space into its vicinity; change its shape; form other entities, such as walls, either as passive objects or active agents; change its behavior through the acquisition of new knowledge; temporarily hide itself from view in the CAD environment; deactivate its agent status and return to a passive object state; and so on.

In the existing ICADS model conflicts arise when system agents either disagree among themselves or with a decision made by the designer. For example, the placement of a window in a room might provoke the latter type of conflict. If the designer places the window in the west wall of a conference room and a loud noise source such as freeway runs parallel to the west boundary of the site, then the 'noise control' expert would probably insist on the removal of the window. The designer resolves the conflict by relocating or deleting the window or, alternatively, overrules the system agent. The conference room, as a passive object, is involved in the conflict resolution process only as an information source that is used by the system agent in its deliberations. In other words, while the validation of the design decision is entirely dependent on the knowledge encapsulated in the object the latter is unable to actively participate in the determination of its own destiny.

The situation is somewhat analogous to a scenario common in real life when one or more persons feel compelled to make decisions for another person, although the latter might be more competent to make those decisions himself. The outcome is often unsatisfactory because the decision makers tend to use general domain information where they lack specific knowledge of the other person. The 'individuality' of the problem situation has been usurped by the

application of generalizations and, as a result, the quality of the decisions that have been reached are likely to be compromised.

In the example of the window in the west wall of the conference room, if the conference room is a 'space' agent then much of the decision making can be localized within the knowledge domain of the agent. As soon as the window has been placed in the wall by the designer the 'space' agent could broadcast two specific requests for service: "What is the expected background noise level in the room due to the window?"; and, "What is the spatial distribution of daylight admitted through the window?". The answers to these questions can be compared by the 'space' agent directly to what it knows about its own acoustic and lighting needs. The development of alternative strategies for resolving the noise problem can now take place within the context of all of the information in the 'space' agent's knowledge domain. For example, the possibility of relocating itself to a quieter wing of the building can be explored by the agent (with or without the active collaboration of the designer) as a direct consequence of its own deliberations. In the existing ICADS model this remedy is less likely to be proposed by the Expert Design Advisor, because the interests of the space are fragmented among the various system agents that drive the conflict resolution process.

There is another kind of conflict resolution scenario that becomes possible with the availability of object agents. An agent may develop a solution to a sub-problem in its own domain that redirects the entire design solution. In the conference room example the 'space' agent may resolve the noise control problem by adopting an expensive window unit (e.g., triple glazing) solution, and then continue to search for a better solution. The search may continue into subsequent stages of the design process, during which the conference room becomes part of 'floor' and 'building' object agents. These higher level agents may now impose certain characteristics onto the 'space' agent for the greater good of the larger community.

However, the 'space' agent, persevering in its search finally comes up with a method of noise control that utilizes a novel type of wall construction in combination with background masking sound. The proposed wall construction is contrary to that adopted for the external west wall of the building by both the 'floor' and 'building' agents. First, it is significant that this alternative solution has been found at all. If the conference room had been a passive object there would not have been any desire on the part of the system to pursue the problem after the initial conflict resolution. Second, having found the alternative the 'space' agent is able to communicate its proposal and have the noise control issue reconsidered. It could notify, in order of authority, the 'floor' agent and the 'building' agent. At each of the agent levels there is the opportunity for wider consultation and interaction with the designer. Finally, if the proposal has been rejected at all higher agent levels, the 'space' agent may appeal directly to the

designer. The designer has several alternative courses of actions open: also reject the proposal; require one or more of the higher level agents to explain their ruling; reset certain parameters that allow the higher level agents to reconsider their ruling; overrule the higher level agents and accept the proposal; or, capture the current state of the design solution as a recoverable view and use the 'space' agent's proposal as the basis for the exploration of an alternative solution path.

An object agent must be knowledgeable of its own characteristics and needs, must be able to acquire additional information from external sources, and must be capable of communicating with other members of the cooperative community. This means, specifically, that the agent must: encapsulate a great deal of information about itself; be able to conduct searches; be able to reason about the information under its purview; be able to send and receive messages from other agents; and, be able to interact with the user.

While it may sometimes be desirable to add more sophisticated action mechanisms to the agent's repertoire, the fundamental capabilities listed above are sufficient for an object agent to seek the necessary assistance for undertaking any of the following actions:

- develop solutions for satisfying its own needs relative to internal and external constraints;
- implement solutions within its own domain subject to the current level of authorization;
- find out about the needs of other objects;
- ascertain the current state of other object agents;
- broadcast requests for services and select the most desirable bidder;
- broadcast information and proposals to the community at large;
- communicate with specific agents;
- interact with the user;
- attach passive sub-objects to itself and reason about those objects.

This set of capabilities is not intended to be comprehensive. It represents the boundaries of our near term plans relating to the implementation of the ICADS-Kernel model, discussed in the next section. Additional capabilities, such as the ability to spawn lower level objects, may be considered appropriate in the future.

THE ICADS-KERNEL MODEL

The ICADS-Kernel is a collection of software pieces that will support current implementations of the ICADS model such as ICADS-DEMO1 and ICADS-

DEMO2, as well as proposed future distributed systems that are also cooperative and interactive. Central to the Kernel is an application independent platform that provides the base facilities for configuring a set of distributed processes, to form a system with particular characteristics. The system will be capable of installing and executing its own processes, with the availability of a high-level user interface, and maintain contact with the processes. Communication among software modules that make up the Kernel, as well as application modules, is provided through a distributed message passing system that is being developed as an independent service facility. In addition there is a set of tools that will be commonly needed by developers and users of application systems that are designed to execute within the framework of the ICADS-Kernel model.

Based on experience with previous implementations of the ICADS model, motivation for the new system came from an increasing desire for a more general experimental environment which could support cooperative problem solving approaches involving many asynchronous, semi-autonomous agents. Of particular interest, in this regard, is the degree to which the decision making activity can be localized and to what extent problem solving behavior can be simplified in agents that have a high degree of self-determination allowing them to freely interact with each other and their environment [Brooks 1990].

A critical review established several general requirements for the new system (Figure 7.13). Foremost among these was the need for a flexible, transparent, non-application specific message passing facility to support the high degree of communication anticipated. A second requirement was a user interface facility that would allow application systems to be customized within the framework of the ICADS-Kernel. In particular, there was a desire to receive interactive assistance from the system during the selection of agents and information resources and their distribution within the network at the beginning of a design session.

A third requirement was the ability to save and restore the current state of the system at any time, both for recovery during the same design session and at a later time. This was considered to be a necessity for large application projects that cannot be completed in a single session. Just as important, it makes it possible for the user to save the state of the session at a particular point in time and perform experiments, knowing that any saved state can be restored later. A save and restore facility also supports multi-user work. One user will be able to save the state of the session, allowing other users to use this saved state as a starting point for further work.

Explanation facilities for users and high-level tracing tools for developers were identified as a fourth requirement. While there is considerable information available to the user of previous ICADS versions that can be used to explain the

current state of the design solution, most of this information is hardcoded and specific to those systems.

The possibility of using a commercially available distributed operating system to provide the process and communication facilities requested for the ICADS-Kernel system was considered, but eventually dismissed. Most distributed operating systems are either transaction oriented or devoid of utilities to support inter-process communication among the distributed processes. They may provide a high degree of reliability in executing the requested processes, or in automatically selecting the host machines to optimize the computation, but they require the user to employ socket-level software within the processes in order to communicate with other processes. ISIS [Birman and Marzullo 1989] is an example of the former type of system and Plan 9 [Pike et al. 1990] is an example of the latter.

Furthermore, the message passing facilities in most distributed operating systems are so focused on fail-safe delivery that the required protocols produce enormous overhead. With one exception they appear to be too inefficient to be used in the highly interactive-reactive ICADS environment. This exception is the PVM (Parallel Virtual Machine) system [Sunderam 1990] which shares many of the goals established for the ICADS-Kernel. The primary difference is that it requires applications to poll in order to receive a message, whereas the message passing facility intended for the Kernel is based on an interrupt paradigm. Our experience with interrupting messages in the GXI module of the previous ICADS model has led us to believe that this is a more efficient approach [Pohl 1992].

In order to modularize the ICADS-Kernel and clarify the relationships between major components, the concepts and code that perform the actual saving and restoring of an ICADS session, or view, have been separately identified as the SARDINE (Save and Restore In a Distributed Interactive Networked Environment) system.

As a result the ICADS-Kernel consists of the Message Management System, SARDINE, and Kernel Tools that include software utilities developed to help with Kernel applications. While the Message Management System is being developed to function independently of SARDINE, the latter is totally dependent on the services provided by the former.

THE CONFIGURATION MANAGER

The Configuration Manager (CM) is an integral component of the SARDINE system. After working within a distributed environment using the ICADS model it was realized that two capabilities would prove to be extremely helpful and would increase the productivity of such an environment. The first capability

is an easier way of establishing the environment (i.e., setting up the processes to execute on different machines across the network). Initially, in the ICADS model, this was handled by different script files

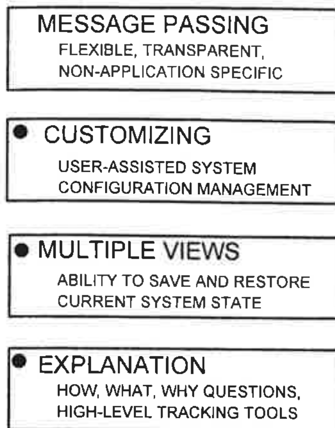

Figure 7.13. ICADS-KERNEL Requirements

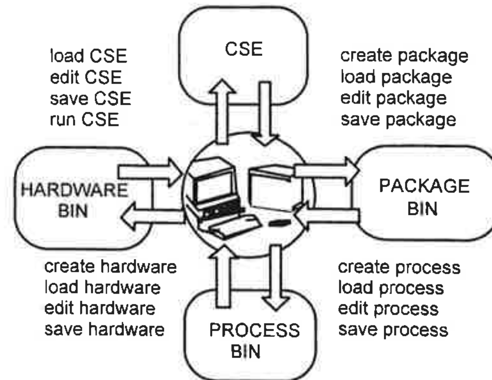

Figure 7.14. Configuration Manager (CM)

that would start up the processes on their respective machines. The limitations of this method quickly became apparent. The script files had to be modified to change the configuration, a non-trivial task for novice users. Another drawback to this method is the inability to save a configuration and load it at a later time.

The second capability is a facility for saving and restoring the running environment to a disk. In the case of ICADS, it was soon found to be unreasonable to expect the user to accomplish all design work in one session, therefore, the capability to save and restore a running environment became a necessity. The task of establishing as well as saving and restoring an application configuration is accomplished by the Configuration Manager.

To facilitate the design of the CM, in terms of implementation and modification, an object-oriented design paradigm was adopted. At the highest level of design, the system is described in terms of the interaction of five primary objects: Current State and Environment (CSE); User Interface; Hardware Bin; Package Bin; and, Process Bin.

Briefly: the CSE represents the configuration and current state of all processes; the User Interface takes commands from the user and translates these into messages to the objects; and, the Bins are used to manage all objects of the designated types. For example, the Process Bin keeps track of Process objects and the Hardware Bin contains all of the Hardware objects.

The object diagram in Figure 7.14 describes the structure of the classes used in modelling the CM. The Hardware object represents either a machine or a

monitor, which are both treated as hardware items. A Machine object contains information specific to a certain machine on the network (i.e., network address, machine name, capabilities, etc.). Similarly, the Monitor object contains information regarding a specific monitor (i.e., size, resolution, color capability, etc.). The CM uses the information in these objects to run and display processes.

The Process object is either a State-Saving Application Process (SAP) or a Non-State-Saving Application Process (NSAP). SAP processes can be saved to a disk during execution and restored at a later time. NSAP processes do not have this ability. Regardless of their type, any information that the CM needs in order to run these processes is stored within the Process object. The Package object contains references to Process and Hardware objects that are bundled together for convenience. No additional functionality is gained from putting a Process or Hardware object into a Package. A Process object does not know what Package it is associated with (as it may be contained in more than one Package). All three of these definition objects are managed by Bin objects (e.g., all Process objects are managed by the Process Bin).

The User Interface object, which serves as the central communication hub for all of the objects, interacts with the Bin objects to retrieve and store definitions. A physical component is associated with the Bin objects (currently a binary file), which persists between sessions. The CSE object contains all information necessary to reproduce the current running environment. All Hardware and Process objects used in the system are referenced in the CSE. The CSE object must also be capable of saving itself to disk for later reloading.

THE SAVE AND RESTORE FACILITY

The need to save the current state in an expert system is generally recognized. For example, both ART [Williams 1985] and KEE-3.0 [Chorofas 1990] provide such facilities through 'breakpoint' options and the notion of 'worlds', respectively. However, CLIPS [NASA 1989] the expert system shell of choice in the ICADS project, required a minor modification to allow for the implementation of a save and restore facility (i.e., the addition of a function that deletes all rules from the execution agenda).

Subject to the assumption that a save request will be explicitly initiated by the user in the ICADS environment, the following logic for performing save and restore operations has been successfully tested with our modified version of CLIPS:

To SAVE:

1. Close all windows that are not produced by the initial start-up of the system.
2. Save the CLIPS fact list.

3. If the save is to be restored in another CLIPS session, save the knowledgebase.

To RESTORE:

1. If the knowledgebase has been changed, or this is a new CLIPS session, load the saved knowledgebase.
2. Load the saved fact list.
3. Empty the agenda.
4. Run.

The application of this scheme to the save and restore facility in the ICADS-Kernel model has necessitated several extensions. First, all components are required to be classified as either Non-Saving APplications (NSAP) or Saving APplications (SAP). Second, all NSAPs must be idle before a save action is initiated and are required to be designed to operate as initial invocations whenever they are called. Third, all SAPs must have both save and restore facilities. Fourth, a mechanism is available to indicate when no messages are in transmission between components (Figure 7.15).

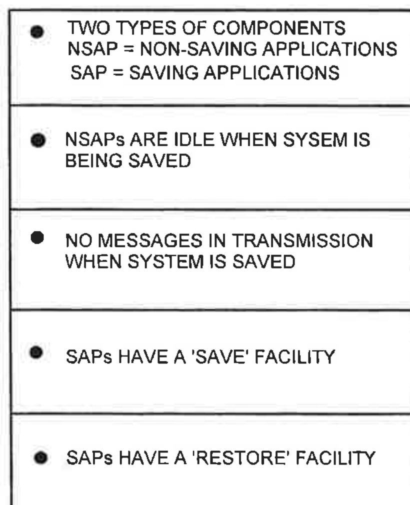

Figure 7.15. Save and Restore Assumptions

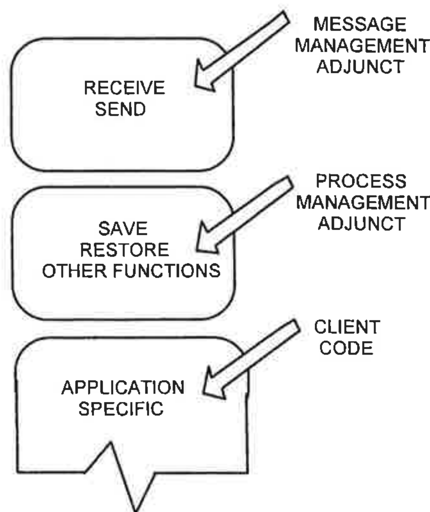

Figure 7.16. Saving Applications (SAP) Structure

In response to the user request for a permanent SAVE, the interface asks the user to supply a name for the saved state and a short text identification for the checkpoint. This name is recorded in a file in the user's account along with all information necessary to describe the saved system. The system waits until it determines that all SAP processes are idle and the message transmission check is satisfied. Then a special 'save' fact is sent to all components. The CLIPS

processes respond by saving their fact lists, and one or more specifically designated NSAPs store those parts of the distributed semantic network that are external to the CLIPS processes in a database.

The restore from a permanent SAVE is largely the inverse of the preceding steps. The external portions of the semantic network are loaded from the database, the geometric representation of the saved design solution is regenerated by a NSAP using the CAD system as a drawing engine, and all CLIPS processes load their saved files and clear their agendas. The interface then returns control to the user and the saved session is continued.

The permanent SAVE is intended to capture a view of the design state for a future session. Therefore, a delay of several minutes should be acceptable. A much faster facility, referred to as a temporary SAVE and restore, is available to allow the user to experiment with alternatives during the same session.

When the user requests a temporary SAVE, the interface will again ask for the user to provide a name for the state to be saved. It will also record a short description of the state being saved, if the user wishes to provide the text. This will make it easier to identify the state that should be restored, if several states have been saved. The System Manager (SM) will wait until the SAPs are all idle and there are no messages in transmission. It will then instruct the special Kernel programs, called Process Managers, to save their SAP processes. This requires some explanation.

The Kernel Configuration Manager (CM) provides names for all of the components, the machines where they should execute, and the displays to which they should attach. The SM initiates a special process called the Process Manager (PM) on each machine that will host a component process. The latter are then 'forked' from the local PM as child processes on the same machine.

Each SAP has some generic code, called its Kernel Adjunct Code or Adjunct, for short. This code is added to non-Kernel programs such as current ICADS system agents to enable them to work in the Kernel environment (Figure 7.16). When the PMs are told to save their component processes, they send a signal to each of their SAPs. This signal is picked up in the Adjunct and causes the SAP to perform a fork. After the fork, the parent passes the PID of the child to the Process Manager and then goes into a wait state. The child will continue to execute the process.

The Process Manager identifies the saved state as the group of PIDs for the parent processes and the currently executing system as the group of child PIDs. Since the fork takes only a few dozen milliseconds the system will be ready to continue in a fraction of the time required for the permanent SAVE. Additional temporary SAVEs may be made in the same way.

If the user wishes to restore the system to a checkpoint created by a temporary SAVE, the interface will be able to provide the names and descriptions of saved states to assist in the proper selection. Then, when the

selection is made, the interface will ask the PMs to put the current versions of the SAPs to sleep and resurrect the versions whose PIDs are associated with the selected name.

THE MESSAGE MANAGEMENT SYSTEM

The communication and event management facilities required in support of a comprehensively distributed system, such as the ICADS-Kernel, exist at both the logical and physical levels. Logically, the underlying communication facility should allow its clients to communicate with each other through object-oriented messages. Clients should be able to use any type of object as an inter-client message without content transformation. Further, client application environments should be able to define and manipulate their own set of objects, both statically and dynamically. These performance requirements are predicated on a high degree of flexibility within the interface presented by the communication facility. In addition, clients must be able to function without knowledge of the physical characteristics of their counterparts [Elmasri and Navathe 1989]. Such characteristics include the current state of execution and the physical site location.

Client authorization is another important issue. Each application environment should be able to define and manage its own authorization of privileged facilities. This external authorization must work in conjunction with the underlying authentication and security mechanisms provided by the communication system. Furthermore, clients must be notified of pending events in real-time [Durfee 1988]. In other words, the elapsed time between the triggering of an event and the subsequent notification of the appropriate client(s) should be reduced to a minuscule amount. This is a critical requirement for providing data validity and currency within the application environment.

To be able to support applications that deal with both knowledge and numerical data, the communication system must offer its facilities in both procedural and rule-based form [Pohl et al. 1988]. Further, clients must be able to interact with one another independently of their implementation language. This means that rule-based clients should be able to freely interact with procedural clients and vice versa.

Physically, all of these capabilities must be implemented in such a way as to provide for as high a degree of performance and application flexibility as possible. This requires the design of the communication system to seek out the most cost effective mechanisms for performing its assigned tasks. In addition, the selected mechanisms must be able to perform in a highly reliable and robust manner. For example, the underlying support system must handle such

situations as receiving unknown message objects or the detection of a potentially malicious authorizer.

In order to take advantage of the benefits of a networked, workstation environment, the entire communication system should be designed and implemented in a distributed fashion [Stevens 1990]. The resulting design must address such issues as: site atomicity; fault tolerance; and, agreement among distributed agents [Berstein et al. 1987]. Moreover, the system should support the dynamic allocation and deallocation of various resources within the application environment. This is a concept that should be extended to cover the entire site environment.

The solution approach takes the form of a fully distributed, real-time communication and event management system referred to as the Mercury Message System (MMS). MMS exists as an underlying support system for highly interactive, multi-agent, distributed application environments. To facilitate heterogeneous procedural and rule-based application environments MMS offers its facilities to both 'C' language and CLIPS clients [NASA 1989]. These facilities take the form of such mechanisms as user-defined objects, user-defined and managed facility authorization, and real-time event notification. However, care has been exercised in the selection of internal mechanisms to find a balance between efficiency and flexibility.

MMS provides a mechanism for client applications to define their own message type protocols. Clients may attach special meaning to each message via a 'message type' field located within the message header of each inter-client message. In addition, clients may set up application specific message protocols to allow one application to trigger an event in another. Multiple message types may refer to the same physical message. However, any particular message type can only refer to a single physical message.

Inter-process communication systems available today, typically, constrain users by providing only a small set of data type primitives with which clients may interact. Applications wishing to interface with this underlying support system are required to transform or translate their data objects into one, or several, of these data type primitives. This restriction on application objects leads to an environment seriously lacking in application flexibility and extendibility. Furthermore, application efficiency is greatly reduced as a result of the considerable overhead associated with object transformation.

MMS addresses this issue by introducing the concept of client defined objects. Each application may define its own set of application specific data types along with a logical and concise set of access methods [Booch 1991]. These objects are not restricted to a single application, but can be shared among MMS clients by passing them as inter-client messages. Applications seeking a series of object types best suited to represent their specific type of data may select from a cumulative list of application defined object types. However, if the

desired object representation does not yet exist, clients are free to define any number of additional object types by adhering to a minimum set of guidelines.

MMS provides facilities to allow application environments to design and implement their own authorization protocols, through the notion of an Acting Client Authorizer Group (ACAG). This group comprises a single, or multiple MMS clients designed in a manner that allows them to work in a collective fashion to determine the result of each client authorization request. By associating an authorization package with each of its clients, MMS is able to allow clients to request an alteration to their authorization protocols. Accordingly, each time a client attempts to perform a 'protected' MMS action, the associated authorization package is checked to determine whether the particular client is, in fact, authorized to perform the desired action.

A MMS site consists of five major components as shown in Figure 7.17. Each of these components is designed to achieve a high level of automicity, integrity, consistency, efficiency, and portability, within a networked UNIX workstation environment. The TCP/IP protocol is used to provide site to site communication [Stevens 1990]. In addition, the design of MMS is based on such fundamental UNIX mechanisms as shared memory and inter-process signals [Bach 1986].

Clients wishing to utilize the facilities of MMS are required to 'link' their application to a library, referred to as the MMS Adjunct (MMA). Any interaction between the client and MMS is handled via this adjunct. Furthermore, it is the MMA that initially receives all communications directed to its associated client. Depending on the particular state of the client, the MMA will either trigger the appropriate client action or simply buffer the message for processing at a later time. In addition to providing the basic MMS interface, the MMA also contains a set of user-defined objects along with their interfaces.

MMS utilizes the concept of a Local Site Catalog (LSC) to record information describing the various clients, objects, and resources known to the particular MMS site. Physically, the LSC exists as a section of memory that can be shared among multiple MMS processes on the local site. Each MMS site contains its own LSC. The particular state this shared resource is in at a specific point in time determines the type of access that is permitted. For example, for data consistency reasons a 'Read' access request is not permitted while the LSC is in an 'Update' state. Access to the LSC is controlled via the acquisition of two types of locks: 'Shared'; and, 'Exclusive'. Shared locks are associated with 'Read' operations and exclusive locks are associated with 'Update' operations. Since 'Read' operations do not physically alter the contents of the LSC they can be performed in parallel. However, 'Update' operations physically alter the LSC's content and consequently require an Exclusive lock. Controlling access to the shared resources ensures the integrity and consistency of its contents relative to 'Read' and 'Update' operations [King and Collmeyer 1973].

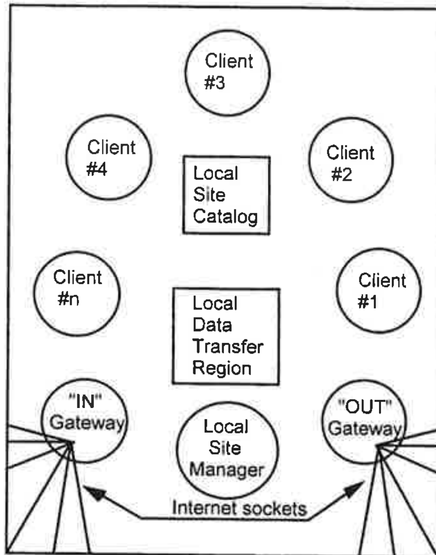

Figure 7.17. Principal MMS Components

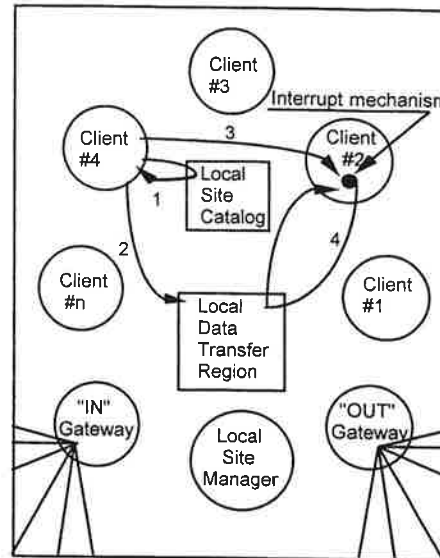

Figure 7.18. Local Site Communication

Similar to the LSC, a Local Data Transfer Region (LDTR) exists as a section of memory that can be shared among the various MMS entities. Access to the LDTR is determined in the same manner as the LSC. The main function of the LDTR is to provide a common mailbox into which MMS agents can place or retrieve inter-client messages. Clients wishing to send an inter-client message acquire 'Send' access to the LDTR and place their message in the appropriate mail slot. Receiving clients acquire 'Retrieve' access to the LDTR and simply pick up their mail. The 'Send' and 'Retrieve' functions are handled directly by the MMA of the particular client, transparent to the associated client application.

Local In/Out Gateways on each machine serve as portholes to the outside world, connecting this MMS site to all other MMS sites. At the physical level, each Out Gateway has a direct internet socket connection to the In Gateways of all other MMS sites [Stevens 1990]. Conversely, each In Gateway has a direct connection to the Out Gateways of all other MMS sites. Therefore, if a message must move between MMS sites it is guaranteed that only a single site-to-site transfer will be required (i.e., no intermediate sites will need to be visited during a message's journey between the source and destination site).

When a new site is dynamically allocated, or an existing site is deallocated, the number of gateway connections will be altered accordingly. To simplify the design of MMS, the functionality of the gateways is similar to any other MMS client. In fact, the gateway itself is viewed as a client by MMS and is listed as

such in the LSC. Therefore, a gateway has the potential to both send and receive inter-client messages.

The various MMS resources and entities located in a particular site are managed by the Local Site Manager (LSM), which also acts as the MMS representative for that site. Any communication directed specifically to a MMS site is handled by the associated LSM. For example, if a new MMS site is to be allocated in the system, MMS will execute a series of protocols requiring internal communication between MMS sites. This communication will typically take place between LSMs located on the various MMS sites. Again, it must be emphasized that MMS clients never specifically address a particular LSM. Rather, any direct communication between a client and MMS is addressed to the MMA associated with the client. The MMA then determines which site to direct the request to.

MMS communications can be divided into two categories: local site communication between two MMS clients existing on the same site; and, site-to-site communication between MMS clients existing on different sites. In local site communication a MMS client initiates the sending of a message by calling the appropriate MMA function. The MMA reads the header information of the client message, such as the size of the message object and the client that the message is directed to. Once the appropriate information has been extracted from the message header, the MMA uses the symbolic name of the destination client to locate the appropriate client identification code in the LSC. This is illustrated as '1' in Figure 7.18. The MMA then places the actual message object into the LDTR ('2'). If this step is successful, the MMA will send a 'message pending' signal to the MMA of the destination client ('3'). This signal essentially interrupts, or preempts, the destination client so that it will be able to handle the particular event in a real-time fashion. Depending on the particular MMS function that was called, the sender's MMA may either wait for an acknowledgment from the destination MMA or simply return control to the client application. When the destination MMA receives the 'message pending' signal, it takes action by retrieving its 'mail' from the LDTR ('4').

Site-to-site communication actions are shown in Figure 7.19, from the viewpoint of the sender site, and in Figure 7.20, from the viewpoint of the receiver site. Access to the LSC by the sender's MMA (Figure 7.19 '1') reveals that the destination client is not located on the same site as the sender. Accordingly, the MMA of the sender extracts the site reference of the destination site and the identification code of the local Out Gateway from the LSC. The MMA then places the physical message object into the LDTR in the same manner as shown in '2' of Figure 7.18. Next, the 'message pending' signal is sent to the local destination client (Figure 7.19 '3'). However, in this case the local destination client becomes the Out Gateway. Acting as any other MMS client, the MMA of the Out Gateway retrieves the message object from

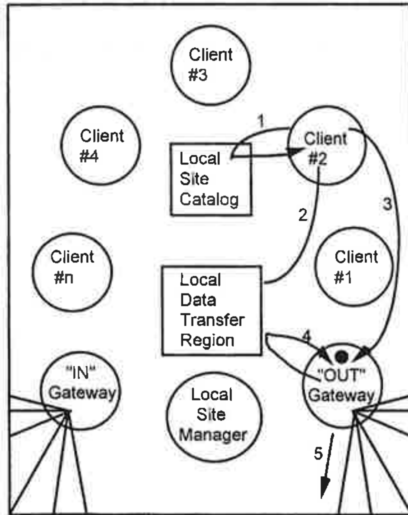

Figure 7.19. Site-to-Site Communication, Sender Site

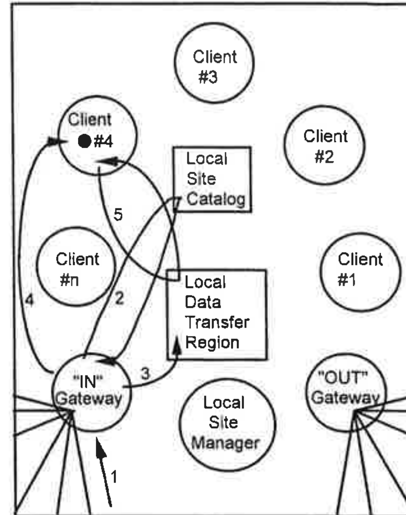

Figure 7.20. Site-to-Site Communication, Receiver Site

the LDTR and passes it to the corresponding Client Event Handler ('4'). The latter processes this message by sending it to the In Gateway of the destination site ('5'). At this point the message object has physically left the local MMS site enroute to the destination site.

Once the client message enters the destination site it is received by the local In Gateway of that site (Figure 7.20 '1'). The In Gateway then processes the message by performing local client-to-client communication of the message object to the destination client. Steps '2', '3', '4', and '5' of Figure 7.20 are identical to steps '1', '2', '3', and '4' of Figure 7.18. The destination client has absolutely no indication that the client who sent the message is located on a foreign site. In terms of client-to-client communication, MMS presents its clients with a view of the world that is completely independent of physical location. This independence is essential to the notion of atomicity.

SOME OPERATIONAL ASPECTS OF SARDINE

In the design of the ICADS-Kernel an emphasis has been placed on the ease with which users, as well as developers, are able to make modifications. We are specifically concerned with minimizing the amount of work that is required to bring independently developed code into an ICADS-Kernel system. In particular, we wish to make it easy to take a knowledge-based system that has been developed on a stand-alone basis and bring it into a SARDINE system as a domain expert, or cooperating expert.

With this conversion process in mind, we have separately identified the functions that are necessary in order for a process to fully participate in the SARDINE environment. Two kinds of functions are necessary. The first type are functions that are needed to enable the process to communicate with the Message Management System. The second group of functions are those that are necessary to interact with the Process Manager.

In Figure 7.21, the composition of these functions is shown. The code that makes up a non-Sardine process is described as Client Code. The added code that handles the interaction with the Process Manager is referred to as the Process Manager Adjunct (PMA), and that for the Message Manager activities is called the Message Manager Adjunct (MMA). Some of these functions are totally transparent to the Client Code. Others must be called from the Client Code itself. In either case, by collecting the additional code modules into the PMA and MMA respectively, it will be much easier to enhance non-Sardine processes to bring them into a SARDINE environment, and it will also be much easier to maintain the Adjunct codes themselves.

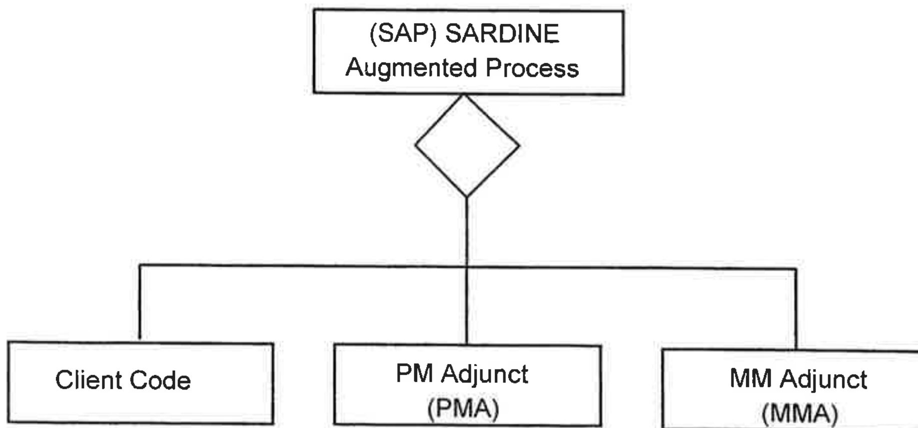

Figure 7.21. SARDINE Augmented Process (SAP)

Any process that is augmented with the PMA and MMA code, and which has the proper calls to the functions in those adjuncts, is referred to as a SARDINE Augmented Process. Such a process now has capabilities that are equal to our previous definition of a SARDINE Application Process and, therefore, can share the acronym SAP.

The basic relationships among the Message Management System, the Configuration Manager, and the rest of the SARDINE support code can be seen in Figure 7.22, which depicts the majority of relations between the central Kernel objects. In particular, this figure shows the objects that are involved

when the user requests the RUN command through the Configuration Manager's main menu. A SARDINE module called the System Manager reads the information from the Current State and Environment (CSE) object maintained by the Configuration Manager. The CSE denotes all processes that will make up the execution session at this point in time, their displays, and all other information necessary to create a working session.

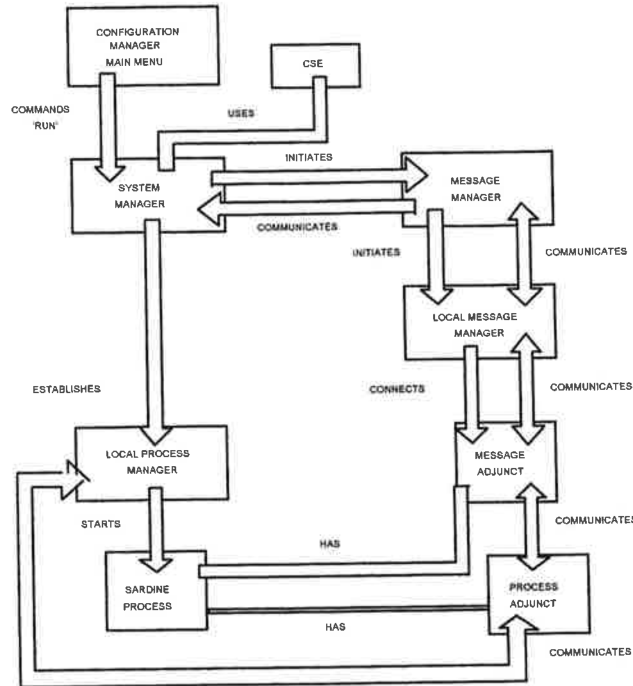

Figure 7.22. Basic SARDINE Relationships

When the user first began to work with the Kernel, the System Manager initiated the Message Management System (MMS). Now, in response to the RUN command, the System Manager communicates with the MMS to request that each machine that will host a process be entered into the communication system. The MMS establishes a Local Message Manager (LMM) on any of these machines that does not currently have a LMM and communicates its success, or failure, to the System Manager.

When all of the LMM units are in place, the System Manager starts a Local Process Manager (LPM) on each of the involved machines. The System Manager will ask the LPMs to start each process that is to execute on the local machine. Then, when all processes are in place, the System Manager directs the LPM to permit execution of the processes. Thereafter, communication within the system will take place almost exclusively through the MMS.

CONCLUSIONS

The development of the ICADS-Kernel model recognizes the convergence of current trends in the configuration of computing environments and the needs of complex problem solving domains. Three trends in the deployment of computing capabilities are specifically apparent: increased computing power at the user workstation; on-line access to larger, shared databases and computing resources; and, connectivity among users at all levels on an increasing scale. Distributed computing systems offer several advantages over centralized, monolithic systems. These advantages are related not only to cost effectiveness, reliability, and diversity, but also to the manner in which such systems can support cooperative problem solving paradigms.

Architectural design has many characteristics in common with distributed problem solving systems. Systems of this kind typically involve loosely coupled networks of semi-autonomous agents that cooperatively interact to solve problems [Durfee 1988, Agha 1988]. Many design tasks are inherently distributed and concurrent in nature. Problem decomposition allows sub-tasks to be performed in parallel with considerable local autonomy. Yet, all elements of the evolving solution are interdependent and have a need to cooperate with each other through direct communication.

The ICADS project has been drawn toward the distributed cooperative computing model because it appears to provide a rich and computationally powerful environment for exploring both the organizational and functional aspects of cooperative decision making. Of particular interest in this regard are global and local planning strategies, communication protocols among computer-based agents and human designers, conflict detection and resolution, knowledge acquisition, and explanation facilities. We look upon the ICADS-Kernel system as an enabling environment in which experiments to test and evaluate computer-assisted design concepts can be conducted.

Since its beginning in 1987 the ICADS project has enjoyed the participation of a substantial number of students and faculty drawn from several disciplines. All of these team members have contributed to the evolution of the ICADS model, through several versions, to the point where we are now able to confidently pursue a considerably more sophisticated implementation of our original concept. Specifically, in respect to the current ICADS-Kernel work, the authors wish to acknowledge the contributions of Kym Jason Pohl who is responsible for the design and implementation of the Mercury Message System, and Tony Rodriguez who is developing the Configuration Manager of SARDINE.

REFERENCES

Agha, G.A. (1988). *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass.

Akin, O. (1986). *Psychology of Architectural Design*, Pion Ltd., London, England.

Akin, O. (1978). "How Do Architects Design?" in Latombe (ed.), *Artificial Intelligence and Pattern Recognition in ComputerAided Design*, IFIPS, North Holland.

Archea, J. (1987). "Puzzle-Making: What Architects Do When No One Is Looking," Kalay (ed.), *Principles of ComputerAided Design: Computability of Design*, Wiley, New York.

Bach, M. (1986). *The Design of the UNIX Operating System*, Prentice Hall, Englewood Cliffs, New Jersey.

Berstein, P.A., V. Hadzilacos, and N. Goodman (1987). *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass.

Birman, K., and K. Marzullo (1989). "ISIS and the META Project," Sun Technology, Summer, 90-104.

Booch, G. (1991). *Object Oriented Design*, Benjamin/Cummings, Redwood City, California.

Broadbent, G. (1979). "Design and Theory Building," *Design Methods and Theories*, **13** (3/4): 103-7.

Brooks, R. A. (1990). "Elephants Don't Play Chess," Maes (ed.), *Designing Autonomous Agents*, MIT Press, Cambridge, Massachusetts, 3-15.

Chorafas, D. (1990). *Knowledge Engineering*, Van Nostrand Reinhold, New York, 99-102.

Cross, N. (ed.) (1984). *Developments in Design Methods*, Wiley, New York.

Durfee, E. (1988). *Coordination of Distributed Problem Solvers*, Kluwer Academic, Boston, Masachusetts.

Elmasri, R., and S. B. Navathe (1989). *Fundamentals of Database Systems*, Benjamin/Cummings, Redwood City, California.

Fischer, G., and K. Nakakoji (1991). "Making Design Objects Relevant to the Task at Hand," Proc. AAAI-91, Ninth National Conference on Artificial Intelligence, MIT Press, Cambridge, Massachusetts, 67-73.

Gero, J., M. Maher, and W. Zhang (1988). "Chunking Structural Design Knowledge as Prototypes," Working Paper, The Architectural Computing Unit, Department of Architectural and Design Science, University of Sydney, Sydney, Australia.

Kim, Y., and L. O. Degelman (1990). "Knowledge-Aided Design: A Theory and Implementation," Focus Symposium on Knowledge-Based Systems in Building Design, 5th International Conference on Systems Research, Informatics and Cybernetics, Baden-Baden, Germany.

King, P., and A. Collmeyer (1973). "Database Sharing: An Efficient Mechanism for Supporting Concurrent Processes," Proceedings of the NCC.

Kirk, S. J., and K. Spreckelmeyer (1988). *Creative Design Decisions: A Systematic Approach to Problem Solving in Architecture*, Van Nostrand Reinhold, New York.

Lakoff, G., and M. Johnson (1980). *Metaphors We Live By*, University of Chicago Press, Chicago, Illinois.

Lawson, B. (1988). *How Designers Think: The Design Process Demystified*, Butterworth, London, England.

Lenart, M., and G. Ketteler (1991). "A Computer Integrated Manufacturing System for Wooden Staircases," Focus Symposium on Computer-User Partnerships in Design, 3rd International Symposium on Systems Research, Informatics and Cybernetics, Baden-Baden, Germany.

Mackinder, M., and H. Marvin (1982). "Design Decision Making in Architectural Practice," Building Research Establishment, Department of Environment, HMSO, London, England.

Mallen, G. L., and P. G. Goumain (1973). "The Analysis of Architectural Design Activity in the Working Environment," *Report 108/4 DDR*, Royal College of Art, London, England.

Myers, L., J. Pohl, J. Cotton, J. Snyder, K. Pohl, S. Chien, S. Aly, T. Rodriguez (1993). "Object Representation and the ICADS-Kernel Design", *Technical Report, CADRU-08-93*, CAD Research Center, Design and Construction Institute, College of Architecture and Environmental Design, Cal Poly, San Luis Obispo, California.

NASA (1989). "CLIPS Architecture Manual (Version 4.3)," Artificial Intelligence Section, Lyndon B. Johnson Space Center, Texas.

Pike, R., et al. (1990). "Plan 9 from Bell Labs," Research Note, Bell Labs.

Pohl, J., L. Myers, J. Cotton, A. Chapman, J. Snyder, H. Chauvet, K. Pohl and J. La Porta (1992). "A Computer-Based Design Environment: Implemented and Planned Extensions of the ICADS Model," *Technical Report, CADRU-06-92*, CAD Research Center, Design and Construction Institute, College of Architecture and Environmental Design, Cal Poly, San Luis Obispo, California.

Pohl, J., L. Myers, A. Chapman, J. Snyder, H. Chauvet, J. Cotton, C. Johnson and D. Johnson (1991). "ICADS Working Model Version 2 and Future Directions," *Technical Report, CADRU-05-91*, CAD Research Center, Design Institute, College of Architecture and Environmental Design, Cal Poly, San Luis Obispo, California.

Pohl, J., L. Myers, A. Chapman, and J. Cotton (1989). "ICADS: Working Model Version 1," *Technical Report, CADRU-03-89*, CAD Research Unit, Design Institute, School of Architecture and Environmental Design, Cal Poly, San Luis Obispo, California.

Pohl, J., A. Chapman, L. Chirica, R. Howell, and L. Myers (1988). "Implementation Strategies for a Prototype ICADS Working Model," *Technical Report, CADRU-02-88*, CAD Research Unit, Design Institute, School of Architecture and Environmental Design, Cal Poly, San Luis Obispo, California.

Pohl, K. (1992). "GXI: A Graphical Interface Builder with Distributed Inter-Process Message Management Capabilities," Focus Symposium on Computer-Based Design Environments, 6th International Conference on Systems Research, Informatics and Cybernetics, Baden-Baden, Germany.

Reitman, W. R. (1964). "Heuristic Decision Procedures, Open Constraints, and the Ill-Defined Problems," in *Human Judgements and Optimality*, Shelley and Bryan (eds.), Wiley, New York, 282-315.

Reitman, W. R. (1965). *Cognition and Thought*, Wiley, New York.

Rittel, H. and M. Webber (1984). "Planning Problems are Wicked Problems," Cross (ed.), *Developments in Design Methodology*, Wiley, New York, 135-140.

Rowe, P. G. (1987). *Design Thinking*, MIT Press, Cambridge, Massachusetts.

Schon, D. A. (1983). *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York.

Schon, D. (1988). "Designing: Rules, Types and Worlds," *Design Studies* **9** (3): 181-190.

Simon, H. (1984). "The Structure of Ill-Structured Problems," Cross (ed.), *Developments in Design Methodology*, Wiley, New York, 145-166.

Simon, H. (1981). *The Sciences of the Artificial*, MIT Press, Cambridge, Massachusetts.

Stevens, W. (1990). *UNIX Network Programming*, Prentice Hall, Englewood Cliffs, New Jersey, 137-169, 258-339.

Sunderam, V. (1990). "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice & Experience* **2** (4).

Wade, J. W. (1977). *Architecture, Problems and Purposes*, Wiley, New York.

Williams, C. (1985). *ART - The Advance Reasoning Too*l, Inference Corporation, Los Angeles, California, 20-3.