# Office Semantics

by

Gerald Ramón Barber
B.S. Electrical Engineering, University of Idaho 1975
B.S. Mathematics, University of Idaho 1975
M.S. Electrical Engineering, University of Idaho 1976

**Submitted in partial fulfillment of the requirements for the degree of**

**Doctor of Philosophy**

at the

**Massachusetts Institute of Technology**

February 1982

Signature of Author . . . [illegible] . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
21 December 1981

Certified by . . . . . . . [illegible] . . . . . . . . . . . . . . . . . . . . . .
Carl E. Hewitt

Accepted by . . . . [illegible] . . . . . . . . . . . . . . .
Chairman, Departmental Committee

# Office Semantics

by

Gerald Ramón Barber

Submitted to the
Department of Electrical Engineering and Computer Science
on 25 January 1982 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

## Abstract

Office Semantics is the study and explanation of organizational behavior with the intent of understanding the problem solving processes behind the physical and mental actions occurring in the performance of tasks. The goals of Office Semantics are to understand what people do in organizations and *why*--the semantics of their actions. The reasons behind the actions are couched in terms of the office workers' knowledge of their environment. This knowledge has two parts: the *organizational* knowledge, both the informal and formal social structure of the organization; and the *application* knowledge, the explicit subject domain of the organization.

A premise of Office Semantics is that organizations are goal oriented mechanisms. Office work is viewed as a problem solving activity. The result of office work is the outcome of the problem solving activity. However, problem solving in the office does not conform to the classical AI paradigm in which a computer solves difficult intellectual problems by search in a state space with a well defined initial state, well defined final state, and a given set of state transformers. We offer an alternative paradigm in which problem solving is *supported.*

An implementation of the Omega description language is discussed along with a Viewpoint mechanism. Viewpoints are a mechanism for containing contradictions, thus when a contradiction is reached reasoning can proceed outside the viewpoint as to why the contradiction was reached and what should be done about the contradiction.

# Acknowledgments

I would like to thank Carl Hewitt for sparking my interest in the topic and for serving as an invaluable source of insightful questions and clarifying suggestions. Carl's push to simplify the complex has deeply influenced this work.

Thanks go to my readers Randall Davis and J. C. R. Licklider for making several suggestions and for pointing out other work related to that presented here.

For sharing many discussions about Omega and many other topics thanks go to Giuseppe Attardi, Gene Ciccarelli, Stan Curtis, Peter de Jong, Bill Kornfeld, Henry Lieberman, Maria Simi, and John Teeter.

For supporting me through what seemed to be an interminable process I would like to thank Peter Droege, Jim and Beth Erdman, Steph Schroeder-Teeter, and Clay and Joel Schroeder-Teeter. I could always count on them to bring me back to reality.

I would like to thank my parents Margaret and Jim and my brother Steve, who have helped so much for so long. I would also like to thank my other family, John and Mary Brockman along with Roberta, Jim, and Nancy.

Lastly and most importantly I would like to thank my wife, Marilyn Brockman. You did so much, I don't think I could have done it without you.

This work is dedicated to the memory of

Scott Tomás Barber,

may he rest in peace.

# Table of Contents

# Table of Figures

# Chapter One

# Introduction

Office Semantics is the study and explanation of organizational behavior with the intent of understanding the problem solving processes behind the physical actions occurring in the performance of tasks. One of the goals of Office Semantics is to understand what people do in organizations and *why*--the semantics of their actions. The work people do is described using the description system Omega. Omega is a description system used to embed knowledge within a workstation and to aid workstation users in problem solving activities.

## 1.1 Office Semantics and Omega

Although Office Semantics and the description system Omega may appear to be separate unrelated fields of study their co-development has had important influence on each other. Omega is developed as a knowledge embedding language for use in real world situations. This contrasts with limited application expert systems that have been applied to domains such as medical diagnosis, geology, or organic chemistry. Office Semantics is developed in order to explain the structure and behavior of organizations in terms of their problem solving functions.

This dissertation investigates the use of the knowledge embedding language Omega in an office environment. Office semantics provides the theoretical foundation within which to explain organizational behavior. Omega is used to express the concepts of Office Semantics in a concrete and precise form. The interplay between the development of Office Semantics and Omega is important as each functions as a test bed for the other. Office Semantics provides a domain within which to test the knowledge embedding facilities while the abstract ideas of Office Semantics are tested and take concrete form in Omega.

## 1.2 A Disclaimer Concerning Office Automation

This dissertation is not about Office Automation. Office Automation is often taken to mean the replacement of office workers with technology, or the reorganization of office work into a style suitable for automation. It is the opinion of the author that this is neither possible given current technology nor desirable. Automation is possible when an environment can be carefully controlled. An example is a production line: the design of parts is carefully controlled; the flow of parts down the production line is controlled; the workstations where workers perform assemblies are carefully designed; tasks the workers must perform are carefully designed to take into consideration the workers' physiological capabilities. That it is not possible to carefully control office work is an inherent characteristic of most office work.

This dissertation is about developing a model of what office work is and developing tools that office workers can use in their work. The position taken is not that people should be replaced with technology but that their capabilities should be augmented with technology.

## 1.3 The Problem

An example will introduce the context and the issues of concern in this dissertation. The scene is an office in the Defense Department that is part of the Officer Transfer Process. This process is the method by which Navy officers are reassigned to tours of duty or *billets*[1] when their present billet assignment expires. The *Assignment Officer* fills the role depicted in figure 1-1 below.

[1] Billets are jobs, an officer is usually assigned to a billet for 3 years.

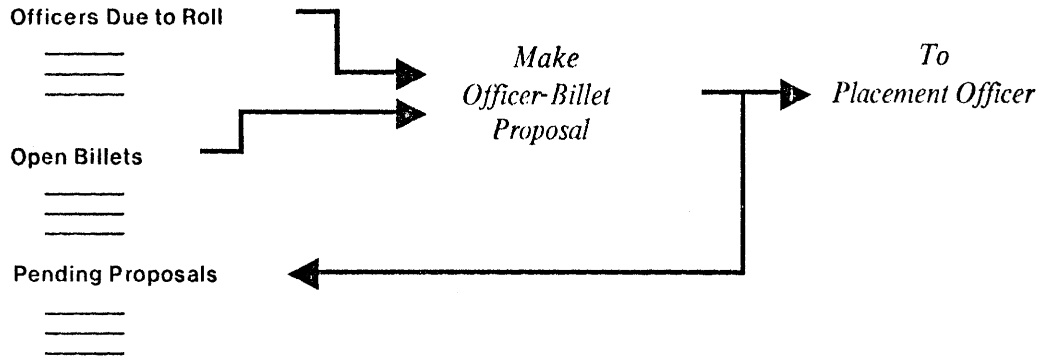


**Figure 1-1:** The Assignment Officer Role

Conceptually the Assignment Officer's role is simple; he or she has a list of officers that are *due to roll*[2] and a list of open billets. The assignment officer chooses a officer-billet pairing and passes this proposal on to the *Placement Officer* for acceptance and keeping a record of the proposal. The Placement Officer accepts or rejects the proposal. The assignment officer represents the interests of the officers that are due to roll. Thus in each officer-billet proposal the Assignment Officer chooses a billet that will help attain the career objectives of the officer due to roll.

The story begins:[3] we find Assignment Officer Watkins in a quandary; Officer Daniels has just called in from the field wondering why he hasn't received his travel orders as his current assignment expires in 6 weeks. Daniels is justifiably concerned; his wife is about to have their third child and his oldest daughter is ready to start school. Officer Daniels knows he must move in six weeks but he doesn't know where or even whether his family can accompany him.

After searching about the office Assignment Officer Watkins finds Daniels' papers. They have been misplaced and as a result Daniels has not even been proposed for his next duty station yet. All is not lost, Watkins finds the perfect billet for Daniels. Daniels won't have to move since the billet is in the same area where he is stationed now. However there is a hitch, Daniels needs schooling to be eligible

[2]An office is due to roll when his or her current billet assignment will expire in 6 months.

[3]This description of a typical situation faced by an Assignment Officer is based on interviews the author conducted with Assignment and Placement Officers in 1978 and 1979. The author wishes to express his grateful appreciation to those officers that took time out of their busy schedules to patiently and frankly answer the authors many questions. The names in the examples presented are fictitious, the examples themselves are based on fact.

for the new billet and the school is full. Watkins remembers that a friend of his is associated with the school, a quick phone call establishes that there has just been a cancellation and Daniels can attend the school. Watkins quickly put together a proposal. He found the Placement Officer in charge of the billet, he explained the situation and he received approval for the proposed transfer. In a matter of hours Daniels' orders are being processed and Watkins calls Daniels to give him the good news.

There are 3 questions that we can ask that will be investigated in this dissertation.

1. **What did he do?** How best can one describe the work that Watkins did?

2. **How did he do it?** How did Watkins do his work, what kind of knowledge did he use in accomplishing what he did.

3. **How can he be helped?** What tools could Watkins use in performing his work. What jobs could a computer do that are currently time consuming annoyances for Watkins; what tasks should Watkins continue to do?

What did he do? In short Watkins did a lot of *problem solving*. Watkins found Daniels' papers, realized they had been lost and went to work on them immediately. He deduced what would be a good job for Daniels given his current situation and he circumvented the problems that presented themselves when insuring that Daniels was qualified for the new job.

How did he do it? Watkins used *Organizational knowledge*, both of the formal structure of the organization and of Watkins' informal social relationships and *Application knowledge* of the domain in question to solve the problem. Watkins used his knowledge about the Officer Transfer Process, about the requirements for billets and about Daniels current needs. Watkins' relationships with the Placement Officer and the officer at the school were also important.

Watkins needs tools that can support his problem solving. He needs tools that will let *him* make important decisions but that will take care of the details. The tools must be able to reason about the organizational structure and about the application structure of a particular office. This dissertation is about developing these tools.

Let us be more specific. Watkins needs a knowledge embedding system that can reason about change and contradiction. Existing AI systems have trouble with change. This is discussed in chapter 6. Popular logics have trouble with contradictions: if a contradiction is derived in the logic then anything can be derived. In a sense the logic *crashes*. This dissertation is about a logic that has valid inference rules, even though it may derive contradictions. The *Viewpoint* mechanism is used to manage contradictions. Reasoning proceeds in a viewpoint, when a contradiction is reached it is reached within the viewpoint and thus its effects are contained within the viewpoint. The allows the reasoning system to function outside the contradictory viewpoint, in a consistent viewpoint, and analyze the contradictory viewpoint.

## 1.4 The Contributions of this Work

The main contributions of this dissertation are:

- Office Semantics: organizations are macro-intelligent systems, their behavior can be explained in terms of the problem solving reasons behind the actions in organizations.

- Work in the office can only be understood and adequately described when the office worker is considered as both a rational being and as a social being.

- Office work is a problem solving activity. The services and products provided by the office are achieved by the office's knowledge of the domain in question and by the office's problem solving capabilities. Therefore, the most valuable tools for individuals in an organization are tools that support the individuals in their knowledge-based problem solving activities.

- Omega's Viewpoints provide a powerful tool for reasoning about change and contradiction. Contradictions can be confined to a viewpoint and the deduction machinery can reason about why the viewpoint is contradictory from outside the viewpoint.

- Description directed Sprite invocation is a more appropriate and efficient method for pattern directed invocation than traditional systems. Description directed Sprite invocation helps the efficiency of the Sprite system by using Omega's inheritance network to turn broadcasts into point-to-point transmissions.

## 1.5 Preview of this Dissertation

The chapters in this dissertation proceed from the more abstract to the more concrete. Chapter two provides a discussion of the relation between Office Semantics and Artificial Intelligence; this chapter is of general interest. The reader interested in the issues related to Office Semantics, describing organizational behavior, and the model of office work will be most interested in the chapters 3 and 4. Chapter 5 is an introduction to the description system Omega. It contains a review of some material that has appeared elsewhere as well as original material. This chapter along with chapter 6, which introduces and discusses the viewpoint mechanism will be of most interest to those readers interested in the knowledge embedding facilities provided by Omega. The implementation of the Omega description system on MIT's Lisp Machine is described in chapter 7. Chapter 8 is also of general interest; it is an extended example showing how the ideas about organizational work are embedded in the Omega description system. The final chapter, 9, is the conclusion.

A final note concerning the use of terms: in this dissertation the terms office and organization are used interchangeably. These terms are taken to mean organizations which have the primary function of gathering, processing, recording and disseminating information. These organizations are to be distinguished from organizations which have the primary purpose of producing some physical product such as an assembly line in a factory.

# Chapter Two

# AI and Office Semantics

This chapter addresses the question "How are AI and the study of Office Semantics related?" Although one may initially believe that there is little in common between these two fields, and historically little cross-fertilization has existed, we argue that AI and Office Semantics have much in common. The discussion in this chapter is a preview of the rest of the dissertation. This chapter provides an overall picture, a map of relationships, between the various issues that are described in detail in the body of the dissertation. The question of the relationship between AI and Office Semantics is addressed along the following lines:

- **Common Issues** - Many issues in AI and Computer Science are similar to those found in organizations.

- **Organizations Exhibit a Kind of Intelligent Behavior** - Organizational behavior has not been extensively studied by AI researches even though organizations exhibit a kind of intelligent behavior. In considering intelligent behavior *micro-* and *macro-intelligent* systems are distinguished.

- **Organizational Work is a Problem Solving Activity** - Current organizational theories cannot explain why organizational workers do what they do. The tasks organizational workers do can be explained if they are viewed in terms of their problem solving capabilities.

- **Society Theories in AI** - The study of organizations is relevant to the *society* theories in AI. In these theories [Kornfeld, Hewitt 81, Minsky 77, Steels 79] an entity's intelligent behavior--be it an individual or scientific community--is understood in terms of a society of cooperating agents.

- **Organizations are a Testbed for AI** - Organizations are a testbed for ideas about the structure of intelligence. Organizations may be analyzed, metered, structured and restructured in ways that is not possible with individual humans.

- **Work vs Work Technology** - Understanding the work that needs to be accomplished and how tools can accomplish that work is important in both AI and Office Semantics.

## 2.1 Common Issues

Many issues that arise in Computer Science and Artificial Intelligence also arise in Organization Theory. One reason for this is that many of the issues in Computer Science arise because computer systems are used in an organizational setting. Some examples of important issues in Computer Science and Organization theory are: reliability, the paychecks must go out on time; robustness, if there is a failure in a sub-system, computer or organizational, it should have a minimum affect on the overall system; and security, personal information about employees must be kept confidential. These are common issues because they are at the interface of the two disciplines.

The general issue of distribution vs centralization of resources is important in both Computer Science and Organization Theory. The advent of inexpensive hardware such as microprocessors and communication networks has stimulated interest in the distribution of computer resources. This same technological force is responsible for stimulating interest in decentralized organizational structures--the distributed office. But the interest in centralization vs distribution in the organization has a longer tradition. The advent of batch computer systems in the 50's and 60's forced centralization of many organizational functions such as accounting and inventory control. However, distribution is not merely an organizational response to technological forces but a fundamental characteristic of the organization. An organization is a distributed mechanism since it involves many people--sometimes geographically dispersed--cooperating to achieve a goal: sell a sewing machine, plan a corporate strategy, or negotiate a construction contract.

Decentralization in an organization implies multiple concurrent activities and the subsequent necessity to synchronize and schedule these activities. Project engineers use PERT charts, managers use appointment books and accountants use logs and audit trails to manage concurrent activity. In Computer Science a parallel concern with concurrency exists. The literature is rich with research on parallel problem solving techniques, mechanism for synchronizing concurrent processes and so on. In comparing the two fields similar approaches are frequently found; many computer science paradigms are derived from our experience with the real world.

Both AI and Organization theory face the problem of controlling complexity. One technique organizations employ to control complexity is specialization. An organization, faced with with the task of making and selling a product for example, consists of specialized departments that fulfill a variety of needs: accounting, production engineering, sales and customer service. Similarly, specialization is a popular AI paradigm, in the name of classification hierarchies, to control complexity. March and Simon explain the structure of organizations in terms of the principle of *bounded rationality* [March, Simon 63], the need for humans to simplify complexity because of the limits of their rationality. This need to simplify complexity explains the division of an organization into specialized subparts.

There are other concerns that Organization theory and AI have in common: the problems of adapting to a changing environment; the use of knowledge, its manipulation and representation; control structures that insure that the parts of an organization/computer system function in harmony to achieve the coherency of the whole. The list of common issues, being long and intertwined, prompts one to ask why so little interplay exists between the two fields. The answer is that Organizations theory has been concerned with the abstract/human side of the issues and AI and Computer Science has been concerned with the more concrete/technical side of the issues. However this distinction is quickly disappearing for two reasons: first, computers must be more human-like to be usable by line managers and clerks with a minimum of training, hence the more abstract/human issues must be considered; second, AI systems, such as Omega, are rapidly becoming practical--they will support human problem solving in the organization.

## 2.2 Organizational Behavior and Intelligence

Organizational behavior is a type of intelligent behavior; it includes such activity as problems solving, knowledge acquisition and manipulation, and adapting to a changing environment. One may speculate that organizational behavior--and work in the organization--is highly constrained and thus easily describable. However this is not the case. AI programmers cannot simulate organizational behavior given current programming methodologies and tools; neither can AI theories adequately explain organizational behavior. In one respect this is not surprising: few AI researchers have applied their theories to explain organizational behavior, one notable exception is [March, Simon 63]. However, a common AI research methodology is to examine behavior that may be considered intelligent and to develop theories that explain this behavior. Various human behaviors such as natural language understanding, problem solving and knowledge acquisition and manipulation have been studied to this end. But AI researchers have not studied non-human systems, such as organizational behavior, that exhibit complex behavior--behavior that would be considered intelligent in humans.

AI researchers have focused on *micro-intelligent* systems, humans and animals for example, as opposed to *macro-intelligent* systems such as organizations. The distinguishing criteria between micro- and macro-intelligent systems is based on the difficulty of analyzing the systems' component parts. Observing human problem solving mechanisms is difficult: either one must deduce the problem solving mechanisms from physical acts--filling out forms, verbal communication--or one must instrument the brain. Instrumentation, of course, is also collecting phenomenological evidence albeit of a more direct nature. Direct evidence such as electrical signals from neurons is even more difficult to interpret than physical acts since it is separated from the task at hand by several levels of abstraction. Instrumentation also presents serious ethical and practical questions. Micro-intelligent systems are also difficult to manipulate; restructuring micro-intelligent systems or changing their components is not possible. cost/benefit analysis. Even the goal itself maybe discarded or reformulated which is a behavior that is not a part of the state-operator paradigm. Another problem with the state-operator paradigm is that it assumes a single problem solver. In an organization many individuals cooperate in solving a problem.

Instead of the state-operator paradigm we propose *Congruence Analysis.* This paradigm accommodates multiple problem solvers. Emphasis is removed from establishing a particular goal. In Congruence analysis the emphasis is on finding a *satisfactory* goal that can be achieved.

Organization theory can also benefit from applying the paradigms of AI to the understanding of organizational activity. Indeed it already has in the work of March and Simon. Although they do not call it by name the use of the GPS style problem solving paradigm [Newell, Simon 72] is applied in explaining the behavior in organizations. A production system technology and means-ends analysis is used to explain behavior in organizations. The revolutionary contribution of Simon's work that led to his 1979 Nobel prize in Economics is based on considering the individual in the organization as a problem solver of limited capacity; this is an application of the state-operator paradigm.

## 2.3 Society Theories in AI

The study of organizational systems is relevant to the current interest in the scientific community, society and communicating experts metaphors in AI research [Kornfeld, Hewitt 81, Minsky 77, Steels 79]. In these metaphors it is assumed that the complexity and sophistication of human intelligence arises out of interactions between simple entities or entities of a limited domain of expertise. This is a metaphor readily adaptable to the study of organizations.

A possible objection to the idea that the complexity of organizational behavior arises from the interaction of simpler parts is the claim that the ability of organizations to function as they do comes from the individuals in the organizations. The individual intellectual capabilities of humans are already highly developed, by combining them into organizational structures we are not producing a more sophisticated entity from simple parts but just a sophisticated entity from sophisticated parts. We gain nothing in our attempts to explain the makeup of sophisticated entities. Nothing is gained by virtue of there being an organization and one can even argue that some of the flexibility of human intelligent behavior is lost. However, it is certainly the case that a large organization performs activities that are much more complex and on a larger scale than what an individual could do, both in terms of physical and intellectual feats. Consider building a DC-10, writing tax legislation or developing a scientific theory.

Organizations are, at the very least, an existence proof that societies of communicating experts can be organized in ways that can accomplish useful work that is beyond the capacity of any individual in the organization.

In AI there is currently an interest in developing computer systems that can aid individuals in accomplishing very complex tasks, such as VLSI design. These systems transcend the traditional functions of computer based design tools which consist of analysis tools and database tools for creating and maintaining graphic or textual information about the design. Complex tasks such as VLSI design require high level facilities for management of the design, in a similar way that architectural or engineering firms manage their projects. Computer systems that support complex tasks must accommodate and record incremental change, allow for a mobile work force, provide project management facilities such as PERT charts, provide cost accounting facilities and provide project status functions.

## 2.4 Organizations as a Testbed for AI Theories

Organizations, or macro-intelligent systems, can be used as a testbed for theories in AI in two ways. First, AI theories can be used to explain the intelligent, problem solving behavior of organizations. Second, AI theories on problem solving can be used to build systems that office workers use in their day to day work.

Explaining the intelligent behavior of organizations is methodologically promising because organizations are accessible in a ways that humans are not. It is possible to examine the workings of an organization in more detail than it is possible to examine the processes by which a human solves a problem or understands natural language. An organization can be metered, analyzed and experimented with in ways that are not possible with humans. Hypothetical organizational structures can be implemented and examined.

There is a continuum of scale when considering organizations that is not present with humans. At one end of the scale we have an organization composed of a single human. At the other end are organizations composed of many thousands of individuals. This continuity is interesting from at least two points of view. First, we may see how functions present in individuals can be implemented using groups of individuals when the complexity or scope of the functions exceeds the capacity of a single individual. Second, we see various ways in which the functions that organizations perform can be factored as the size of the organization increases in the presence of differing demands on the organization.

## 2.5 Work vs Work Technology

One can ask the questions "What have the years of study in Organization Theory produced?" "What can Artificial Intelligence contribute?" "Is the wheel about to be reinvented again?" To answer this question we consider the following view of organizations. There is a kind of work that organizations--especially information intensive organizations such as offices--perform and there is a technology by which this work is accomplished. By and large the technology by which the work is accomplished has largely consisted of paper-based and verbal communication, paper-based storage of information, and the members of the organization. The relation between the work that offices accomplish and the technology used to accomplish the work has not been of concern because it has changed until recently. Thus Organization theory has not dealt with the question of the relationship between work in the office and how it is done. Much can be gained by examining the work in the office as knowledge manipulation and problem solving activity.

The relationship between work and work technology has been an issue in more routinized, production line style, non-information related tasks. There has been much study in the name of Management Science and Industrial Engineering as a result. Within the office there has been the use of centralized computer facilities for accounting and inventory. These functions have a highly structured and rigid interface to the workers in the office. In their capabilities they are extensions of the paper based systems. Technology that has changed the work in the office has been limited to devices such as the batch computer facilities, telephone, typewriters and recently, word processing. The introduction of each of these has affected the way office work is done. The impacts have been handled on a case by case basis; no theory of what is happening when new technology is introduced exists. The unpredictable results of the efforts to introduce word processors into the office is testament to the fact that both the relationship between technology and office work is not well understood and that office work itself is not understood. In the cases of the technologies mentioned above the work in the office, the thinking, the knowledge processing, has not been affected in any significant way. It certainly has not been as drastically affected as it will be in the years to come.

## 2.6 Summary

The study of organizational behavior and AI are ripe for cross fertilization. The two fields have many issues in common. Organizations exhibit a kind of intelligent behavior that AI theories should be able to explain; to some extent this has begun to happen. AI theories can be tested in the organizational setting more readily than in humans. Organizations are also a source for ideas on how complex tasks can be accomplished by cooperating experts. The research described in this dissertation is the product of much mutual influence between AI and Organizational theory.

# Chapter Three

# Office Semantics

The theme of this chapter is that the characteristics of organizational behavior, referred to as *Office Semantics*, derive from two sources. The first source is the organization's workers' rational abilities: their problem solving capacities and their ability to make rational choices. The second source is the setting of the office: office workers are social beings cooperating in a social environment to accomplish their work. These two sources are sometimes in conflict, sometimes complimentary; taken together they can form a firm foundation on which to build a theory of organizational activity.

## 3.1 Background

Previous approaches to understanding organizational behavior have considered the individual in the organization in different ways. The *mechanistic* view is best exemplified by the work in Scientific Management [Taylor 47]. In this approach it is assumed that a worker's activities can be described in terms of overt physical actions. The triumph of Scientific Management was that some types of work could be adequately described in terms of overt behavioral movements. This approach is adequate for describing such tasks as assembling farm tractors but is inadequate for information processing tasks such as evaluating a load application or proposing an officer for a new tour of duty.

The individual as an *optimal decision maker* is the view of *Administrative Management*. Here the individual is seen as being faced with a given set of alternatives and a utility function by which he or she ranks the alternatives. The individual makes the optimal decision given the alternatives and the utility function. This approach is tailored for use with the tools of Operations Research. The major criticism of this view of the individual is that in reality people do not know all possible alternatives to a given decision. Operations Research is not properly applied to the organizational setting because a utility function is usually not available with which to evaluate each alternative.

The individual as an *informed decision maker* presents the office worker as a decision maker given alternatives from which to choose. For the informed decision maker emphasis is placed in the information flow between decision makers. In this case not much is said about the criteria upon which the decisions are made. The specifics of the decision making process is not treated nor is the effect of the kinds and timeliness of information on decision making addressed. The emphasis is on designing and analyzing information transport systems. Zisman's SCOOP system [Zisman 77], information flow models such as Information Control Nets [Ellis 79], and [Sandewall 79] and forms flow models [Tsichritzis, Hudyma 80] are examples of this approach.

In [March, Simon 63] the individual is presented as a *rational decision maker of limited capacity.* Emphasis is on the fact that the individual is limited in his or her cognitive abilities. The individual's activity is characterized as problem solving. The ODYSSEY system [Fikes 80a] is an example of a system that treats the individual as a *rational problem solver* and provides support in problem solving activities.

Many sociologically based theories also exits. Systemic approaches such as [Katz, Kahn 78] consider the organization as a open system[4] in analogy to biological system. Office work as *Practical Action* has been proposed by [Suchman 79]. This approach considers the problem of applying general office knowledge to specific instances in the social context of the office. Conversation as an information media is focused on in [Wynn 79]. In this study the characteristics of conversation as a means of acquiring and disseminating information are explored. All these approaches treat the individual as a *social being* in various manners in the organizational context. These approaches do not consider the characteristics of the work the individual performs but only his or her interactions with fellow workers.

These views fall into two basic categories: those that treat the individual as a decision maker of some sort, focusing on the subject matter of the decisions and those that emphasize the individual's social relationships. A thesis of this dissertation is that in understanding organizational behavior a human must be treated as both a rational problem solver and as a social being. To consider one without the other in explaining organizational behavior is difficult and misleading.

[4] An open system in one that is modeled as existing in an environment. The system interacts with the environment, consuming resources and producing products.

## 3.2 Office Semantics

Office Semantics is the study of information intensive organizational work. Its name reflects the concern with *the intent behind the act.* Office Semantics is concerned with understanding the reasons behind the physical and mental tasks that are performed in organizational work. To understand organizational behavior a distinction is made between the *application structure* of the organization and its *organizational structure.* As illustrated in the diagram below the definition of Office Semantics hinges on this distinction. The remainder of this chapter is concerned with what the application and organizational structures comprise and what each contributes to explaining organizational behavior.

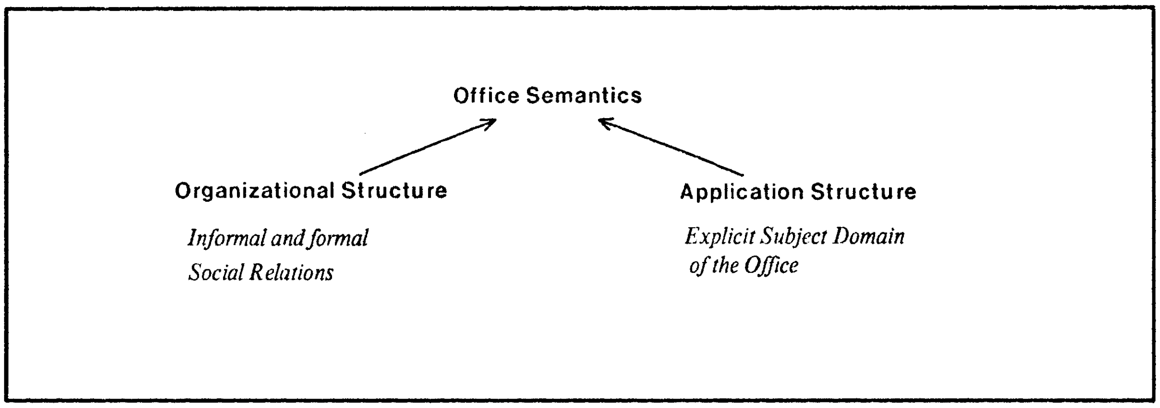

Office Semantics
Organizational Structure
Informal and formal Social Relations
Application Structure
Explicit Subject Domain of the Office

**Figure 3-1:** Office Semantics: Application and Organizational Structures

It is beyond the scope of this dissertation to investigate in depths the issues related to the organizational structures of Office Semantics. Many of these issues are treated by fields such as group dynamics and individual psychology. We argue that these issues must be treated before a satisfactory understanding of organizational behavior can be achieved. The main point of the argument is that organizational factors have a direct effect on the performance of the organization, thus they cannot be ignored in understanding organizational behavior. We do not, however, investigate the effect of organizational structures in any depth on the behavior of organizations.

A fundamental premise is that office work is a problem solving behavior. The rest of this chapter is developed with this premise and the following goals for Office Semantics in mind.

**Description** - Office Semantics should present a model that is powerful enough to describe the phenomena of importance in Organizational Behavior.

**Explanatory Power** - Office Semantics should explain the structure of organizations as they exist.

**Predictability** - Office Semantics should be capable of predicting the effects of changes on organizations.

**Role of Technology** - Office Semantics should clarify the role of technology in the organization. In particular Office Semantics should explain how technology is used in the accomplishment of office work and how new technology influences the organization.

**Character of Office Work** - Office Semantics should provide an explanation of the nature of office work and how office workers perform office work.

With the above goals in mind for Office Semantics the distinction between application structure and organizational structure is put forth as a foundation from which to proceed. Below a brief description of these two structures is given followed by sections which discuss them further.

The social structure of the organization will often be referred to as simply the *organizational structure.* This is concerned with the aspect of an office which stems from the fact that the activity in an office is realized by people cooperating in a social system. The organizational structure includes both the formal organizational structure and the informal structure of social relations between the members of the organization. The informal social structure is of interest for as pointed out in [Browner, Chibnik, Crawley, Newman, Sonafrank 79] a system of formal controls and lines of authority in an organization has a complementary structure of informal relations among the office workers.

In contrast to the organization's social structure is its application structure. The application structure concerns the subject domain of the office. The application structure comprises the rules and objects that compose the intrinsic functions of a particular office system. For example, in a Credit and Loan office the application structure includes such entities as loans, credit ratings and rules such as criteria for accepting or rejecting loans. The application structure of an insurance company is concerned with insurance policies, claims and actuarial tables. The application structure explains the scope of the functionality of an office system on a subject domain as well as providing a model by which those functions are characterized. The application structure is, overtly, the primary reason for the existence of the office.

## 3.3 Organizational Structure

The organizational structure is of interest because social factors directly affect the behavior of an organization and because changing technology influences both the formal and informal social structures in an organization. Unfortunately, the study of organizational structure, especially informal social relationships, has been neglected in past efforts to introduce computers into the office.

As previously mentioned Organizational structure encompasses both the formal social structure of the organization and the informal social relations that arise between organizational workers. The formal organizational structure includes the authority relationships between workers and the departmentalization of an organization into subunits. The informal social relations consist of the ties coworkers form and include knowledge about coworkers with respect to the working environment.

### 3.3.1 Effects of Social Factors on Organizational Performance

The performance of an organization is directly influenced by the informal social structures among its members. For example:

- The decisions individuals make that affect their coworkers are based in part on the social relationships between the workers. They include the individuals' trust in their coworkers, their assessment of their coworkers' competence, their beliefs about what their coworkers know and their knowledge of their coworkers' habits.

- Pools of office workers, where each worker is performing the same task tend to form their own informal social hierarchies. The more experienced and skilled workers tend to be accepted as the informal leaders and representatives of the groups. These informal leaders are the ones most likely to form working relationships with managers of the work pools. Via these relationships decisions are made and strategies and policies are developed.

- When individuals depend on each other to accomplish the same goals informal working relations are strongest and the common goal is most easily accomplished. In the case where the relationship is less bidirectional, establishment of the goal becomes a more difficult task; to the point that formal sanctions may be necessary to insure that the goal is accomplished properly and in a timely manner [Browner, Chibnik, Crawley, Newman, Sonafrank 79].

- When a worker is introduced to unfamiliar technology he or she must learn about the technology as well as new dependencies and informal understandings. Workers generally learn this kind of information from more experienced members of the office. In the case of new technology there may be no experienced members and a learning period in which the dependencies and understandings are evolved must be entered.

Each of the above are examples where social factors influence decision making and thus influence the behavior of the organization. The first and second concern coworkers' knowledge about each others capabilities. The third is concerned with workers' relations in the context of organizational goals. The last concerns the worker's relation to office technology.

### 3.3.2 Conflicting and Common Interests

An important problem solving paradigm in AI is the use of proponents and skeptics in the process of achieving a goal [Kornfeld, Hewitt 81]. In this metaphor, when a goal is to be achieved, both proponents and skeptics are put to work on the goal. The aim is that the proponents attempt to achieve the goal while the skeptics attempt to prove that the goal is not achievable or narrow the goal to one which is achievable. If the goal is shown to be unachievable then no more resources are expended on achieving the goal. The intuition behind having skeptics is that the skeptics can help focus attention on promising possibilities. In the case where the goal is to maintain some constraint the proponents and skeptics work in balance with each other, each does its part to establishing the constraint and checking that the others do not violate the constraints.

This paradigm is found in organizational problem solving. It is frequently found as a system of checks and balances or *controls* between offices charged with advancing somewhat conflicting interests. An important strategy for maintaining balance is to establish separate groups in an adversarial relationship within an organization to look after conflicting interests. Policies are then established and evolved by negotiation. This strategy is often used in preference to the alternative of attempting to have one group attempt to rationally balance the conflicting interests.

Accounting systems are an example where controls are maintained by adversarial relationships between different groups. For example, accounting systems are required to have certain controls by law. As a result some proposed computerized accounting systems which do not maintain these controls would be illegal to use. This requirement influences the design of office system by placing a constraint on information flow and requires that office systems be designed so users cannot violate these information flow and authority constraints [Bailey, Gerlach, McAfee, Whinston 81].

Systems of common interest are used to advantage in offices. It has been noted [Browner, Chibnik, Crawley, Newman, Sonafrank 79] that workers cooperate better and form strong social relationships if they share the goals of a task and are mutually dependent on each other to achieve the goals. Care must be taken to avoid inadvertently upsetting these systems of controls and dependencies. New technology, by changing the social fabric of the organization as well as by information flow can have this effect.

Structured conflict affects the social structure of the organization. Designers of organizations often structure members into competitive groups. This mock conflict strengthens ties between group members by creating shared goals among the group members.

### 3.3.3 The Impact of Technology on the Organization

The organization can be metered in scope and detail that was not previously possible. Mass storage technology is such that large quantities of data can be inexpensively stored compared to paper based storage methods such as file cabinets. This simply means that the volume of information that can be kept for the same price is larger. This trend will continue in the future.

### 3.3.4 Measurement of Performance

Organizational performance is difficult to measure. This is because descriptive models of the office have been weak in their expressive power in either the application or organizational structures. Many of the problems are described below.

1. Organizational performance is difficult to measure for the same reason office procedures are hard to describe. This is discussed more fully in section 4.4 on page 38. The problem is that mental processes are very difficult to discern from an individual physical actions.

2. New and unexpected situations are the norm in an office. Measuring performance in an unfamiliar situation is difficult. A descriptive system must accommodate open-ended situations.

3. Reactions to situations may be dictated by powers beyond the individual's control such as organizational policy or laws. Metering must take these non-local influences into consideration. The application as well as the organizational constraints need to be considered when measuring performance.

4. In many cases a long time constant is involved in discerning the results of an action on a organizational performance.

This information is useful for *regulatory functions* which gear organizational work to certain factors such as production demand. Performance information is also useful for the *adaptive* purposes which seek to help the office evolve so that it may continue to survive in a changing environment. However, care must be exercised about what information is kept and how it is interpreted.

Numbers are exceedingly easy to collect in an electronic office system, but if these numbers are used to drive an adaptive or regulatory mechanism it is essential that an attempt be made to analyze the effect on the future behavior of the office. If this is not the case the resultant behavior may not reflect goals of the organization.

A major problem here is that there is little understanding about how offices work in their day to day operation. Initial performance measurement often points out surprising discrepancies between the believed and actual office performance characteristics. As [Browner, Chibnik, Crawley, Newman, Sonafrank 79] point out, the temptation to enforce a particular behavior on an office must be resisted until the implications of the change are well understood. This is particularly true in regard to the effects of an enforced behavior on the social structure of an office.

## 3.4 Application Structure

In section 3.2 organizational work as problem solving is advanced as a premise for Office Semantics. The application structure is the model in which many of the organizational goals and constraints are expressed. The application structure also furnishes actions by which constraints may be maintained and goals may be achieved. The importance of the Application Structure to office work is central--it is a description of the organization's work neglecting the details of the organization accomplishing the work.

The Application Structure is what most non-sociological attempts to describe organizational work discuss.

1. The application structure is the model by which the objects of concern in the office are characterized. The application structure also provides rules that indicate how the objects of concern to the office may be manipulated.

2. Rules that pertain to objects in an office may be derived locally in the office itself, may be derived from organizational rules or policies or may be laws governing acceptable practices such as accounting laws.

3. The application structure includes the records, accountability, and audit trails that are used to classify and describe past actions in an office.

For example, in the Officer Transfer Process described in the introduction the application structure is made up of records describing the officers, records of billets and their requirements, and financial and budgetary records. Application structure rules are those that control under what circumstances officer's may occupy billets, when they may be suggested for promotion and when the may be enrolled in schools.

# Chapter Four

# The Nature of Work in the Office

## 4.1 Introduction

A useful characterization of office work is as a goal oriented activity, therefore, work in the office can be viewed as problem solving in order to establish goals. The individuals' work contributes toward the organization's goals; the organization as a whole can be viewed as a problem solving mechanism.

In order to understand the office worker's activity a model of problem solving is needed. This model differs from the AI state-operator problem solving paradigm in subtle ways.

## 4.2 Some Fundamental Problems

As seen in chapter 3, individuals within an organization are considered more than instruments in the functioning of the organization: they are considered as rational beings with limited cognitive capabilities. The individuals' rationality is with respect to a subjective reference frame corresponding to the simplified model of what they know about their world. The processes of determining the alternatives to a decision, judging the outcomes to each of those alternatives and comparing the alternatives to make a choice are resource consuming tasks. In most cases all the alternatives to a decision are not known and the outcome to any particular choice is not fully known.

Given the above mentioned assumptions about people in organizations and the organizational environment certain fundamental problems arise with our goal of describing the behavior and structure of organizations. These problems must be addressed by a description system that is used to describe organizational behavior and knowledge. We identify the following four problem areas.

### 4.2.1 Open-Ended Knowledge World

In contrast to some knowledge worlds such as the Blocks World [Winograd 71] the world of organizational knowledge is not a closed knowledge world. The complete set of actions relevant to the organizational world is unknown and unknowable. The set of all possible states are unknowable as are all possible alternatives for achieving a goal. The result is that unforeseen situations are a common occurrence. This is as much a property of the perceiver of the world as it is of the world itself since it is our assumption that the perceiver is of limited cognitive capabilities.

The open-ended character of the organizational knowledge world places demands on the kind of description system used to describe organizational knowledge. In particular the description system must be able to assimilate new information about actions, situations, and alternatives to achieving goals in an incremental fashion. The description system must be able to reason with partial information about problem solving states.

### 4.2.2 An Evolutionary Environment

Organizational environments are continuously changing. Any attempt to understand and describe organizational behavior must cope with the problem of trying to hit a moving target. This is a central problem both in talking about organizations and in doing work within organizations. A description system must be able to describe an organization that is continuously changing. A description system must also furnish tools to manage change so office workers may use it in performing their tasks.

### 4.2.3 Perception of Cognitive Processes from Overt Physical Actions

Trying to understand what task someone is doing and the reasons for each action performed in carrying out the task by watching the person perform the task is in general not possible. Information used in performing the task is not manifest in the physical actions the task entails. Even asking someone how they accomplish a particular cognitive task yields at best partial information and often apparently contradictory information. Thus the quality of information gathered by observation or interview is limited. Hence, a reliance on acquiring information on organizational behavior in this manner is not desirable.

### 4.2.4 Describing Cognitive Processes

Our goal of describing cognitive processes is not to develop a psychological theory of the individual in an organizational setting but to describe the individual in a way that--taken in aggregate--explains organizational behavior.

The premise is that there is a way to describe an *organizational person* in terms of application and organizational knowledge. In adopting this premise an assumption is made that an organization works in such a way as to factor out the individual idiosyncrasies of its members. The reason for making this assumption is that many organizations have similar behaviors but are made up of diverse personalities.

The approach adopted in this dissertation is to describe cognitive processes in terms of the application and organizational knowledge required to perform organizational work.

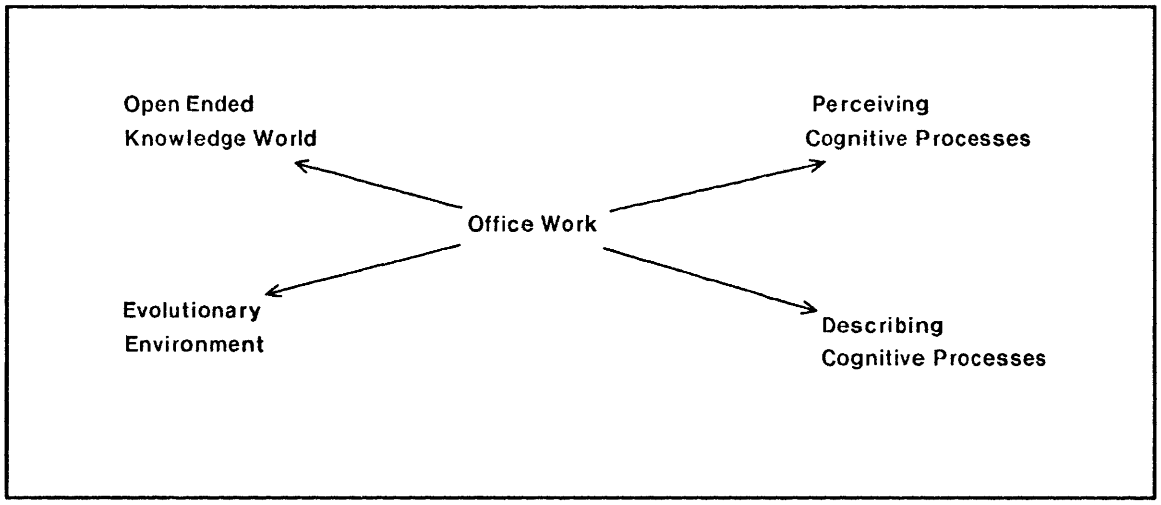


**Figure 4-1:** The Fundamental Characteristics of Office Work

These four problems central to office work, shown above in figure 4-1, are accepted as given characteristics of the organizational world. They make the task of describing organizational behavior more difficult but they also help to explain why the organizational world is structured as it is. It is beyond the scope of this dissertation to develop a complete theory of organizational behavior base on the ideas of Office Semantics and the above four characteristics of the organizational world. The belief is that they form a foundation upon which such an organizational theory can be developed.

## 4.3 The Pervasive Nature of Problem Solving

Organizations by nature are goal oriented. For example, a primary goal of organizations in the private sector is to show a profit. Other organizations such as government agencies exist to provide a service. From the standpoint that organizations have goals that they seek to establish and maintain, they are problem solving mechanisms.

The details of problem solving and how it is supported are treated more fully in sections 4.5 and 4.5.2. For the purposes of this discussion we consider problem solving to be the heuristically guided search, deduction, judgment and application of knowledge in determining what actions are appropriate in acquiring a goal.

### 4.3.1 The Need for Problem Solving

Since the function of individuals in an organization is to carry out the organization's goals the individual's work is also problem solving in nature. One may initially believe that office procedures describe an individual's task in sufficient detail that all an office worker need do is follow the steps described in the specification of the office procedure. If this were the case office work would not involve much problem solving. However consider the following example due to [Fikes 80b].

> A certain organization accepts orders for copy machine supplies and delivers and issues invoices for those supplies. When a customer calls to order supplies an order entry clerk records the customer's identity, the supplies needed, and the delivery location of the supplies. The customer is told the items are to be delivered within some period of time but the exact delivery date is not known. After the order is taken the billing calculation is made and the order is sent on to the shipping department.
>
> When the items are delivered the customer is sent an invoice for the items delivered. The amount billed is the sum of the cost of the items and a sales tax. The sales tax is the sales tax in the state in which the supplies are delivered.

What seems like a trivial office procedure can easily become problematic:

> A customer calls and orders supplies for a copier. When the customer is asked where the supplies are to be delivered he answers that the copier is on a barge and that he must know the delivery date in order to supply the delivery location. However, the order entry clerk doesn't know the delivery date. The solution the order entry clerk selects is to take the customer's phone number and attach a note to the order saying that the delivery location can be determined by calling the phone number when the delivery date is known. Note that the order entry clerk does not know that the delivery location is also needed in order to calculate the sales tax.

When the cost of the items is being prepared the clerk finds a note and a phone number instead of a delivery address. The clerk understands the problem, determines when the delivery can be made, and calls the customer to determine the delivery location. With the extra information the clerk can finish the billing calculation. The clerk sends the order on to the shipping department noting the date the item must be delivered and the reason why.

When the shipping department receives the order, special arrangements are made to deliver the supplies on the specified date. It is not standard operating procedure to make deliveries on specified dates but the shipping clerk can rearrange the delivery schedules to accommodate the situation.

There are many interesting points in this story that will be discussed in the following pages. Certainly the story is an atypical example but it serves to illustrate two points. First, even the simplest office procedures can involve a good deal of problem solving and second, this problem solving is handled with relative ease on the part of the office workers. Although the story is atypical, that is not to say that atypical situations are uncommon. Indeed, as has been pointed out in [Ellis, Nutt 80] the precise specification of even simple clerical procedures is an open problem.

In the example the need for problem solving arises from an unforeseen contingency. There are other situations where problem solving arises. A common need for problem solving activity is in diagnosing anomalous behavior. In this case problem solving is needed to both determine the causes of the anomaly and also to determine any corrective action. Problem solving can arise as a result of disturbances in the environment in which the organization operates. A strike by air traffic controllers sparks a flurry of problem solving in any organization that depends on air transport. Changes in laws, organizational polices or procedures all introduce situations where it is necessary to make a transition from an old way of doing things to a new. During a transition period, problem solving is necessary to maintain consistency of information. For example, parts of old procedures cannot be mixed with parts of new procedures without care.

### 4.3.2 The Degree of Problem Solving in Office Work

With the above view of problem solving we can characterize office work in terms of the amount of problem solving it entails as shown in figure 4-2 below.

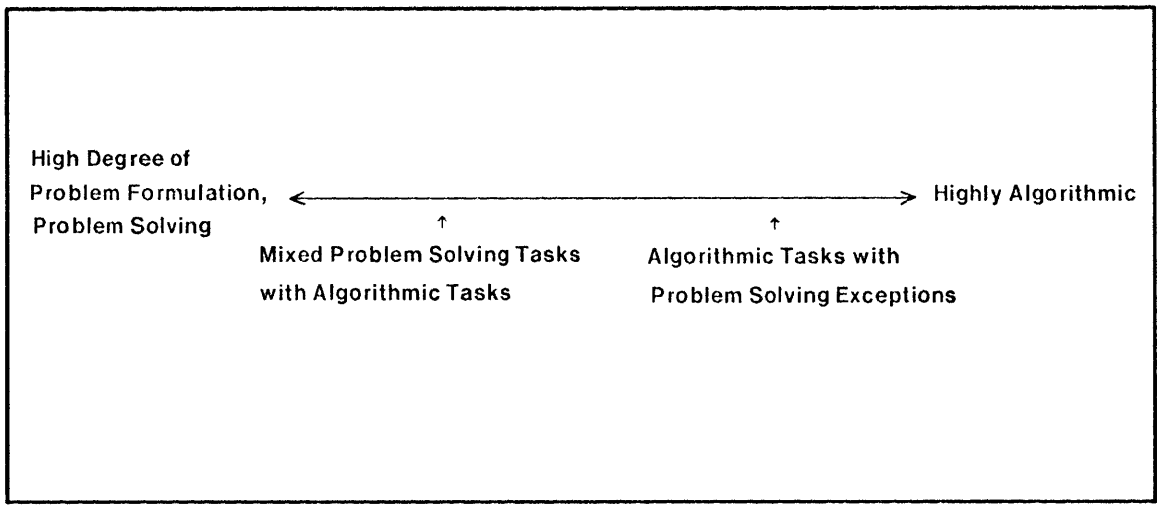

High Degree of Problem Formulation, Problem Solving
Highly Algorithmic
Mixed Problem Solving Tasks with Algorithmic Tasks
Algorithmic Tasks with Problem Solving Exceptions

**Figure 4-2:** The Degree of Problem Solving in Office Work

At one end of the spectrum is office work that involves a high degree of problem solving. This would be work in weakly structured knowledge rich environments such as policy formation, establishing long range objectives and formulating plans to achieve those objectives. At the other end of the spectrum is office work that we characterize as algorithmic. This is office work that can be described in sufficient detail so as to be realizable by machine. Examples of algorithmic office work are payroll processing procedures, certain accounting functions and certain data base queries.

We distinguish two important points on this spectrum of problem solving. The first we can characterize as algorithmic with problem solving exceptions. This is the case for algorithmic office work that occasionally involves problem solving for cases the algorithm cannot handle. An example of this is the order entry procedure described above.[5] In most cases the order entry process proceeds normally. However there are exceptional occasions--when the delivery address is unknown--when problem solving comes into play as illustrated above.

The other point we distinguish on the problem solving-algorithmicity spectrum is mixed problem solving with algorithmic tasks. An example of this type of office work is the Officer Transfer Process

[5]We are not considering the problems of understanding the customer's utterances and exacting from the customer his or her precise needs as part of the order entry process. These tasks are clearly not algorithmic. We consider the process of entering the order once the information is known as the order entry process.

presented in the introduction to this dissertation. The Assignment Officer uses his or her knowledge of the goals of the procedure and the requirements of the officers due-to-roll and the billets to make an assignment proposal. Once the officer-billet pair is chosen the task of entering the information into the proper forms and sending the proposal to the proper Placement Officer is algorithmic.

### 4.3.3 The Role of Automation

If we consider the automation of office work as the delegation of algorithmic tasks to machine we can see the role that automation plays in supporting office work. The tasks that don't involve problem solving, the algorithmic tasks, can be automated. Tasks that involve simple problem solving may be automated but more complicated problem solving cases must be done by humans. Our belief is that problem solving is one essential aspect of office work and can never be fully automated.

The goal of automating tasks in the office is to aid office workers in their problem solving functions. This is accomplished by removing the time consuming tasks that, in their performance, don't contribute to the office worker's goal of solving the problem. The results of the tasks do contribute to the problem solving process, e.g., the mechanics of finding officer's files or entering information into a form don't help the office worker, but once the files are collected or once the form is filled out the office worker can use this result in accomplishing his or her goal. Once algorithmic tasks are automated they can be powerful tools to the accomplishment of goals since their cost is drastically reduced hence they may be used more frequently. Querying data bases is a good example of this.

As illustrated by figure 4-2 above, much office work involves a mix of algorithmic and problem solving tasks. An important property automated tasks must have is that they must be well integrated with the problem solving they are supporting. For example, in tasks that are algorithmic with problem solving exceptions the automated tasks must be so designed such that when an exception arises the information necessary to accomplish problem solving is readily available. We achieve this by letting *problem solving requirements drive the automation of tasks.* Thus algorithmic tasks are automated in the context of the problem solving in which they will be used.

## 4.4 Describing Office Work

The purpose of describing office work is to make explicit the work that is done in the office. This includes the mental and physical activities that an office worker engages in and the reasons for these activities. An approach to characterizing work in the office is to consider it as organized in procedures in a fashion similar to the computer science notion of procedure. In this way office work would be described as a sequence of steps with decision points to manage flow of control.

### 4.4.1 Pitfalls of a Sequential Procedural Description Methodology

A sequential procedural characterization (e.g. flow chart, Cobol program etc.) is problematic for several reasons. Even routine tasks in offices are beset by a plethora of contingencies. In a procedural approach it is necessary to foresee the possible alternatives that may arise and write the procedure to accommodate them. When trying to describe office work in this step by step manner it becomes clear that all the alternatives cannot be enumerated. Determining what the alternatives are is part of what office work is; all alternatives cannot be determined in advance. As a result a procedural approach is not a very useful style of work description.

Office work exhibits a looseness of step ordering that is not captured by a sequential procedural description method. Steps may often be done in an arbitrary order; often steps that are order dependent may be switched if precautions are taken to accommodate for the order dependency.

Office work is of an event-driven nature. A task begins when some event occurs, be it the arrival of a form, a phone call, or the arrival of a particular day or time. This behavior can be thought of as various actions that are triggered when a particular condition arises. A procedural approach is cumbersome when trying to describe event-driven systems.

A procedural approach is too rigid a framework to accommodate change. As was mentioned above a central problem in organizations is that of dealing with change. A requirement for a description language for office work is that it can accommodate incremental change easily.

A procedural approach is useful for suggesting what the key elements of a particular office procedure are in a concise manner. However, the reason for describing office work is to elucidate, in detail, the work that it entails. Thus a procedural description of office work is not appropriate because it tends to *hide* much of the work that is actually performed in order to accomplish the goals of the office procedure. Much of the work in following a procedural description of office work is in the problem solving required in order to accomplish the steps in the description.

A result of viewing office work in a procedural manner is the *in vs out of the system* syndrome. Since, in office work, all future contingencies cannot be enumerated and since a procedural description cannot easily accommodate change, exceptions and unforeseen situations have to be handled outside of the system. The barge story in section 4.3.1 is a good example where this problem can arise. The paper based forms system had no mechanisms for handling exceptions. Thus information had to be sent along with the supply order form for the office work to be accomplished. This is misleading at best: the amount of work necessary to fill the order is not apparent from the record of the order. Accomplishment of the task depends on adhoc information workers know about their environment which may not be right or which may change.

A procedural description methodology is useful as a way of describing an ideal. It has the advantage that an overabundance of detail can be avoided. It provides, often implicitly, the goals the the office procedure is trying to achieve. However, a procedural approach is not general enough and not amenable to incremental change.

### 4.4.2 Explicit Representation of Goals and Actions

A description of office work in terms of goals and actions is a direct way of characterizing office work. A procedural description of an order entry task, for example, succinctly characterizes the important points of the task. But Precisely because of its succinctness a procedural description suffers from two defects: first, it glosses over minor details--which it assumes are easily treated--that may be problematic or critical in practice; second, the reasons for the actions specified by a procedural description must be inferred. Thus if it is impossible to fulfill a requirement in the procedural description, such as obtain the delivery address for an order, the office worker must rely on intuition and experience to select an alternative action. The more desirable approach is to state explicitly the reasons the action is needed--the goals the action achieves.

The explicit representation of goals and actions provides a recourse to handle unexpected contingencies. Office workers are able to handle unexpected contingencies in their daily work because they know the goals of the office work and because they know what actions are needed to achieve the goals of the office work. These goals and constraints are often implicit in the work and in the office workers' knowledge of their work. If a particular action cannot be performed the computer system can possibly suggest an alternative action. Failing this the office worker can use the computer system to examine the goals an alternative action must inherit from the action that cannot be performed. Together, the office worker and computer system can construct a new plan of action that maintains the necessary constraints and makes progress toward achieving the goals in question.

To support the problem solving activity in office work knowledge about the goals and constraints of the office work are explicitly represented. This builds a teleological structure of the office work. Actions that would be performed during the course of the office work are linked to the reasons they are performed and to the constraints that they are required to maintain. Explicit representation of the goals and actions exposes hidden assumptions and implicit goals about the office work. In addition, explicit representation makes the actions performed by an office worker more understandable by machine or by another individual.

Added coherence between different functional elements of a system has the benefit that the user's actions and the goals of the office procedure can be understood in terms of each other. It is useful for the system to understand the goals in order to interpret the user's requests and suggest problem solving tools for achieving the goals. In turn the user's actions suggest what the current goals are and narrows the variety of problem solving methods and size of the solution space.

## 4.5 Problem Solving Paradigms

One classical problem solving paradigm in AI is the state-operator paradigm shown below in figure 4-3. However, this paradigm is difficult to apply in the organizational world. The organizational world differs from the traditional AI worlds such as crypt arithmetic or the blocks world in that: it is distributed and parallel in that there is more than one individual working on the problem; and it is open ended in the sense that knowledge about the work is incomplete. In the following we discuss the drawbacks of the state-operator paradigm.

### 4.5.1 The State-Operator Paradigm

In the state-operator paradigm the problem solver is given a well defined initial state, for example the configuration of a chess board, a well defined final state, to win the game, and a finite collection of actions or state transformers. Consider the diagram depicting the state-operator paradigm below.

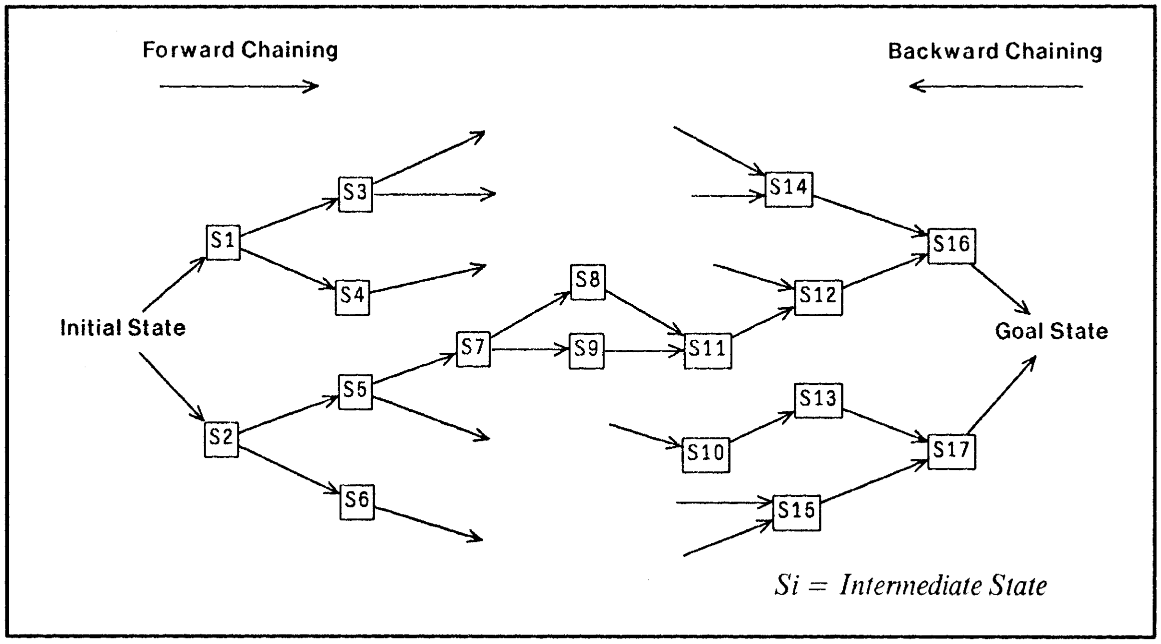


**Figure 4-3:** The Classical AI Problem Solving Paradigm

Problem solving is characterized as a search for the sequence of actions that will achieve the goal state. The test to see if the goal state has been achieved is objective and two valued, either the goal is achieved or not. (We will discuss in section 6.1 how current problem solving systems are based on the this paradigm.) The problem solver is assumed to be a single individual, thus there are no problems with synchronization or conflict with other problem solvers. When more than one problem solver is working on problem, as in an organization, a global state description of the problem and the problem solver is no longer practical.

This is a seductive paradigm but it is hard to apply in the organizational setting. The reason for this is that in using the state-operator paradigm one determines a possible means to achieve a goal by examination of the current and goal states. But in many cases in office work the goal is vague and how much information is relevant to achieving the goal is not clear; this makes an assessment of the current state difficult. This problem is suggested by the case studies in [Wynn 79] and has been pointed out by [Suchman 79, Garfinkel, Sacks 70].

Consider the order entry function described in section 4.3.1. We analyze the order entry function in the light of the state-operator paradigm. Since this paradigm is not applicable to a distributed environment only the tasks carried out by a single order entry will be considered. The clerks state when a customer calls may be described as follows: there are several partially complete orders on the clerks desk as well as a new order for the customer on the phone; the clerk has access to other state information such as inventory status, and customer account status. The clerk's goal is a completed order form; enough information must be entered into the form so that the following functions--billing and shipping--may be completed. The actions the clerk uses to achieve the goal are to take information from the customer and enter it into the order form, possibly referencing information such as inventory state.

In the description of the initial state when a customer calls a great deal of information may be irrelevant. However what is and is not relevant is not known until the goal is accomplished: the clerk may reference the partially completed orders for inventory information or if one of them is one that the customer has made that they want to change; or the clerk may reference inventory status information or customer account status information.

The actions the clerk uses may be incrementally augmented. In the story about the barge the clerk created a new action which was to attach a note to the order form describing how the supply clerk could obtain the delivery address. Note that the goal is not to fill out the ord r entry form but to obtain enough information for other function to proceed. In the barge example this was achieved by attaching the note to the order form. The goal is necessarily vague to accommodate new actions that may achieve it.

These examples suggest that much of the work in offices does not fit into the state-operator paradigm but more a style of problem solving where goals and relevant information towards achieving the goals .are developed simultaneously. In fact, as March and Simon point out [March, Simon 63], often there isn't a particular goal that is important but any goal that is in some sense *satisfactory* is acceptable. The emphasis in this case is not so much on reaching a goal via some sequence of actions but on finding a an acceptable sequence of actions to any one of a number of satisfactory goals. In traditional goal oriented problem solving achievement of a specific goal is the desired outcome of the problem solving process.

A great deal of work in the office is analyzing anomalous situations. Suchman describes an example where an accounting office received a past due invoice that the accounting office's records showed as paid. In diagnosing the situation the only goal initially is the vague "try and understand what happened." The initial state is the knowledge that something isn't right and a collection of data, such as the past due invoice and the records of the transaction, that may or may not be relevant. In this case a state-operator paradigm is of little use. After examination the data is partially assembled into some coherency and a more explicit goal suggests itself. This interplay continues until finally a course of action is determined.

In a sense the state-operator problem solving paradigm is so general that one may argue that it is all that there is. Whether this is accepted or not the state-operator paradigm is a low level problem solving paradigm. An analogy can be drawn with the Turing Machine. Although is some sense the Turing Machine is all there is to computation and in fact defines it, it is often necessary to adopt higher level computational models such as message passing models. Problem solving paradigms of a higher level of abstraction than the state-operator paradigm are needed to describe office work.

The state-operator paradigm is a problem solving technique that is most easily used in a fact centered deduction system as opposed to a object oriented system. In a fact centered systems knowledge is represented as a data base of facts or logical statements. Fact centered systems include logic based systems such as PROLOG, resolution based theorem provers, production systems such as GPS, and pattern directed invocation systems such as PLANNER and AMORD. Typically, reasoning proceeds when facts in the database trigger procedures that then assert new facts. In object oriented systems information is structured around objects. Examples of object oriented systems are FRL, KL-ONE, NETL, and KRL. These systems use if-added, if-removed, and if-needed daemons (FRL), explicit database queries (NETL), or pushers or pullers (KRL, KL-ONE) to drive reasoning processes. The state-operator paradigm is more amenable to fact centered systems since facts, initial and goal states of the problem, drive the search for sequences of actions that will transform the initial state into the goal state. Below we argue that object centered systems are a more natural choice for a system that will support office workers.

### 4.5.2 Supporting Problem Solving

In supporting office workers in their problem solving the intent is to help them perform their work as opposed to replacing them. The aim is to design a system that allows the office workers to do what they do best and allows the computer to do what it does best. The computer's purpose is to help the user, therefore the difficult problem solving tasks are done by the user and the simple problem solving and algorithmic tasks can be done by the computer.

An important property of problem solvers in an organization is that of *delegated authority*. There are cases when a problem solver will make a choice not because he/she/it has made a reasoned decision but because the choice was mandated by a supervisor. Thus an order entry clerk may not understand the status of a particular customer account and ask a supervisor. The clerks supervisor will tell the clerk what to do. The clerk's supervisor will either explain to the clerk the reason for the decision or will ask the clerk to accept the decision on faith. In the first case the clerk will know what to do if the situation arises again.

A similar relationship will exist between the office worker and the computer supporting problem solving. A computer may delegate decision making to its user if either, the computer doesn't recognize the situation as one it can handle or the situation is one that requires authorization from the office worker.

In the classical problem solving paradigm the problem solver does all the work itself. It does not include the idea of more than one entity working on a problem. Our approach is to support problem solving. As a result we propose the *Problem Solving Support Paradigm* shown below.

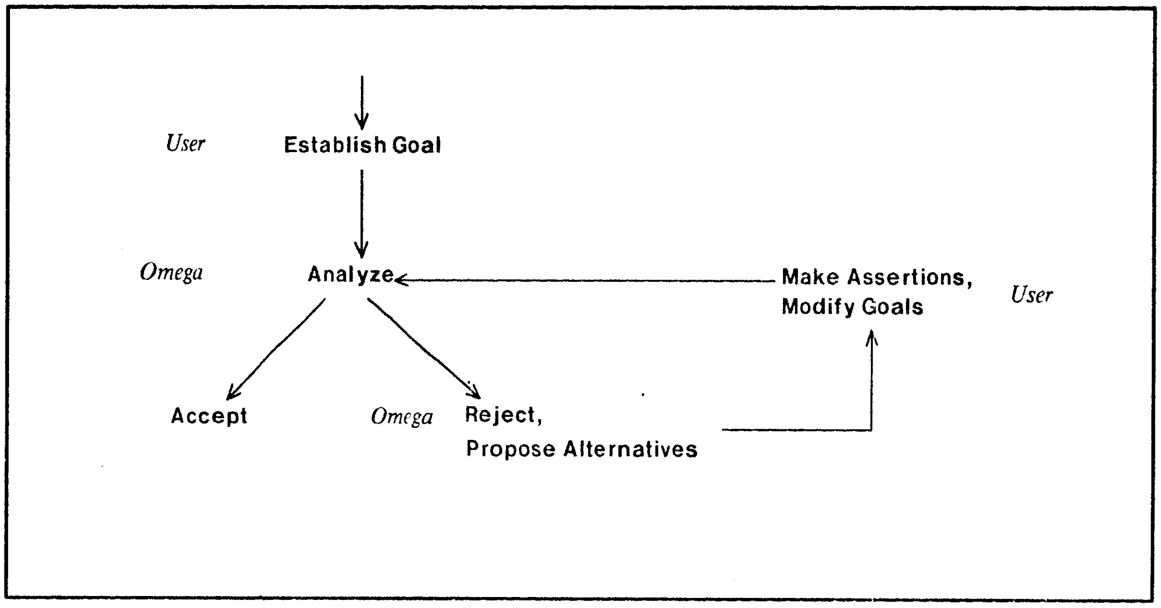


Figure 4-4: The Problem Solving Support Paradigm

In the problem solving support paradigm the office worker establishes a goal, for example to send a message or to complete a step in an office procedure. Based on what Omega knows about the goal it either tries to establish or refute the goal. If the goal can't be attained Omega complains to the office worker that the goal cannot be established or that contradictory information has been discovered during the attempt to establish the goal. At this point the office worker can either modify the goal or make further assertions possibly supplying necessary information to establish the goal. Omega then attempts to establish the goal again. This cycle continues until the goal is established. The analysis is accomplished using Omega's viewpoint mechanism.

## 4.6 Acquiring Knowledge to Support Problem Solving

Knowledge about the goals and actions of office work must be gathered in order support problem solving. March and Simon have suggested three approaches for collecting information on office work [March, Simon 63].

1. Observing the behavior of organization members.
2. Interviewing member of the organization.
3. Examining documents that describe standard operating procedures.

Each of these possibilities presents problems. In the first case it is difficult to determine what an individual is doing by observing his or her actions without understanding the reasons behind the actions. Since much office activity involves knowledge processing one cannot understand an individual's actions successfully by just observing it.

In the second case members of organizations themselves have difficulty articulating what they do. It is very difficult to explain to an interviewer the minute details of work that a worker has learned over many years of experience. This is partly because much of the task has been internalized and happens subconsciously. To ask workers to explain the reasons behind their every move is asking them to know what they do subconsciously. In addition, individuals differ in their ability to articulate what they do and observe what they do. There is no way to determine how their descriptions differ from reality. Two people asked to describe the same work will produce two differing accounts of the work. There is no objective means to determine which of the two accounts or which parts of the two accounts conform to reality.

We are ignoring further problems that arise from the fact that the work can be described as it is actually done or as it should be done. The workers may well give differing accounts because they may do the work differently. The workers could be asked to give their conceptual model of how the work should be done. But the "what should be done" vs "what is done" distinction may be misleading because after many years at a task what is actually done may well become the conceptual model of how the work should be done.

What the worker possesses is an ability to plan and execute, in the face of unexpected contingencies, actions that achieve the goal of the office work. The results of interviewing office workers are the outcome of this planning process. What is really desired is the knowledge that drives the planning process and knowledge about how the planning process works. Consider interviewing the order entry clerk described in 4.3.1. The interviewer would ask the clerk to describe the common case and then the exceptional cases such as what happens when the inventory for the desired item is low, what happens when the customer's credit is in question, or what happens when a customer wants an unusual item the is not in the inventory. But there are an infinite number of exceptions; it is doubtful that even the most farsighted interviewer would ask what would happen if the delivery location was a barge.

The third case involves examining documentation of the standard operating procedures. It is well known that this documentation is rarely accurate and seldom used. In most offices procedures manuals are paper documents hence difficulty and expensive to update. Examination of these kinds of documents can only give the barest of ideas of what the programs in an organization are.

A drawback of all the approaches mentioned above is that the interviewer is human, thus the interviewer asks questions based on what he or she understands. It is then difficult to transfer the knowledge to a machine because many assumptions that a human makes about the work are not brought to light.

A more direct approach to acquiring office knowledge is to let the office workers, in conjunction with their workstation, collect the knowledge needed to perform their roles in the office. In this way the machine can ask questions about the work and in particular point out assumptions that the office workers make. Although this subject is beyond the scope of this dissertation the use of systems such as TINKER [Lieberman, Hewitt 80] look very promising in this respect.

# Chapter Five

# OMEGA

Omega is used to build, maintain and reason over a lattice of descriptions. Descriptions are related via an inheritance relation called the `is` relation. The lattice has a top-most or most general element called ***Something*** and a bottom-most element called ***Nothing***. The element ***Something*** can be interpreted as representing the state of no information while ***Nothing*** can be interpreted as over-information or contradictory information.

This chapter provides an information introduction to the Omega language. The basic types of descriptions and statements are presented with axioms and examples. Viewpoints are not discussed, they are presented in latter chapters. For a more thorough treatment the reader is referred to [Hewitt, Attardi, Simi 80]. A semantics and consistency proof for a subset of Omega is presented in [Attardi, Simi 81].

Listed below are several of Omega's important properties. They have been derived from the desire to use Omega in a knowledge rich environment (as described in section 4.2). These properties influence the way Omega is implemented and the way it is used. They are themes that continuously reappear in this dissertation.

- **Monotonicity of Information** - Information is only added to the system, never lost. As information becomes older it may take longer to access because older, unused information is migrated to mass storage archival devices that have slower access times. Information is never changed within Omega. Updates are replaced by a mechanism which marks the updated information as out of date and uses the new updating information.

- **Commutativity of Assertions** - Omega is designed to accommodate the accumulation of information. No fundamental differences occur if Omega is told the same things in a different order. In practice, response times may vary but Omega is not sensitive to the order in which it is given information. Mechanism exist within Omega to maintain the consistency of descriptions when new descriptions are added. One of these is the deduction mechanism described below.

- **Making Deductions** - Omega has basic inference rules called *merging* and *fusing*. When an object which is described in the system is further described, the new description is combined with old descriptions resulting in a more specific description about the object.

- **Partial Descriptions** - Partial descriptions are an important tool to the knowledge worker. Partial descriptions are frequently used and are used to refer to a particular object or may be accumulated and merged until the object they refer to can be deduced. As an example a description of the form "the invoice for 100 widgets" is a partial description because it does not describe an invoice fully but is useful in identifying it.

- **Incrementality** - Information can be added to Omega in an incremental fashion. In this way unforeseen situations and new information can be added to Omega. The knowledge necessary to function in a particular role can be accumulated over time, minimizing the amount of start up knowledge necessary of effective use of Omega.

- **Finding the Extension of a Description** - A common operation is to retrieve information based on a description, e.g., retrieve all invoices whose total exceeds $10,000, or retrieve all items that can fill a certain field of a form. This latter retrieval is possible because each field contains a description of what can fill it.

- **Managing Change with Viewpoints** - Change is managed with a viewpoint mechanism. The viewpoint mechanism insures that change proceeds in an orderly fashion, avoiding race conditions that can lead to circumstantial contradictions.

## 5.1 The Axiomatization of Omega

The behavior of Omega is described via a set of *description axioms*. The axiomatization of Omega is a powerful tool by which to communicate the behavior of the description system. The axioms function as a contract between the user and the implementor of the system as well as a base from which to derive theorems about the system's behavior. This contrasts with other methods for describing the behavior of knowledge embedding languages such as informal English descriptions or large program texts.

It should be noted that this use of axioms is only distantly related to the attempts to axiomatize systems in the program verification area. In program verification an attempt is made to describe a program's behavior axiomatically and then to verify that the program indeed behaves as dictated by the axioms. Omega's approach is to use axioms as a vehicle to study the structure of description systems in a manner similar to that which set theorists have used axioms to study the structure of different set theories.

The axioms are used to guide the implementation of Omega. Thus the soundness of Omega as a logic can be investigated independent of the implementation. For example, a model theory of a set of axioms similar to those presented here has been developed [Attardi, Simi 81]. To be able to have assurance that the conclusions a system makes are sound is of utmost importance. Some knowledge systems are not based on a firm mathematical foundation and thus their conclusions are always suspect. An example is Fahlman's NETL system [Fahlman 77] and its "copy confusion" problem. This particular problem has been fixed by Fahlman but since there is no precise semantics for the system and no consistency proof there is no assurance that the copy confusion problem or some other problem may not reappear in a more subtle form.

Complicated knowledge processing problems can often be difficult to understand. In some knowledge embedding systems that don't have a precise semantics it may be difficult to decide whether suspicious behavior in the implementation is anomalous or correct. Omega's axiomatization provides an objective way to verify the implementation's behavior.

## 5.2 Descriptions

The simplest description in Omega is the *atomic* description; the following are examples of atomic descriptions:

**proposal-17**
**invoice**

Atomic descriptions are called as they are because they have no structure. A non-atomic description is the *Instance Description.* A simple example of an instance description is of the form:

**(*an* Office-Procedure))**

This description is used for describing a typical member of a class of objects, in this case the class of office procedures. Note that we use the convention that Omega's keywords are written in a boldface italic font, such as the ***an*** above. The only *part* the instance description above has is its *concept* which in this case is **Office-Procedure.** An Instance description with more structure is:

```
(a Supply-Order (with Customer customer-17)
                (with Item widget-5)
                (with Item widget-7))
```

This is an instance description for the concept **Supply-Order** and with three *attributions.* Attributions further restrict the class of objects an instance description describes. In some cases the information in attributions will restrict the class of objects to a single object or to the null class. In this latter case the instance description will be identified with ***Nothing***. The attributions of instance descriptions are similar to the attribute-value pairs of frames as in FRL [Goldstein, Roberts 77] and the indicator-value pairs of Lisp's property lists. However Omega's attributions are more highly developed, for example in the case above attribute names and values are descriptions and there can be multiple attributions with the same attribute name. There are other differences which will be described more fully below. The instance description above describes a typical member of the **Supply-Order** class with **Customer** attribute **customer-17** and two **Item** attributes with descriptions **widget-5**, and **widget-7**.

The simplest sentence in Omega is the *predication* and is of the form:

**(customer-17 *is* (*a* Customer (*with* Address address-5)))**

This statement says that **customer-17** describes a typical member of a class of objects--no commitment is made as to the number of members of the class--that are also members of the **Customer** class with the given attribute. The ***is*** relation is the fundamental relation in Omega and is referred to as the *inheritance* relation. The ***is*** statement is used to construct Omega's inheritance lattice. Sometimes the statements in Omega will be written as infix expressions as above and sometimes as prefix expressions depending on which is more readable. The bold italic font distinguishes Omega's operators and disambiguates the statements. One of the basic properties of the ***is*** relation is transitivity:

**Axiom 5-1:** (Transitivity of ***is***)

If $(d_1$ *is* $d_2)$ and $(d_2$ *is* $d_3)$ then $(d_1$ *is* $d_3)$.

The ***is*** relation is reminiscent of the *is-a* relation of systems such as FRL. They are different for in the systems that use the is-a relation it is possible to say

**John is-a Human** *and* **Human is-a Species**

and if transitivity is allowed one can erroneously conclude that

**John is-a Species**

In Omega this example appears as:

**(John *is* (*a* Man))** *and* **(Man *is* (a Species))**

and in this case, since **Man** and **(a Man)** are different descriptions transitivity cannot be applied to obtain the erroneous conclusion. What can be concluded is the following:

**John** ***is*** **(*a* (*a* Species))**

Which says that **John** is a typical member of a class of objects which are also members of a second class of objects. What is known about the second class of objects (note, not the members of the class!) is that it is a member of the **Species** class of objects. Note in the above example we have used the concept of the instance description to talk about the class of objects as a whole that the instance description refers to. This will be used later when we describe viewpoints.

The ***is*** relation also is reflexive:

**Axiom 5-2:** (Reflexivity of ***is***)

For every description **d**, **(d *is* d)**.

The ***same*** relation is defined in terms of the ***is*** relation.

**Axiom 5-3:** (***same*** Relation)

For all descriptions $\mathbf{d}_1$, $\mathbf{d}_2$,
$\mathbf{d}_1$ ***same*** $\mathbf{d}_2$ if and only if $\mathbf{d}_1$ ***is*** $\mathbf{d}_2$ and $\mathbf{d}_2$ ***is*** $\mathbf{d}_1$

## 5.2.1 Description Lattice Operations

Other operators are available for creating new description from existing descriptions. These are the familiar lattice operations, meet, join, and negation which are called ***dand***, ***dor***, and ***dnot***. The operators are prefaced with a ***d*** to indicate that they are description operators. Some examples of the use of these descriptions are shown below

**(*dand* (*a* Supply-Order (*with unique* Customer Floating-Copiers-Inc))**
**(*a* Task-Input (*with* Task (*a* Billing-Task))))**

This description describes those objects that are both **Supply-Order**'s for a particular customer and inputs for the **Billing-Task**. The ***dor*** description joins the objects referred to by descriptions. For example the following states that those individuals that perform a billing calculation are either **Billing-Clerks** or **Billing-Clerk-Supervisors.**

**(*is* (*a* Task-Performer (*with* Task (*a* Billing Calculation)))**
**(*dor* (*a* Billing-Clerk)**
**(*a* Billing-Clerk-Supervisor)))**

The negation operation is used to describe the complement of classes of objects. The following example states that a **Billing-Clerk** is not an **Order-Entry-Clerk**.

**(*is* (*a* Billing-Clerk) (*dnot* (*an* Order-Entry-Clerk)))**

Descriptions obey a DeMorgan's Law and a law of double negation. The axiom for the DeMorgan's law is:

**Axiom 5-4:** (DeMorgan's Law)

**(*same* (*dnot* (*dand* $d_1$ $d_2$))**
**(*dor* (*dnot* $d_1$) (*dnot* $d_2$)))**

## 5.2.2 Attributions

There are 4 types of attributions that can be used with instance descriptions in Omega, these are ***of***, ***with***, ***with every***, and ***with unique***. The attributions differ in how they can be manipulated. First, the properties that hold for all attributions are presented. The axiom of commutativity of attributions states that the order of the attributions of an instance description does not matter.

**Axiom 5-5:** (Commutativity of Attributions)

For every description C and attributions $\alpha_i$
**(*a* C $\alpha_1 \ldots \alpha_{i-1}\ \alpha_i \ldots \alpha_n$) *same* (*a* C $\alpha_1 \ldots \alpha_i\ \alpha_{i-1} \ldots \alpha_n$)**

For example:

**(*a* Supply-Order (*with unique* Customer Floating-Copiers-Inc)**
**(*with unique* Delivery-Date (*a* Date)))**
***same***
**(*a* Supply-Order (*with unique* Delivery-Date (*a* Date))**
**(*with unique* Customer Floating-Copiers-Inc))**

The omission axiom can be used to produce more general instance descriptions:

**Axiom 5-6:** (Omission Axiom)

For every description C and attributions $\alpha_i$
**(*a* C $\alpha_1 \ldots \alpha_i \ldots \alpha_n$) *is* (*a* C $\alpha_1 \ldots \alpha_{i-1}\alpha_{i+1} \ldots \alpha_n$)**

For any instance description the deletion of an attribution produces a more general instance description. For example:

**(*a* Supply-Order (*with unique* Customer Floating-Copiers-Inc)**
**(*with unique* Delivery-Date (*a* Date)))**
***is***
**(*a* Supply-Order (*with unique* Customer Floating-Copiers-Inc))**

All attributions also have a monotinicity property:

Axiom 5-7: (Monotinicity of Attributions)

If $\tau$ is one of the attribute types ***of***, ***with***, ***with unique***, or ***with every***,
for descriptions C, $\nu$ and **(d$_1$ *is* d$_2$)**
then **(*a* C ($\tau$ $\nu$ d$_1$)) *is* (*a* C ($\tau$ $\nu$ d$_2$))**

For example, suppose that the following is known to hold:

**(*a* Billing-Calculation) *is* (*a* Task)**

Then the following inheritance relation can be deduced:

**(*a* Task-Performer (*with* Task (*a* Billing-Calculation)))**
***is***
**(*a* Task-Performer (*with* Task (*a* Task)))**

Monotinicity is an important property that holds for other types of descriptions as well. Other monotinicity axioms will be discussed in section 5.4. The ***with*** attribution is mergible in the following sense:

Axiom 5-8: (Merging of ***with***)

Let $\alpha_1$, and $\alpha_2$ be ***with*** attributions, then
**(*dand* (*a* C $\alpha_1$) (*a* C $\alpha_2$)) *same* (*a* C $\alpha_1$ $\alpha_2$)**

Using this axiom incremental information is merged together to produce new descriptions. For example, if it is know that

**form-1 *is* (*dand* (*a* Supply-Order (*with* Item widget-5))**
**(*a* Supply-Order (*with* Item widget-7)))**

it can be concluded that

**form-1 *is* (*a* Supply-Order (*with* Item widget-5)**
**(*with* Item widget-7))**

In some cases merging of attributions in the manner shown above in combination with the axiom of omission is not desirable. This is the reason for the ***of*** attribution. Consider the following description for the sum of two numbers:

**6 *same* (*a* Sum (*of* arg1 2)(*of* arg2 4))**

By the axiom of omission and since **6** is also a sum with an argument of **1** one can conclude that

**6 *is* (*dand* (*a* Sum (*of* arg1 2)) (*a* Sum (*of* arg2 1)))**

If merging were allowed for *of* attributions then a conclusion that 6 was the sum of 2 and 1 could erroneously be produced. Thus the difference between the *of* attributions and the ***with*** attributions is that ***with*** attributions are mergible. The *of* attribution is used where the attributions have some relation to each other. In the example above, 2 and 4 are related to 6. The ***with*** attribution is used for attributes that have no relationship with each other. The ***with every*** attribution is used to specify that all attributes with a certain name have a particular property.

Axiom 5-9: (Fusion of ***with every*** Attribution)

**(*is* (*a* C (*with* $v$ (*dand* $d_1$ $d_2$)))**
**(*a* C (*with every* $v$ $d_1$) (*with* $v$ $d_2$)))**

If it is necessary to specify that all **Items** of a **Suppiy-Order** are constrained to be of a certain class of objects this is expressed by the following statement:

**(*a* Supply-Order) *same* (*a* Supply-Order (*with every* Item (*a* Supply-item)))**

In some cases an attribution of an instance description may be unique. This is the function of the ***with unique*** attribution.

Axiom 5-10: (Fusion of ***with unique*** Attributions)

**(*is* (*a* C (*with unique* $v$ (*dand* $d_1$ $d_2$)))**
**(*a* C (*with* $v$ $d_1$) (*with unique* $v$ $d_2$)))**

This axiom allows one to conclude that different attributes with the same attribute name are mergible. For example suppose it is known that

```
(form-1 is (dand
            (a Supply-Order (with Customer (a Local-Firm)))
            (a Supply-Order (with unique Customer (a Customer)))))
```

It is possible to conclude that

```
(form1 is (a Supply-Order (with unique Customer (dand
                                                  (a Local-Firm)
                                                  (a Customer)))))
```

In summary, attributions are a way of specifying relationships between descriptions. The triary relation Sum can be expressed as:

**(*a* Sum (*of* Arg1 5) (*of* Arg2 3))**

The first instance description expresses a triary relation between the object the description describes--the sum, **8** in this case--and the two arguments. Note that in keeping with Omega's goal of supporting incremental accumulation of information relations do not have to predefined as to the number of arguments they have. An n-ary relation between a **Supply-Order** and its items may be expressed as:

> **(*a* Supply-Order (*with* Item widget-3) (*with* Item widget-9))**

This says that the **Supply-Order** has two items associated with it. In the example above there is no limit to the number of **Items** to the relation. New items can be merged in with the **Supply-Order** description as needed.

### 5.2.3 Variables

There are two variable descriptions that are used during pattern matching and in statements. These are:

| | |
|---|---|
| *Variable Description, written as:* | **≡Var** |
| *Qualified Variable Description, written as:* | **(*which is* ≡Var d)** |

These descriptions are used as the bound variables in universally and existentially quantified statements. The use of these descriptions in pattern matching will be considered in more depth in section 7.4.1 on pattern matching. Variable descriptions are used during the pattern matching process to bind descriptions to the variables. The qualified variable description, ***which is***, restricts its variable to be bound to descriptions that inherit from the qualifying description, the description **d** in the example above. The qualified variable is used in pattern matching.

## 5.3 Statements

In addition to the descriptions described above Omega provides statements as a way to modify and talk about the description lattice. The descriptions presented above cannot talk about the structure of the lattice. Statements are used to represent relationships or constraints that cannot otherwise be represented in the lattice. Statements are descriptions and thus are subject to all the axioms that apply to descriptions. In addition, universal or existentially quantified statements may range over any type of description, statements included.

## 5.3.1 Statement Types

Two statements have already been presented in the earlier sections of this chapter, these were the ***is*** and ***same*** statements. Omega includes all the standard logical operators, these are shown below.

| | |
|---|---|
| ***land*** | *Logical conjunction* |
| ***lor*** | *Logical disjunction* |
| ***lnot*** | *Logical negation* |
| ***limplies*** | *Implication* |
| ***lequiv*** | *Equivalence* |

As with descriptions, the first letter of these operators is ***l*** to distinguish them as logical operators. Often the the word form of the logical operator will be replaced by its logical symbol, e.g. $\wedge$ for ***land*** etc. In addition to the logical operators above a few more are defined for convenience. It is often necessary to be able to state that a description refers to a single entity. This is done with the ***individual*** predicate which is defined as follows:

**Definition 5-1:** (Individual Predicate)

```
individual(d) ⇔
(∧ ¬ (d same Nothing)
    ( x is d ⇒ (∨ (x same d)
                  (x same Nothing))))
```

The interpretation of the ***individual*** predicate is that it is true of a description **d** if and only if **d** is not ***same*** with ***Nothing*** and any description that inherits from **d** is ***same*** with **d**. In other words, **d** is the most specific description in the lattice that isn't ***Nothing***. The motivation for this definition is that if **d** is to refer to a class of objects with a single member then any class of objects that inherits from **d** is either ***same*** with the class, i.e. it refers to the same object or it is the null class of objects which is ***Nothing***.

Another logical predicate that will be used is the ***exclusive*** predicate. The ***exclusive*** predicate is used to express that two descriptions describe mutually exclusive classes of objects.

**Definition 5-2:** (Exclusive Predicate)

```
exclusive(x y) ⇔ (∧ ¬ (x same Nothing)
                    ¬ (y same Nothing)
                    (x is (dnot y))
                    (y is (dnot x)))
```

In addition to stating that two descriptions are mutually exclusive the ***exclusive*** predicate expresses the fact that the classes of objects referred to by the descriptions are non-null. The reason for this is to make answering questions easier. Suppose **(*a* man)** and **(*a* woman)** are stated as being exclusive and then we asked whether joe, a man, inherited from **(*a* woman)**. If **(*a* man)** was same with ***Nothing***, the joe would be to and the answer to the question would be yes. We avoid this situation by stating that the arguments of an exclusion are not ***same*** with ***Nothing***.

### 5.3.2 Quantification

Omega allows both universal and existential quantification, the first explicitly and the second implicitly. The ***for all*** construct is used to universally quantify a statement. For example, the following is used to say that all customer accounts that are overdue are accounts under review.

```
(for all ≡acct
     (⇒ (land (≡acct is (a Customer-Account
                                (with unique Status Overdue)))
               (lnot (is ≡acct (a Special-Account))))
          (≡acct is (an Account-Under-Review))))
```

Existential quantification is handled via the description mechanism. The following expresses the fact that a customer account, for example **account-17**, has an account balance:

```
(account-17 is (a Customer-Account
                    (with unique Account-Balance Something)))
```

If more information is known about the customer account the ***Something*** may be replaced with another description. In this case it could be replaced with **(*an* Dollar-Amount)** since that much is true about all account balances.

## 5.4 Monotinicity Rules for Descriptions

The monotinicity axioms define the specificity or information content of descriptions. They are used to relate descriptions of the same type in the description lattice. The monotinicity properties of descriptions are central to Omega's semantic sprite invocation mechanism discussed in section 7.4. In axiom 5-7 the monotinicity of attributions were defined, a monotinicity rule also holds for for concepts in instance descriptions:

Axiom 5-11: (Monotinicity of Instance Description Concepts)

If $(C_1\ \mathit{is}\ C_2)$ and $\alpha_i$ attributions, then
$(a\ C_1\ \alpha_1 \ldots \alpha_n)\ \mathit{is}\ (a\ C_2\ \alpha_1 \ldots \alpha_n)$

Thus if it is known that

**Supply-Order *is* (*a* Type-of-Order)**

Then it can be concluded that

**(*a* Supply-Order) *is* (*a* (*a* Type-of-Order))**

Below is the monotinicity axiom for the ***dand*** description.

Axiom 5-12: (Monotinicity of ***dand***)

For descriptions $d_i$, and d, if d *is* $d_j$ then
**(*is* (*dand* $d_1 \ldots d_{j-1}$ d $d_{j+1} \ldots d_n$)**
**(*dand* $d_1 \ldots d_{j-1}$ $d_j$ $d_{j+1} \ldots d_n$))**

A similar axiom exists for the ***dor*** description. The monotinicity axiom for the ***dnot*** description is defined in a similar way taking into considerations the properties of negation. Note that the sense of inheritance for the argument to the ***dnot*** description is reversed.

Axiom 5-13: (Monotinicity of ***dnot***)

If $d_1$ *is* $d_2$ then
**(*dnot* $d_2$) *is* (*dnot* $d_1$)**

The most specific ***dnot*** description is **(*dnot Something*)** which is ***same*** with ***Nothing*** and the most general ***dnot*** description is **(*dnot Nothing*)** with is ***same*** with ***Something***. For example, if we have

**(*a* Billing-Clerk-Supervisor) *is* (*a* Billing-Clerk)**

then we can conclude

**(*dnot* (*a* Billing-Clerk)) *is* (*dnot* (*a* Billing-Clerk-Supervisor)).**

Next we consider when predications, or ***is*** statements, are related by the ***is*** relation or in other words, when one predication is a more specific case of another predication. The monotonicity axiom for the ***is*** statement is more complicated.

Axiom 5-14: (Monotinicity of ***is***)

Given the *is* statements $d_1$ *is* $d_2$ and $d_3$ *is* $d_4$
then **($d_1$ *is* $d_2$) *is* ($d_3$ *is* $d_4$)**
if both **($d_1$ *is* $d_3$)** and **($d_4$ *is* $d_2$)**

Note that the sense of inheritance between the first and second arguments of the ***is*** statements is reversed.

*Most General is Statement* **(*Something is Nothing*)**
*Most Specific is Statement* **(*Nothing is Something*)**

Note that there is no connection between inheritance and the truth of statements. So although the most general ***is*** statement inherits to all ***is*** statements, it does not inherit its truth value. To understand the inheritance of ***is*** statements more fully we consider the following example. Suppose we have the two inheritance relations:

**(*a* Billing-Clerk-Supervisor) *is* (*a* Billing-Clerk)** **(5-1)**
**John *is* (*a* Billing-Dept-Employee)** **(5-2)**

The monotinicity axiom states that predication 5-2 inherits from (or is more specific than) predication 5-1 if the following is true:

**John *is* (*a* Billing-Clerk-Supervisor)** **(5-3)**
**(*a* Billing-Clerk) *is* (*a* Billing-Dept-Employee)** **(5-4)**

Note that intuitively predication 5-2 is more specific that predication 5-1 because is states that a smaller class of objects(**John**) is a member of a larger class of objects **((*a* Billing-Dept-Employee))** which is a stronger statement than predication 5-1 in the following sense. The ***is*** statement is more specific or stronger in that is discriminates a smaller class of things from a larger class of things. Another possible definition for monotinicity of ***is*** statements is the following.

Given the ***is*** statements A ***is*** B and **C *is* D** then **(C *is* D) *is* (A *is* B)**
if **C *is*** A **and D *is* B**

But this leads to the anomalous conclusion

**(A *is* A) *is* (A *is* B)**

or that **(A *is* A)** is more specific than **(A *is* B)**. This same anomaly arises in the other two possible definitions for monotinicity of predications.

The monotinicity axioms for the other statements are not as tricky as that for the predication. Those for the ***land*** and ***lor*** statements are similar to those for the ***dand*** and ***dor*** descriptions. Consider the monotinicity axioms for the ***same*** and ***lnot*** statements.

Axiom 5-15: (Monotinicity of ***same***)

If **($d_3$ *is* $d_1$)** and **($d_4$ *is* $d_2$)** then
**(*same* $d_3$ $d_4$) *is* (*same* $d_1$ $d_2$)**.

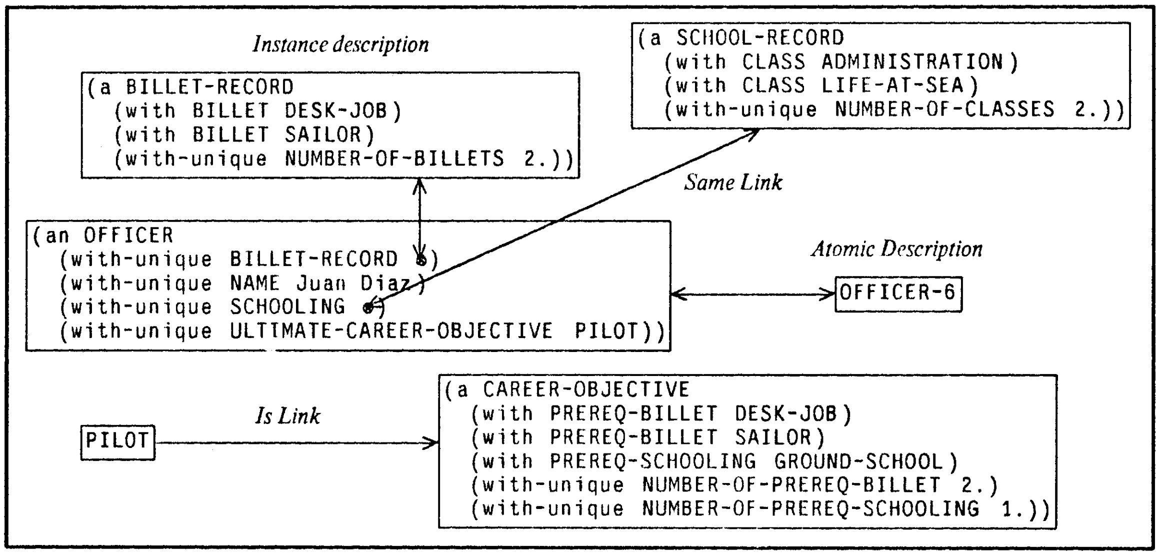
Axiom 5-16: (Monotinicity of ***lnot***)

If **($d_1$ *is* $d_2$)** then **(*lnot* $d_1$) *is* (*lnot* $d_2$)**

Note that logical negation does not invert the sense of inheritance for its argument as does description negation. Monotinicity axioms for the other statements in Omega, ***limplies***, ***individual***, ***lequiv*** etc., follow in a similar manner.

## 5.5 The Omega Presenter

The Omega Presenter is a graphical interface to the Omega system developed by Gene Ciccarelli. It allows a user to view portions of an Omega description lattice graphically. In addition the the Presenter allows the user to make assertions and thus build the description lattice further. In the remainder of this dissertation we will often use Presenter diagrams to illustrate the relationships between descriptions. Show below is an example of the Presenter's graphical output.



Instance description
(a BILLET-RECORD (with BILLET DESK-JOB) (with BILLET SAILOR) (with-unique NUMBER-OF-BILLETS 2.))
(a SCHOOL-RECORD (with CLASS ADMINISTRATION) (with CLASS LIFE-AT-SEA) (with-unique NUMBER-OF-CLASSES 2.))
Same Link
(an OFFICER (with-unique BILLET-RECORD ) (with-unique NAME Juan Diaz) (with-unique SCHOOLING ) (with-unique ULTIMATE-CAREER-OBJECTIVE PILOT))
Atomic Description
OFFICER-6
PILOT
Is Link
(a CAREER-OBJECTIVE (with PREREQ-BILLET DESK-JOB) (with PREREQ-BILLET SAILOR) (with PREREQ-SCHOOLING GROUND-SCHOOL) (with-unique NUMBER-OF-PREREQ-BILLET 2.) (with-unique NUMBER-OF-PREREQ-SCHOOLING 1.))

**Figure 5-1:** The Omega Presenter

The boxes represent descriptions, thus **OFFICER-6** and **PILOT** are atomic descriptions while the other descriptions are instance descriptions. The single headed arrow indicates and ***is*** relation thus the bottom two descriptions and their connecting arrow represent the following:

```
PILOT is (A CAREER-OBJECTIVE
             (WITH-UNIQUE NUMBER-OF-PREREQ-SCHOOLING 1)
             (WITH PREREQ-SCHOOLING GROUND-SCHOOL)
             (WITH-UNIQUE NUMBER-OF-PREREQ-BILLET 2)
             (WITH PREREQ-BILLET DESK-JOB)
             (WITH PREREQ-BILLET SAILOR))
```

The double headed arrow represents the *same* relation between two descriptions. Thus the top most configuration of descriptions represents the following:

```
OFFICER-6
same
(AN OFFICER
   (WITH-UNIQUE NAME "Juan Diaz")
   (WITH-UNIQUE ULTIMATE-CAREER-OBJECTIVE PILOT)
   (WITH-UNIQUE BILLET-RECORD
                (A BILLET-RECORD
                  (WITH BILLET DESK-JOB)
                  (WITH BILLET SAILOR)
                  (WITH-UNIQUE NUMBER-OF-BILLETS 2)))
   (WITH-UNIQUE SCHOOLING
                (A SCHOOL-RECORD
                  (WITH-UNIQUE NUMBER-OF-CLASSES 2)
                  (WITH CLASS LIFE-AT-SEA)
                  (WITH CLASS ADMINISTRATION))))
```

Note how the diagrammatic representation of the descriptions and their relations is more concise and easier to understand. Descriptions that are not atomic and appear as a part of a description are presented outside the descriptions that contain them and are linked with a ***same*** arrow. This is the case with the **BILLET-RECORD** and **SCHOOL-RECORD** instance descriptions shown above. Whether these embedded descriptions are presented or not is up to the user, thus allowing control of the amount of detail in the diagram.

# Chapter Six

# Viewpoints

## 6.1 Motivation

Viewpoints may be thought of as repositories for descriptions and thus statements. Viewpoints are reminiscent of McCarthy's situational calculus [McCarthy, Hayes 69] and the contexts of QA4 [Rulifson, Derksen, Waldinger 72] and Conniver [Sussman 72]. The most notable difference between viewpoints and these systems is that viewpoints are objects within the system, they may be reasoned about and described just as any other description in the system.

### 6.1.1 Viewpoints Compared to Contexts

In comparison with contexts viewpoints are a more flexible approach to organizing alternative descriptions of a situation. Contexts are arranged in a tree-like fashion; to move from one context to the next involves pushing the current context--moving from root to leaves of the tree--and popping the current context--moving from leaves to the root. Switching contexts is expensive because the environment must be set up when switching to a new context. With viewpoints there is no current viewpoint, they all exist in the environment at the same time. Thus there is no current viewpoint and no expense in switching reasoning processes from one viewpoint to the next.

In addition, viewpoints can be organized into arbitrarily complex graphical structures, they are not limited to trees. The relations between viewpoints are explicitly described, they are not limited to being ancestor or descendant relations of a tree. An important aspect of the viewpoint mechanism that has no analogue in contexts is the ability to stand outside of a viewpoint and reason about its contents. This is important in reasoning about viewpoints that have derived contradictions.

A key property of viewpoints that is different from contexts is that information is only added to them and is never changed. Consider, for example, a description of an invoice. The description is in a viewpoint and may be further described in the viewpoint increasing its specificity. There may be rules that maintain constraints between attributes of instance descriptions. thus as information is added to a viewpoint further information may be deduced. For example, a rule for invoice descriptions may state that the subtotal plus a sales tax must equal the total; thus when any two of the attributes is know the third may be calculated. Should a description in an attribute be changed in a particular viewpoint, for example the subtotal change from $5 to $10, then the following scenario might occur:

1. A new viewpoint is created and described as being a successor to the old viewpoint.

2. All descriptions that were not derived from the changed description are inherited to the new viewpoint.

3. The new description is added in the new viewpoint, any deductions resulting from this new information are made.

4. The descriptions in the new viewpoint describe the changed state of the invoice.

In this case the new viewpoint inherits all but the changed description and the descriptions deduced from the changed description from the old viewpoint. What actions are taken when information in a viewpoint is changed is controlled via sprites. In the example above a simple action is specified: all information not derived from the changed information is inherited into the new viewpoint. Other actions would be to disallow change, in the case of protected information, or to signal a contradiction and allow the user to help resolve it. Examples of these actions are presented in detail in this chapter. In the next paragraphs and in chapter 8 various approaches to handling change are surveyed.

## 6.2 Managing Change

Many approaches have been developed to manage change; we begin with the most simple and proceed to the more sophisticated. In some cases the approach to keeping track of changing information has been via updates to data structures. Systems based on property lists or records such as in Lisp or Pascal have used *put* and *get* types of operations to update and read database information. These are low level operations and have the disadvantage that they provide no support for propagating changes. Thus, deductions based on updated information must be handled explicitly leading to excessive complexity and modularity problems. Languages like FRL [Goldstein, Roberts 77] solve this problem by using triggers on data structure slots (actually the slots of frames), to help propagate changes. When an object is added to a slot in a frame it may trigger an *if-added* daemon that propagates the change. In addition there are *if-needed* and *if-removed* daemons that fire when slots are queried or retracted. The problem with this approach is that there is little support for keeping track of what was deduced and why (particularly between slots in different data structures linked by constraints). This makes changes difficult because deduction dependencies are not explicitly recorded.

The language KRL has been used to implement a knowledge-based personal assistant called ODYSSEY [Fikes 80a]. ODYSSEY aids a user in the planning of travel itineraries by keeping track of what cities a traveler will visit, how the traveler will get to the cities and where the traveler will stay in the cities. In this system *pushers* and *pullers* are used to propagate deductions as a result of updates and to make deductions on reads. A simple dependency mechanism is used to record information dependencies. When a value is changed the dependency mechanism is used to retract those values deduced from the original value and to recalculate new values if possible. The problem is that there is a transition period from the time when a value is changed to the time that all changes are propagated. During this transition period the database is in an inconsistent state and rules may fire making deductions based on inconsistent information. The solution adopted was to retract a value, let all values dependent on the original value be retracted and then assert a new value. This non-monotonic approach avoids the transition period when the database is inconsistent. This technique has efficiency problems since a change involves a retraction and reassertion and is cumbersome since two steps are involved in changing information in the database. In addition, this technique precludes the use of concurrency since retractions and assertions must proceed sequentially. In both KRL and FRL it is necessary to be very careful about the order in which triggers fire for as updates are made there is both new and old information in the database making it difficult to prevent anomalous results due to inconsistencies.

The AMORD system attempts to maintain a globally consistent database at all times. A Truth Maintenance System [Doyle 77] maintains the status of facts, when a fact becomes *out*ed, or disbelieved, the status of all facts that depend on the original fact are also set to out. During the period of time that facts are being outed or reinstated the database is unavailable for the firing of rules. Thus when the rules do fire, they always see an *apparently* consistent database.[6] This system reduces the possibility of making erroneous conclusions considerably at the expense of a global notion of truth and efficiency. In a sense AMORD goes too far because it is impossible to reason

[6] We call a database *apparently* consistent when no contradictions have been derived but the database hasn't been proven to be consistent. Thus some contradictions may be derivable.

within an inconsistent database, thus methods for resolving the inconsistency cannot use the reasoning machinery. We will discuss this point at some length latter.

Steele has developed a constraint based programming language [Steele 80]. In this system a network of nodes and connections is used to build a constraint network. Values deduced by rules at the nodes propagate through the network creating a flow of information through the net from input and constant values to deduced values. A dependency network is also maintained so that when values are changed the dependent values may also be updated. In character this system is an interesting median between the KRL system used for ODYSSEY and AMORD. Like AMORD, Steele's system enforces a global notion of truth, so if a consistent interpretation of the input values is at all possible the system searches for it (using dependency directed backtracking when contradictions are encountered), until it is found. Like KRL, rules can fire on an inconsistent database signaling contradictions. These rules represent false alarms because once the propagation caused by the original change is done, the database is consistent and the fired rules are no longer relevant. Partially because of the "false alarm" problem Steele has devised a system of prioritized queues that defers the processing of fired rules that are likely due to false alarms until rules that are likely to bring the database into a consistent state have fired. Because of the global truth requirement false alarms will not cause inconsistent results as is the case with the KRL system, they will just lead to inefficiency. Steele has prioritized the queues to minimize this inefficiency but in an environment with parallel processors this approach can't work because the queues would represent an unacceptable bottleneck.

A characteristic shared by all these systems is that they are non-monotonic, information is lost when, for example, values of slots are changed in FRL or when inputs are changed in Steele's constraint system. This is a fundamental limitation because it means that the systems are constrained in their history keeping capabilities. If a value of some parameter in one of these systems is changed from A to B causing the implications of B to be deduced and then it is changed back to A there is no way for the system to know the parameter's value was A at a previous time. One might object that this is not the case in AMORD, that when the parameter is changed back to A the status of the relevant facts are simply changed from *out* to *in* and no recomputation is necessary. This may be true but this behavior is implemented by a mechanism beyond the reach of the system's deduction machinery. There is no way for AMORD to reason about the fact that the parameter's value was A.

One might also point out that Steele's constraint system and AMORD's Truth Maintenance system use so-called *nogood* sets to remember past states. Nogood sets are limited and do not represent a general history keeping facility. They are used primarily as an aid to the dependency directed backtracking mechanism to remember particular state configurations that are inconsistent. In addition, nogood sets, like AMORD'S *out*ed facts are not within the grasp of the reasoning mechanism.

The systems described above are inspired by the classical view of problem solving. In this view problem solving is characterized as a search in a state space from a well defined initial state to a well defined goal. A finite set of actions or state transformers are available; the problem solver finds a sequence of state transformers that transforms the initial state into the goal state. The backtracking machinery of these problem solvers is used to perform the search. Often, as in the AMORD, failure to find the requisite sequence of actions signals a contradiction. This invokes the backtracking mechanism that finds assumptions that led to the contradiction. One of these assumptions is randomly selected and retracted and the search continues for a consistent state of affairs. The point is that no reasoning is applied to determine which might be the best assumption to retract or even if the search should continue.

## 6.3 Contradiction Handling with Viewpoints

The systems described above cannot reason directly about contradictions because they are based on logics that suffer from the garbage in garbage out problem (GIGO). Since these system enforce a global notion of truth, when a contradiction exists anything can be derived by their inference rules. Thus when a contradiction is detected the systems deduction machinery is useless. The approach taken by AMORD and Steele's constraint system for example, is to use a mechanism outside the logic to reestablish consistency. Once the world is apparently consistent, the deduction mechanism can operate normally.

The Viewpoint mechanisms provides a method to quarantine inconsistency to within a viewpoint so that reasoning can be done outside of the inconsistent viewpoint and thus valid inferences can still be made. Thus the truth of a statement is relativized to a viewpoint, dispensing with a global truth interpretation for statements. Show below is a diagrammatic representation of how contradiction processing proceeds.

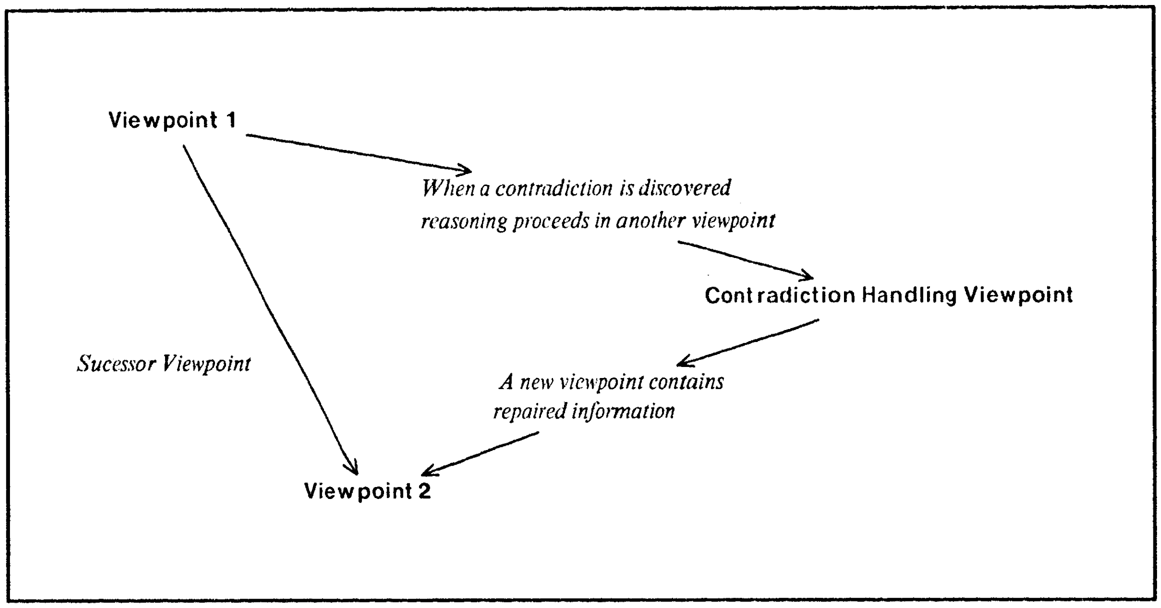

Viewpoint 1
When a contradiction is discovered
reasoning proceeds in another viewpoint
Contradiction Handling Viewpoint
Sucessor Viewpoint
A new viewpoint contains
repaired information
Viewpoint 2

**Figure 6-1:** Contradiction Handling using Viewpoints

The ability to limit the effect of contradictions to within viewpoints and to relativize truth to viewpoints is done by explicitly keeping track of what is believed to be true, i.e. *assertions*, and why it is believed to be true, *justifications*. This information is expressed in the Omega language so it is within reach of the deduction mechanism. Steele has stated that a general purpose programming language isn't one if an implementation for the language cannot be written in the language itself. We agree and recast this statement: a knowledge embedding language isn't one if it can't represent and reason about why it believes what it believes. Given any statement, Omega can answer whether the statement is believed to be true and why, whether the statement is believed to be false and why, or whether Omega doesn't know.

The "I don't know" answer has a precise interpretation. When a person is asked a question like "Is John McCarthy standing at this moment?" and the person is no where near John McCarthy or has no information about what John McCarthy is doing or normally does then the person will usually answer "I don't know." In this case the person can convincingly argue that he *can't* know what John McCarthy is doing and in a sense can prove that he doesn't know what McCarthy is doing. This is not the interpretation of Omega's "I don't know" answer. When a person is asked a question like "Is the square root of 1849 the number 43?" the person may well answer "I don't know." This is the sense of Omega's "I don't know" answer. In the case of the person the "I don't Know" means that the answer is not immediately known but if more time is allowed there is a possibility that the answer can be determined. In the case of Omega the answer is not directly represented in Omega's semantic network but if more resources are allowed it is possible that the answer can be determined.

## 6.4 Assertions and Dissemination

A statement that is believed to be true has more information associated with it than just what the statement itself says. In particular a statement has a justification. For this reason statements that are believed to be true in omega are represented as assertions. An assertion for a statement $\sigma$ in a viewpoint **vp** has the structure shown below. Note that the ***in*** construct is use to indicate the viewpoint in which a description resides.

```
(an Assertion (with unique Content σ in vp)
              (with unique Justification β)) in vp
```

Note that by convention the assertion and the statement reside in the same viewpoint. To motivate the structure of justifications consider that there are 3 operations one would like to be able to do given an assertion:

1. Find out if the assertion is true in a given viewpoint.

2. Find out what justifications an assertion depends on.

3. Find out what assertions depend on a particular justification.

To accommodate these operations, justifications have the following structure:

```
(a Justification
   (with unique Assertion (an Assertion))
   (with unique Type (dor User System Axiom Compound))
   (with unique Time (a Timestamp))
   (with unique number-of-depends-on (a non-negative-integer))
   (with every depends-on (a Justification))
   (with every depended-on-by (a justification))) in viewpoint-vp
```

Justifications can be of several types. The **User** type is a justification for an assertion the user has entered; the **System** type is a justification for assertions that Omega uses in its operation; the **Axiom** type is the justification for an Omega axiom; and the **Compound** type is when a justification depends on other justifications. In this last case the **number-of-depends-on** attribute will be non-zero and it will correspond to the number of **depends-on** ***with*** attributions for the justification. The description in each of these ***with*** attributions will be a justification. The **depended-on-by** ***with*** attributions are used to keep track of all the justifications that depend on the given justification. The **Time** attribution is used to keep track of when and where assertions where made.

Thus given an assertion we can determine its justification and in turn determine what justifications that assertion depends on. In addition, given a justification we can find what assertions depend on it by following the depended-on-by links to other justifications and then the assertion links in those justifications. What remains is how we tell if an assertion is true in a viewpoint and this gets us the relationship between justifications and viewpoints.

*A viewpoint describes a collection of justifications.*

We will need the ability to include new justifications in viewpoints and the ability to talk about the viewpoint itself. The convention we adopt is the following: given a viewpoint description **d** the collection of justifications described by the viewpoint is referred to by the notation **(*a* d)**. Often we will speak of the instance description as being the viewpoint, this is a matter of convenience, the viewpoint is actually the concept of the instance description. We use the instance description to include new justifications in a viewpoint. This is discussed further below.

In the diagram below are shown 2 assertions the user has made. Only the justification for assertion **A-1** is shown completely. Also note that in order not to clutter up the diagram we have not shown that the assertion descriptions and the statements are in the **OFFICE-VIEWPOINT** viewpoint and that the justification is in the **viewpoint-vp** viewpoint.

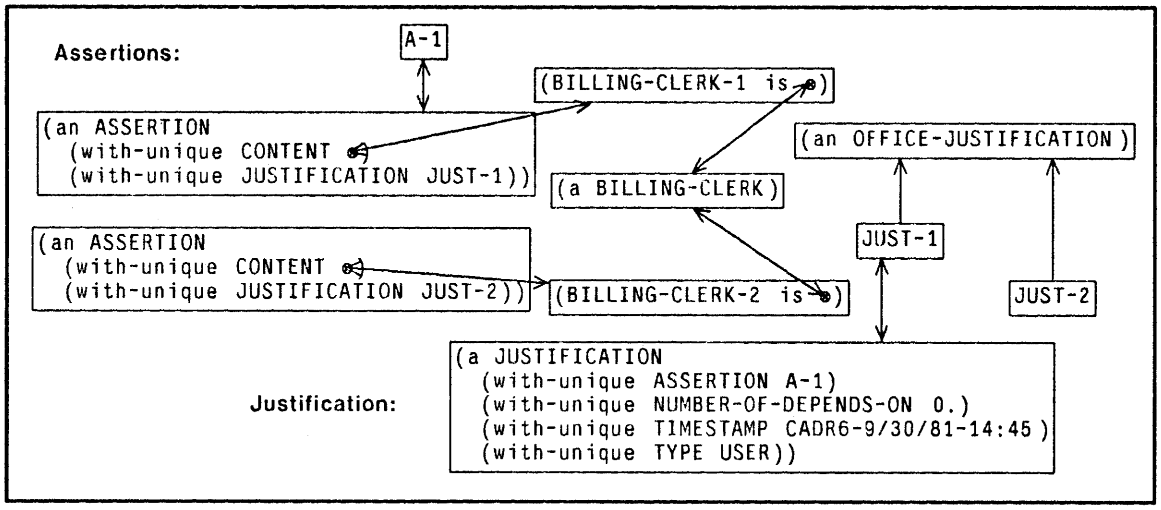


**Figure 6-2:** Two Assertions and Justifications

The process of determining if an assertion is true or not in a particular viewpoint is (almost, see next paragraph) reduced to a matter of determining if the assertion's justification inherits from the viewpoint or not. It is also easy for sprites (rules) to fire on assertions in particular viewpoints (discussed in section 7.4). Intuitively this means that sprites are viewpoint specific. This will be important when we discuss viewpoint inheritance in section 6.5 below. Not only can assertions be inherited between viewpoints but so can sprites.

There is one aspect to assertions that we have not discussed yet. This is the problem that, so far, just the existence of an assertion description with a justification could imply that the assertion is true. This makes it difficult to talk about assertions without their being considered true by virtue of their existence in the description system, certainly not a desirable situation. An assertion should be considered true only if it is asserted. For this reason every description has a disseminated status that is set to true if that description is asserted. A list of the steps that are done when a description is asserted is:

1. Create the assertion description.

2. Create the justification description.

3. Record that the assertion and the justification are individuals.

4. Link the assertion and justification descriptions.

5. Insure that the justification description inherits from the viewpoint in which the assertion is being made.

6. Set the disseminated status to true in the assertion description.

These steps are executed by the *assert* command so that they are not performed by the user explicitly. In fact the user does not have access to the disseminated status except implicitly by creating sprites that are triggered by disseminated descriptions.

## 6.5 Tangled Hierarchies of Viewpoints

Viewpoints, being descriptions, can be structured in flexible ways. Consider describing the work an order entry clerk does. Facts and rules constrain what the order entry clerk does; these facts and rules come from diverse sources. There are facts and rules that arise because the order entry clerk is a member of the order entry department and because the clerk is working on a specific order. We can use viewpoints as names for collections of facts and rules that should impinge on a particular situation. The diagram below shows a hierarchy of such viewpoints.

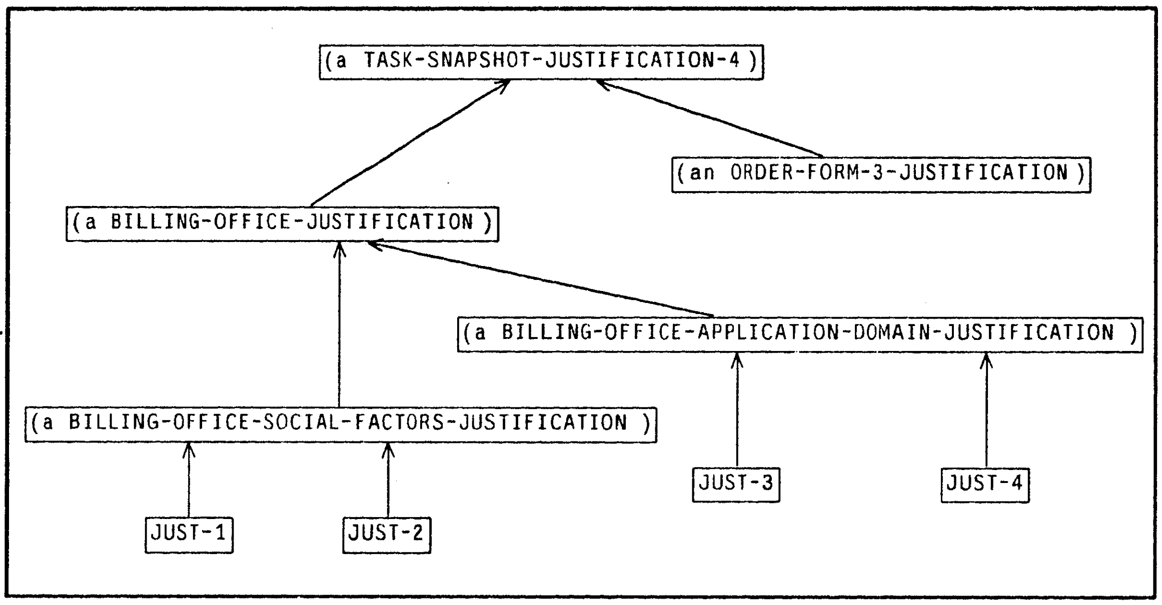

(a TASK-SNAPSHOT-JUSTIFICATION-4 )
(an ORDER-FORM-3-JUSTIFICATION )
(a BILLING-OFFICE-JUSTIFICATION )
(a BILLING-OFFICE-APPLICATION-DOMAIN-JUSTIFICATION )
(a BILLING-OFFICE-SOCIAL-FACTORS-JUSTIFICATION )
JUST-3
JUST-4
JUST-1
JUST-2

**Figure 6-3:** A Structure of Encompassing Viewpoints

When a collection of justifications inherits from another collection of justifications we say that the viewpoint corresponding to the second *encompasses* the viewpoint corresponding to the first. In the diagram above the viewpoint **BILLING-OFFICE-VIEWPOINT-17** encompasses the two viewpoints **VIEWPOINT-9** and **VIEWPOINT-14**. Note that the viewpoint that refers to a most specific situation is the most general viewpoint, in this case **TASK-SNAPSHOT-VIEWPOINT-5**, and it encompasses all the other viewpoints shown. This is because the more general a viewpoint is, the more justifications there are that the viewpoint encompasses. In the above example viewpoint **VIEWPOINT-9** encompasses the rules and facts from the organizational domain pertaining to the billing office as discussed in the Office Semantics chapter. The viewpoint **VIEWPOINT-14** encompasses general facts and rules concerning the Billing Office applications domain. The combination of these viewpoints into **BILLING-OFFICE-VIEWPOINT-17** encompass the facts and rules pertaining to the billing office in general. More specific facts and rule pertaining to a particular order form are encompassed by **VIEWPOINT-6**. All these viewpoints or sources of knowledge combine into the **TASK-SNAPSHOT-VIEWPOINT-5**.

So far we are able to state that justifications inherit from classes of justifications or, from another point of view, viewpoints encompass certain justifications. So from the above diagram we can state that **JUST-1** inherits from **(*a* TASK-SNAPSHOT-VIEWPOINT-5)** or **TASK-SNAPSHOT-VIEWPOINT-5** encompasses **JUST-1**. However we need to be able to talk about the viewpoint itself, ie, about the collection of things the viewpoint encompasses to describe them as, for example, being in contradiction. Note that we can't say that:

**(*a* TASK-SNAPSHOT-VIEWPOINT-5)** **Wrong!**
***is* (*a* Contradictory-Viewpoint)**

because then **JUST-1** would be a contradictory viewpoint. The solution is to use the approach described in section 5.2. We use the concept of the instance description to describe properties of the class the instance description describes. We have used this tacitly in the above paragraphs when we referred to viewpoints by just the mention of their concepts. Thus, to express the fact that a viewpoint is in contradiction we would say:

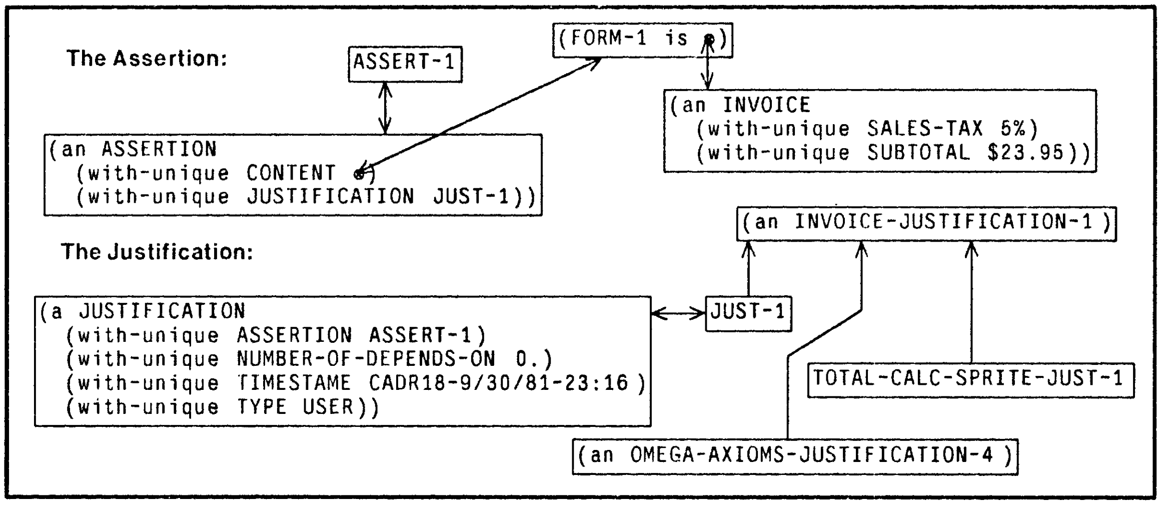
**TASK-SNAPSHOT-VIEWPOINT-5** **Correct**
***is* (*a* Contradictory-Viewpoint)**

The ability to describe and reason about viewpoints is very useful. We will discuss further later. Now it is time to present an example.

## 6.6 Tracking Change With Viewpoints

As an introduction to the viewpoint mechanism we consider a simple example in this section. In this example the value of a field in a form is updated when the form is described as having a new value different than its old value. In later chapters more complicated uses of the viewpoint mechanism will be presented.

Change in a viewpoint is handled in the same way as contradictions are. Information is only added to viewpoints, never changed. Thus, when information in a viewpoint is overspecified, e.g., the total field of an invoice has two distinct values such as $23.00 and $26.00, a contradiction is signaled and a new viewpoint is created for the information. Consider the diagram shown below in which **FORM-1** is described as an invoice with subtotal and sales tax attributes of $23.95 and 5% respectively.



The Assertion:
ASSERT-1
(FORM-1 is •)
(an ASSERTION (with-unique CONTENT •) (with-unique JUSTIFICATION JUST-1))
(an INVOICE (with-unique SALES-TAX 5%) (with-unique SUBTOTAL $23.95))
(an INVOICE-JUSTIFICATION-1 )
The Justification:
(a JUSTIFICATION (with-unique ASSERTION ASSERT-1) (with-unique NUMBER-OF-DEPENDS-ON 0.) (with-unique TIMESTAME CADR18-9/30/81-23:16 ) (with-unique TYPE USER))
JUST-1
TOTAL-CALC-SPRITE-JUST-1
(an OMEGA-AXIOMS-JUSTIFICATION-4 )

**Figure 6-4:** An Invoice Described in a Viewpoint

Notice that the justification **JUST-1** is encompassed by the viewpoint **INVOICE-VIEWPOINT-1**. In addition to this justification there is a justification for a sprite, **TOTAL-CALC-SPRITE-JUST-1**, (which we use below) and another viewpoint, **OMEGA-AXIOMS-VIEWPOINT-1**. This last viewpoint contains the justifications for the various sprites that implement Omega's axioms. We won't always show all the justifications in a viewpoint as we have this time. Suppose we have a sprite (whose justification we see above) that, given an assertion describing the subtotal and tax of a form, deduces what the total is for the form. (How this sprite is written will be considered in section 7.4.) After this sprite fires we have the following situation.

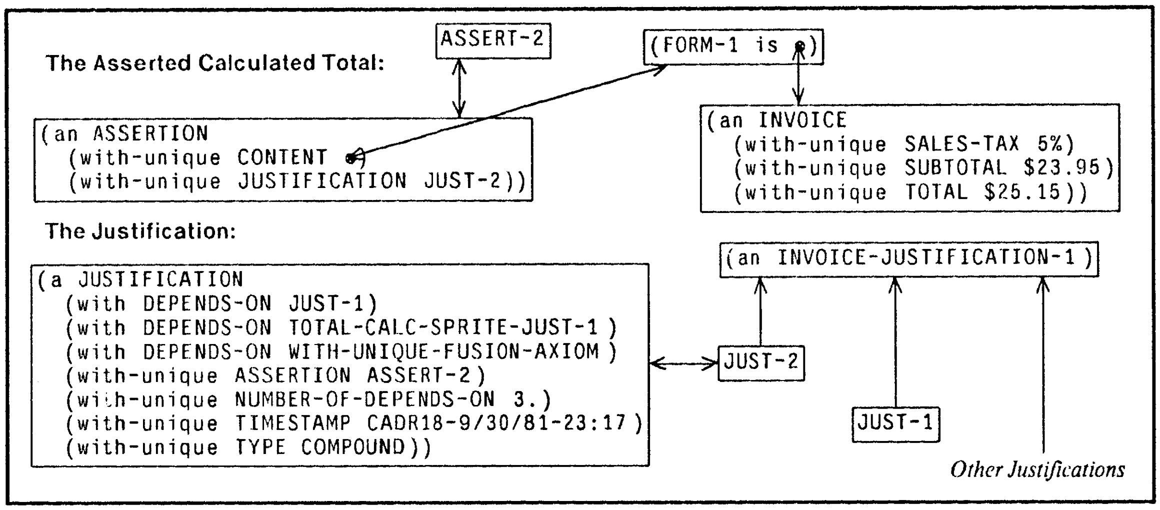
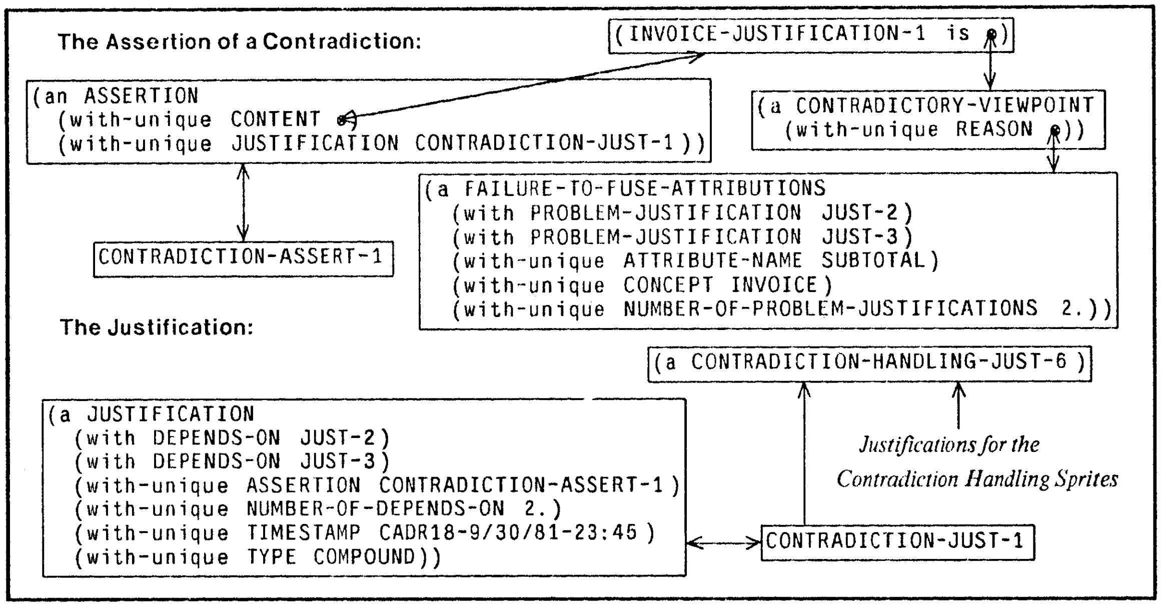

The Asserted Calculated Total:
ASSERT-2
(FORM-1 is •)
(an ASSERTION
(with-unique CONTENT •)
(with-unique JUSTIFICATION JUST-2))
(an INVOICE
(with-unique SALES-TAX 5%)
(with-unique SUBTOTAL $23.95)
(with-unique TOTAL $25.15))
The Justification:
(an INVOICE-JUSTIFICATION-1 )
(a JUSTIFICATION
(with DEPENDS-ON JUST-1)
(with DEPENDS-ON TOTAL-CALC-SPRITE-JUST-1 )
(with DEPENDS-ON WITH-UNIQUE-FUSION-AXIOM )
(with-unique ASSERTION ASSERT-2 )
(with-unique NUMBER-OF-DEPENDS-ON 3.)
(with-unique TIMESTAMP CADR18-9/30/81-23:17 )
(with-unique TYPE COMPOUND))
JUST-2
JUST-1
Other Justifications

**Figure 6-5:** An Invoice with a Deduced Total Attribute

Notice that the total has been calculated for the form. Notice also the new justification **JUST-2** that has been added to the viewpoint. This is a compound justification that depends on the previous justification **JUST-1**; the justification for the sprite that did the calculation **TOTAL-CALCULATION-JUST-1**; and on the justification for the fusion axiom for ***with unique*** attributions **WITH-UNIQUE-FUSION-AXIOM**. Now suppose that the subtotal must be changed, this would be the case if running subtotals and totals were being displayed as the form was being filled out interactively. We assume that the assertion for the new value for the subtotal has justification **JUST-3**. When the fuse attribution axiom tries to fuse the two values for the subtotals it complains because they are individuals and they are not the ***same*** and thus cannot be fused. The fuse attribution axiom asserts the viewpoint to be in contradiction resulting in the following situation.


The Assertion of a Contradiction:
The Justification:
Justifications for the
Contradiction Handling Sprites

```
(INVOICE-JUSTIFICATION-1 is *)

(an ASSERTION
  (with-unique CONTENT *)
  (with-unique JUSTIFICATION CONTRADICTION-JUST-1 ))

(a CONTRADICTORY-VIEWPOINT
  (with-unique REASON *))

CONTRADICTION-ASSERT-1

(a FAILURE-TO-FUSE-ATTRIBUTIONS
  (with PROBLEM-JUSTIFICATION JUST-2)
  (with PROBLEM-JUSTIFICATION JUST-3)
  (with-unique ATTRIBUTE-NAME SUBTOTAL)
  (with-unique CONCEPT INVOICE)
  (with-unique NUMBER-OF-PROBLEM-JUSTIFICATIONS 2.))

(a CONTRADICTION-HANDLING-JUST-6 )

(a JUSTIFICATION
  (with DEPENDS-ON JUST-2)
  (with DEPENDS-ON JUST-3)
  (with-unique ASSERTION CONTRADICTION-ASSERT-1 )
  (with-unique NUMBER-OF-DEPENDS-ON 2.)
  (with-unique TIMESTAMP CADR18-9/30/81-23:45 )
  (with-unique TYPE COMPOUND))

CONTRADICTION-JUST-1
```

**Figure 6-6:** A Viewpoint Asserted to be in Contradiction

The assertion that the viewpoint is in contradiction depends on the two contradicting justifications JUST-2 and JUST-3. The reason for the contradiction is described as a failure to fuse. The description of the fusion failure contains enough information for contradiction handling sprites to take appropriate action.

What happens at this point depends on the sprite that is handling contradictions for the particular concept and attribute name combination. In our case it will simply assume that the latest value for the attribute is the one of concern. Shown be low is how the new and old viewpoints are linked.

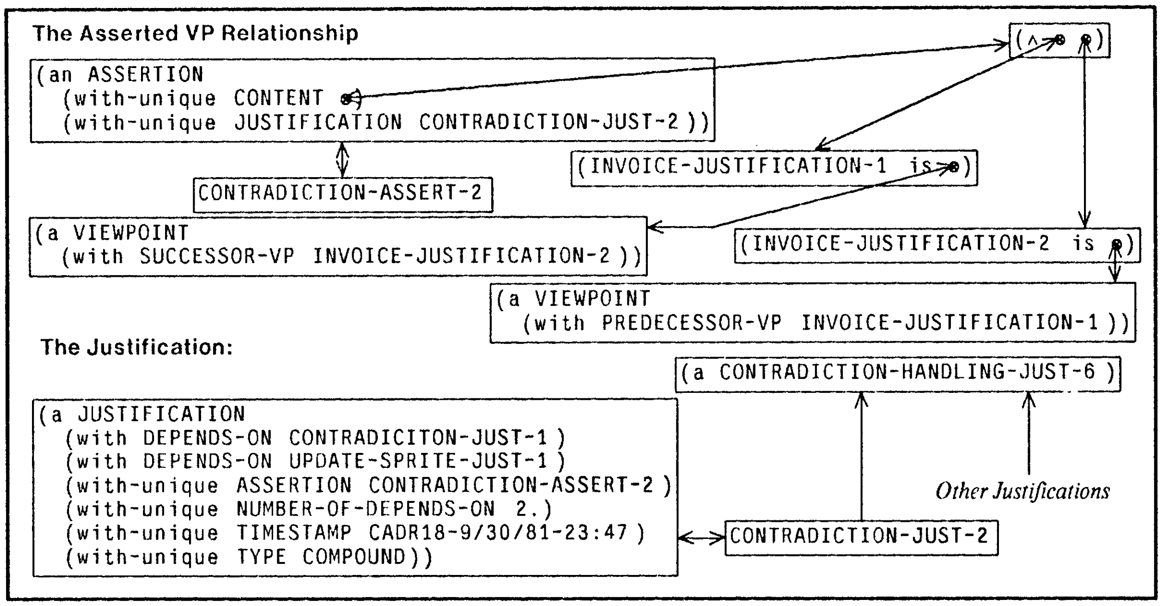

The Asserted VP Relationship
(an ASSERTION
(with-unique CONTENT
(with-unique JUSTIFICATION CONTRADICTION-JUST-2 ))
CONTRADICTION-ASSERT-2
(INVOICE-JUSTIFICATION-1 is
(INVOICE-JUSTIFICATION-2 is
(a VIEWPOINT
(with SUCCESSOR-VP INVOICE-JUSTIFICATION-2 ))
(a VIEWPOINT
(with PREDECESSOR-VP INVOICE-JUSTIFICATION-1 ))
The Justification:
(a CONTRADICTION-HANDLING-JUST-6 )
(a JUSTIFICATION
(with DEPENDS-ON CONTRADICITON-JUST-1 )
(with DEPENDS-ON UPDATE-SPRITE-JUST-1 )
(with-unique ASSERTION CONTRADICTION-ASSERT-2 )
(with-unique NUMBER-OF-DEPENDS-ON 2.)
(with-unique TIMESTAMP CADR18-9/30/81-23:47 )
(with-unique TYPE COMPOUND))
Other Justifications
CONTRADICTION-JUST-2

Figure 6-7: The Description of a New Viewpoint

As a result of the contradiction handling a new viewpoint, **INVOICE-VIEWPOINT-2**, and **INVOICE-VIEWPOINT-1** are linked by the successor and predecessor relations. The justification for this new assertion is a justification which depends on the justification for the contradiction handling sprite **UPDATE-SPRITE-JUST-1** (the one that picks the most recent value to be the appropriate value in the new viewpoint) and on the justification for the statement that asserted the first viewpoint to be in contradiction **CONTRADICTION-JUST-1**. Both contributed to the deduced knowledge, if either of these justifications is not supported then the assertion **CONTRADICTION-ASSERT-2** cannot be justified. The final result of the contradiction handling is depicted in the figure shown below.

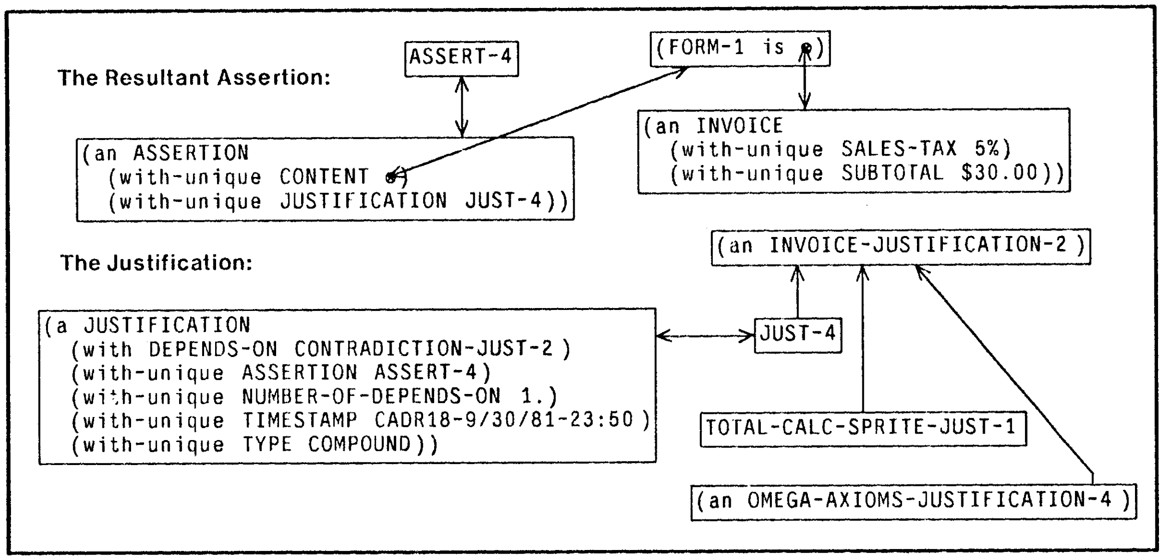

The Resultant Assertion:
ASSERT-4
(FORM-1 is *)
(an ASSERTION (with-unique CONTENT *) (with-unique JUSTIFICATION JUST-4))
(an INVOICE (with-unique SALES-TAX 5%) (with-unique SUBTOTAL $30.00))
The Justification:
(an INVOICE-JUSTIFICATION-2 )
(a JUSTIFICATION (with DEPENDS-ON CONTRADICTION-JUST-2 ) (with-unique ASSERTION ASSERT-4) (with-unique NUMBER-OF-DEPENDS-ON 1.) (with-unique TIMESTAMP CADR18-9/30/81-23:50 ) (with-unique TYPE COMPOUND))
JUST-4
TOTAL-CALC-SPRITE-JUST-1
(an OMEGA-AXIOMS-JUSTIFICATION-4 )

**Figure 6-8:** The Result of Contradiction Handling

In this figure the new value for the subtotal can be seen. In addition to the justification for the new assertion and for the sprite that will calculate the new subtotal, this viewpoint also encompasses, as before, the viewpoint in which the sprites that implement the Omega axioms are justified. Note that the contradiction handling sprite knew that the subtotal and sales-tax were independent although they were originally asserted by the same statement. This allowed the sprite to bring the sales-tax into the new viewpoint. This is not always the case.

This example has demonstrated one way updating of information in a viewpoint might work. An attempt to describe the subtotal in a viewpoint led to a contradiction because two distinct values were specified for the same attribution. This led to an assertion that the viewpoint was in contradiction and the handling of the contradiction via sprites. The sprites picked the descriptions of interest in the old viewpoint and brought them into the a new viewpoint. In addition the new viewpoint was described as being a successor to the old viewpoint and the old viewpoint was described as a predecessor to the new viewpoint. Information that depended on the changed viewpoint was not brought into the new viewpoint: the total was not brought into the new viewpoint. The situation above is as it was at the beginning of the example; we have shown it just prior to the point when the sprite that will calculate the new total will fire.

We have shown the example in this manner to demonstrate how contradictions might be handled.

Depending on the application updates may be handled this way or they may be handled algorithmically instead of using the reasoning mechanism. This addresses the issue of when to use algorithms and when to use reasoning which will be discussed below. The new viewpoint would be created without waiting for a contradiction to occur. Processing from this point on would proceed in a manner similar to the example.

## 6.7 Knowledge and Reasoning vs Algorithms

As noted above and in other places in this document we have distinguished two fundamentally different ways to perform a task, algorithmically or using knowledge and reasoning. When faced with the decision of how to approach a certain task the criteria we use is the amount of problem solving needed to perform the task.

If the task requires much problem solving where the structure of the problem is likely to change frequently, where exceptions are frequent and each instance of the problem must be handled on a case by case basis, and where new knowledge useful in achieving the goal frequently arises then a knowledge based approach is called for. If, on the other hand, task is well defined and a satisfactory way to accomplish the task for all conditions is known then an algorithmic approach is called for.

In the case where problem solving exceptions occur in algorithmic tasks it is necessary that the problem solver have access to the details of the task at hand in order to understand the nature of the exception. Kornfeld [Kornfeld 82] describes a method, called Virtual Collections of Assertions (VCAs) that allows the problem solver to interface cleanly with tasks that are done algorithmically.

It is often useful for a process to proceed in an algorithmic fashion but notifying the problem solver of certain important events. For example consider tracking a mouse cursor on a display. The actual tracking of the cursor may be done in compiled code, whereas the fact that the cursor has moved over a new window or that a button has been pushed may cause the assertion of a statement in the description system. Reasoning based on this assertion may or may not affect further tracking of the mouse. Another example is considered later in this document and concerns Sponsors, resource management objects that control computation used in achieving a goal. Sponsors perform their tasks algorithmically. However when an important event occurs--they run out of allocated resources or they can proceed no further in their processing activities--they notify Omega of the situation by making assertions.

## 6.8 Different Types of Viewpoints

The kinds of viewpoints that we have described in this chapter are called *Logical* viewpoints. Within these viewpoints logical rules of inference are operable. For example if A and A ⇒ B are asserted in a logical viewpoint one can conclude B in that viewpoint.

In contrast to logical viewpoints are *Opaque* viewpoints. These viewpoints are useful for modeling a person's beliefs. For example if we have a viewpoint that represents someone's beliefs and we know that the person believes A and A ⇒ B we *cannot* conclude that the person believes B and thus we cannot assert B in the opaque viewpoint.

A third viewpoint we distinguish is the *Omega* viewpoint. The Omega viewpoint is used as a meta-descriptional capability to talk about the structure of descriptions. For example suppose we want to talk about the structure of an ***is*** statement in the viewpoint **vp**. Then using the ***in*** operator to indicate viewpoint we can say:

```
((A is B) in vp) is (an Is-Statement
                        (with unique Subject A)
                        (with unique Object B))
in
Omega
```

The type of viewpoint is controlled by what assertions and sprites are present in that viewpoint. A logical viewpoint includes sprites and assertions that implement Omega's logical rules of inference. The opaque and Omega viewpoints would have special rules of inference that control the deductions that can be made in these viewpoints. Further development of these ideas are beyond the scope of this dissertation but they look promising in that they give the knowledge base designer great flexibility in the ways viewpoints may be used.

## 6.9 Summary

Omega separates information into different viewpoints. Contradictions are contained within viewpoints. The propagation of information between viewpoints is controlled via justifications. An advantage of the approach using viewpoints is that the system has a historical character. This is an important step toward our goal of aiding office workers in problem solving about dynamic processes. Viewpoints can be used as historical records of past processes, as an aid in tracking ongoing processes, and as an aid to determine the implications of postulated actions. Descriptions about viewpoints can be used to express the changes between viewpoints.

# Chapter Seven

# The Omega Implementation

This chapter describes the implementation of the Omega Description System on the Lisp Machine. The implementation described is the second implementation of Omega, the first implementation was written by G. Attardi and M. Simi. The primary goals of the Omega implementation are:

- To explore the implementation and use of Viewpoints.

- To explore the implementation of description based Sprite invocation.

- To explore control structures using sprites and sponsors in an inheritance network.

- The use of axiomatization of Omega as an interface between the users and implementors of the system.

- A unification of systems that are object-centered (systems that have their information structured around objects) and systems that are fact-centered (system that have their information structured in a fact database of assertions and goals).

- To explore the implementation and maintenance of declarative structures using objects and message passing.

- To keep the individual mechanisms of the description system as simple as possible. An organization of well-complemented simple mechanisms is judged superior to a collection of seemingly powerful mechanisms the are hard to combine.

- To interface cleanly with Lisp Machine Lisp. The facilities of the description system should be usable from lisp code without special readers or translation.

Omega is implemented using the Lisp Machine Flavor system [Weinreb, Moon 80]. The Flavor system provides an object oriented programming environment that supports message passing; it is a further evolution of such object oriented languages as Smalltalk, CLU and Simula [Ingalls 78, Liskov, Snyder, Atkinson, Chaffert 77, Dahl, Myhrhaug, Nygaard 70]. The major capability that the Flavor System has with respect to other object oriented languages is that it maintains an inheritance hierarchy of object classes, as does Smalltalk and Simula, and provides for inheritance from multiple super classes of objects.

In the description of Omega's implementation there are several programming techniques to note. These techniques helped to structure the implementation in a way that is relatively easy to change and intuitively appealing. The most interesting techniques are:

- **Serializers** - Used for scheduling of concurrent requests. A serializer manages requests to the description system so they are processed in an orderly fashion [Hewitt, Attardi, Lieberman 79].

- **Sponsors** - Resource control is accomplished through the use of a trees of resource control objects called Sponsors. Sponsors help implement a more general event based control structure than Lisp's recursive control structure [Kornfeld 79].

- **Inheritance of Flavors** - The inheritance capabilities of the Flavor system are used to aid modularity and save duplicated code.

In the next section (7.1) the top level structure of the description system is described along with a scenario of what happens when a description is given to the description system. In section 7.2 the structure of the flavor objects that implement descriptions is discussed along with the messages descriptions receive. In section 7.3 the process by which a list representation of a description is mapped to its Flavor object is described. Section 7.4 describes the implementation of Sprites and their description based invocation scheme. Section 7.5 describes Omega's approach to resource control using Sponsors. Section 7.6 describes the Omega's user interface including facilities that aid interaction with Omega and facilities for extending Omega.

## 7.1 Overall Description

The description system as well as its constituent parts are implemented using Flavor objects. Shown below is the description system object and its constituent parts. The single-headed arrows indicate that the object at the base of the arrow has the object at the head of the arrow as an *acquaintance.*[7] The double headed arrow indicates mutual acquaintancy.

[7] Acquaintancy means that an object has access to its acquaintance and can send it messages directly. In this case a pointer is the means of access; in a multiprocessor implementation a more indirect form of access would be used.

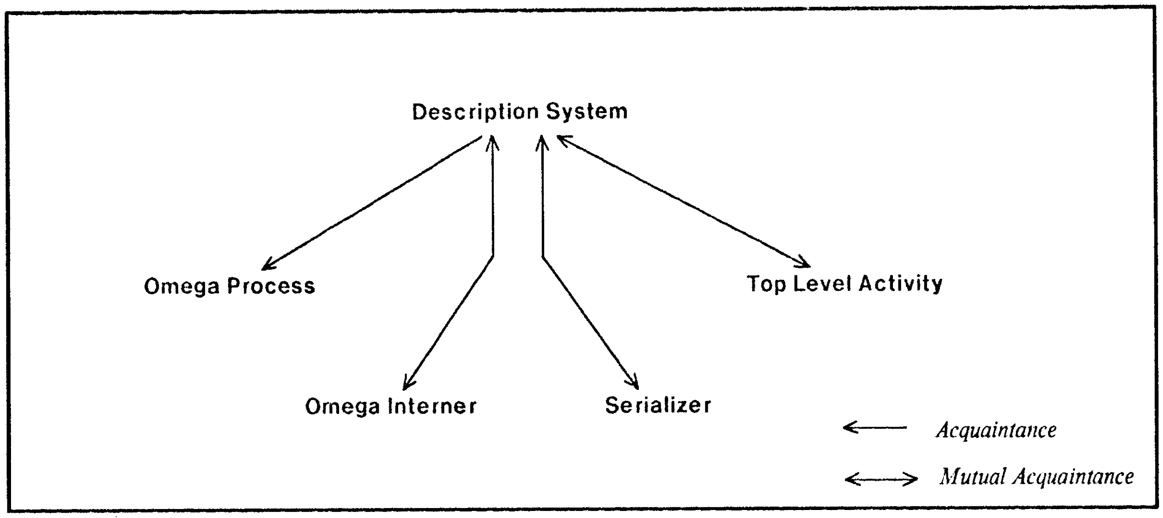


**Figure 7-1:** The Components of the Omega Description System

The description system is composed of the following components:

- **A Process** - Each instantiation of a description system runs in its own process. This was done so multiple Lisp Machine Processes could run in parallel and use the description system. This increases the robustness of the description system since a crash in a process using the description system will not affect the description system's process. Thus the description system will still be available for other processes.

- **An Interner** - The process by which descriptions are translated from their external list representation, eg **(an Office-Procedure-Goal)**, to their corresponding flavor objects is called internment. The internment process insures that each use of the external representation will map to the same Flavor object. If no Flavor object exists for the external representation of a description then one is created.

- **A Serializer** - Since Omega runs in its own process, requests to the description system are scheduled via a serializer [Hewitt, Attardi, Lieberman 79]. Messages sent to the description system are intercepted by the serializer and forwarded to Omega when it is ready for the next request. When a reply from the description system is given to the serializer, the serializer gives the reply to the process waiting for the reply.

- **A Sponsor** - Sponsors are the way Omega does processor resource control. They are used to control processing that proceeds in parallel. The Top Level Sponsor is the root of a tree of sponsors that control parallel computations. Sponsors are used as a more flexible control structure that the recursive control structure provided by Lisp.

### 7.1.1 Questions Concerning the Structure of the Description System

There are several questions of interest to address concerning the structure of the description system. First, why is the description system itself implemented as a Flavor object. One can argue that in a single workstation only a single instantiation of the description system is desirable. Multiple instantiations of a descriptions system in a single workstation may cause communications problems between the descriptions systems and lead to problems in manipulating knowledge. This problem is seen as inherit to the manipulation of knowledge in an open knowledge domain. As Simon has pointed out [March, Simon 63], a major structuring force in organizations is the fact that human knowledge processors have limited capabilities. Thus humans structure their problem space and divide up problem solving responsibilities. The resulting situation can be characterized as centers of expertise communicating over channels of limited bandwidths. A similar scenario is expected for descriptions systems. There is no apriori reason to expect that centers of expertise (description systems) should be allocated on a one-per-workstation basis. The situation is further crystallized when considering description systems in a distributed environment. In this case the problem of communicating between description systems cannot be avoided for it is unacceptable to have one centralized description system in a distributed network.

Implementing a descriptions system as an object allows one to make multiple description systems. In this way mechanisms for communicating between descriptions systems can be examined and the question of what exactly a description system is composed of can be addressed. Although communications between description systems is not a subject addressed by this dissertation it is expected to be a question for future examination.

There is a practical reason for allowing multiple description systems: robustness. A description system runs in its own process to isolate itself from the vagaries of client processes. A similar reason exists for having multiple descriptions systems. Why should a description system concerned with describing the details of a graphical interface be sensitive to problems that may develop in the description system concerned with the structure of office procedure goals. It is true that the workstation is to be an integrated environment. However this should not be done at the expense of modularity.

Another question concerns the serializer. The serializer represents a bottleneck when the description system is structured as presented. The reason for this is more a question of interfacing the description system with Lisp Machine Lisp than a requirement of the description system. The Lisp Machine has a theory of concurrency built around stacks and a recursive control structure. Omega has a different theory of concurrency built around events, Sprites, and Sponsors (more on these in section 7.5.) In effect, Omega implements its own theory of concurrency within a single Lisp Machine process. In order to allow Lisp Machine processes to use the description system the serializer was employed. In a more pure system serializers would be used at a finer level of granularity rather than around the whole description system. These serializers would control access to objects such as parts of the interner that deal with the hash table, descriptions when their instance variables were being modified, and modification of the instance variables of Sponsors.

### 7.1.2 A Simple Scenario

A typical request to the description system is to request the flavor object corresponding to the external representation of a description, to make an assertion or to establish a goal. Assertions and goals are discussed in section 6.4. To retrieve a flavor object corresponding to the external representation of a description, the representation is sent to the Interner. Here the representation is hashed and the corresponding object is retrieved from a hash table. If the object is found, it is given to the serializer and another request is processed. If the object is not found it is created, placed in the hash table and *installed* in the description lattice. The idea behind installation is to place description in the description lattice in such a way that all Sprites relevant to the description will be inherited to the description. This will be further discussed in section 7.3.

In the following pages at times a distinction between a description and its list representation will be made. In the notation this distinction is made in the following manner. A description object will have its keywords in italics for example:

**(*an* Office-Procedure (*with* Goal (*an* Office-Procedure-Goal))**
**(*with every* Action (*an* Office-Procedure-Action))**
**(*with unique* Procedure-ID (*an* Eight-Digit-Number)))**

The list representation for the same description will appear as:

**(an Office-Procedure (With Goal (an Office-Procedure-Goal))**
**(WithEvery Action (an Office-Procedure-Action))**
**(WithUnique Procedure-ID (an Eight-Digit-Number)))**

This will only be important during this chapter.

## 7.2 Anatomy of a Description

Before discussing the structure of descriptions some understanding of the Flavor system is required since descriptions are implemented in terms of Flavor objects. A Flavor object is composed of instance variables (or slots) where the object's state is kept and a method dictionary. Flavor objects are sent messages which consist of a keyword and arguments. When a flavor object receives a message, it invokes the method in the method dictionary that corresponds to the keyword in the message. This method can perform state changes, send further messages, and finally return a value. A flavor object is defined using the following construct:

```
    (Defflavor <flavor-name> <instance-variable-list> <superiors-list>
               <options>)
```

Where <flavor-name> is the name of the Flavor being defined, <instance-variable-list> is list of the Flavor's instance variables and <superiors-list> is a list of flavors this flavor inherits from. The <options> are optional and will not be of relevance to this discussion but they control such things as accessibility and initialization of instance variables, required instance variables and flavors etc.

For the purposes of this discussion, flavor inheritance can be illustrated by the following example. Suppose **flavor-1** is defined by the following statement:

```
    (Defflavor flavor-1 (a-slot b-slot) (flavor-2 flavor-3))
```

Thus **flavor-1** has the two instance variables **a-slot** and **b-slot** and inherits from **flavor-2** and **flavor-3**. The inheritance means that **flavor-1** will also have all the instance variables that **flavor-2** and **flavor-3** have. In addition, **flavor-1** will inherit any methods that are defined for **flavor-2** and **flavor-3**. Methods can be shadowed: suppose **flavor-1** defines a method for the keyword **:YOU-ARE**, if either **flavor-2** or **flavor-3** define a method for this keyword, that method will not be inherited and only the method defined explicitly for **flavor-1** will be used.

The Flavor system's inheritance capabilities are used to build a hierarchy of description types. The motivation for this is that there are certain characteristics that all types of descriptions have in common. Other characteristics are shared by a smaller group of description types, statements for example, and some characteristics are unique to a particular description type, an instance description for example. This arrangement facilitates adding or changing the behavior of select groups of description types.

This hierarchy is shown below in figure 7-2. The Flavor instance variables and methods common to all descriptions are defined on the Basic-Descr object. Those instance variables and methods common to all statements are defined on the Basic-Stmt object. The figure is the Flavor hierarchy for the complete implementation of Omega. As will be seen in section 7.6 on the User Interface, more kinds of flavor objects can be defined with ease. As can be seen the hierarchy is not strict for the ***true*** and ***false*** objects are both statements and atomic descriptions. Future implementations of Omega will take more advantage of multiple inheritance capabilities base on what was learned from this implementation.

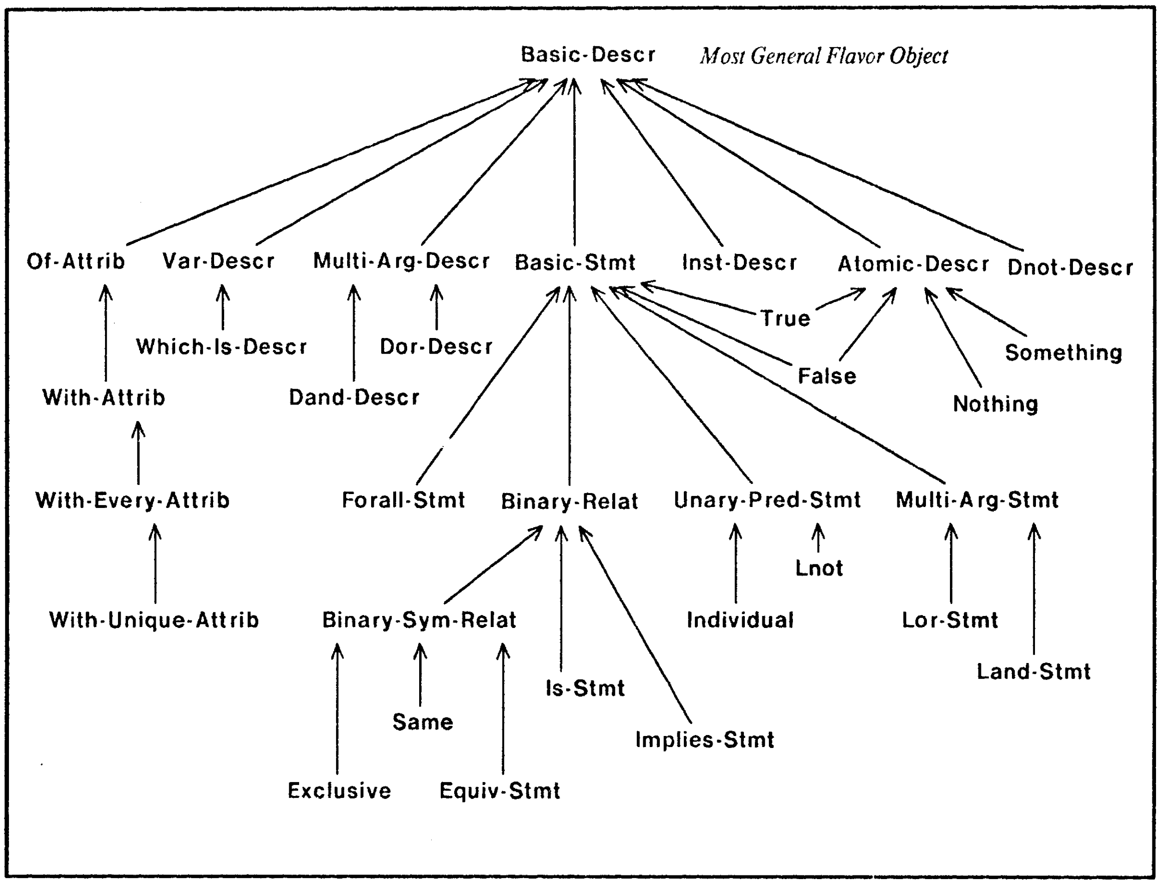


**Figure 7-2:** Inheritance Structure of Flavor Objects for Descriptions

Shown below in figure 7-3 are the instance variables that each description, regardless of type, has. These instance variables are defined for the Basic-Descr object.

```
(Defflavor Basic-Descr
 (Inherit-From        ;Descriptions this one inherits from
  Inherit-To          ;Descriptions this one inherits to
  Same                ;Descriptions Same with this one
  Negation            ;This description's negation
  Individual-P        ;Whether description is an individual
  Exclusions          ;Descriptions exclusive with this one
  Mono-descrs         ;Descriptions related by Monotinicity axiom
  Ex-Rep              ;External representation
  Sprites             ;List of inherited Sprites
  Disseminated-P      ;Whether this description has
                      ;been disseminated.
  Description-System  ;The description system object this
                      ;description belongs to.
  Marks)              ;Mark depository for graph traversing
                      ;algorithms
 (Id-Mixin))          ;A flavor which generates interned
                      ;lisp symbols, and binds this
                      ;flavor object to it (for debugging)
```

**Figure 7-3:** Structure of the Basic-Descr Object

Most of the instance slots are self explanatory. The description lattice is maintained via the pointers in the **Inherit-From**, **Inherit-To**, and **same** slots. The **ex-rep** slot contains the fully expanded external representation of the description object. The information in the **Individual-P** and **Exclusions** slots is kept for efficiency considerations. They contain information about the description object that cannot be directly represented by the **is** relationship as discussed in section 5.3.

```
(Defflavor Inst-Descr
 (Concept          ;Instance description's concept
  Attributes)      ;Instance description's attributes
 (Basic-Descr))    ;Include Basic-Descr's structure
```

**Figure 7-4:** Flavor Definition of an Instance Description

The structure of the Instance Description flavor object is shown above in figure 7-4. Note that this description inherits the structure of **Basic-Descr** and adds the two slots **Concept** and **Attributes.** The **Concept** and **Attributes** are two parts that make up the instance description. The **Concept** is a description, typically an atomic description and **Attributes** is a list of descriptions, typically one of the four possible attribution types.

```
(Defflavor Of-Attrib
 (name              ;Attribute name
  description)      ;Attribute description
 (Basic-Descr))

(Defflavor With-Attrib ()
 (Of-Attrib))
```

Figure 7-5: Flavor Definition for With and Of Attributions

In figure 7-5 the structure of the Of and With attribution Flavor objects are shown. The Of attribution contains the two parts **name**, which is usually an Atomic Description, and **description** which is the attribute description. Note that the With attribution inherits the instance variables from the Of attribution and does not add any of its own. The reason for making a distinction between these two flavor objects is that there will be methods defined on the With attribution that are peculiar to it. Figure 7-6 shows the definition of the Basic-Stmt, Binary-Relat, and Is-Stmt Flavor object. The Binary-Relat object is where the behavior common to all binary relations is defined. It adds the two slots **Descr-1** and **Descr-2**; these are the descriptions for which the binary relation holds. The Is-Stmt is one example of a binary relation the inherits the **Descr-1** and **Descr-2** slots from the Binary-Relat object.

```
(Defflavor Basic-Stmt
 ()                 ;No instance Variables.
 (Basic-Descr))     ;Inherit Basic-Descr's structure

(Defflavor Binary-Relat
 (Descr-1           ;First argument of binary relation
  Descr-2)          ;Second argument of binary relation
 (Basic-Stmt))      ;Inherit Structure of Basic-Stmt

(Defflavor Is-Stmt () (Binary-Relation))
```

Figure 7-6: Structure of Basic-Stmt, Binary-Relation, and Is-Stmt Objects

The other description types are defined in a manner similar to the definitions for Instance Descriptions and Is statements. Once the slots are defined for a description the behavior of the description can be defined. The behavior of a description is defined in two ways, by defining Flavor methods, either directly for the Flavor object or indirectly via inheritance and by defining Sprites for the particular description. In this section the methods for the descriptions are discussed. Sprites are considered later.

There are various messages sent to descriptions that are of interest. These messages are inplemented in terms of the Flavor system's methods. By convention all messages sent to flavor objects are prefixed by a colon.

**:you-are** This messages establishes an inheritance relation between two descriptions, the target and a description which is an argument. A second argument is a justification description, since inheritance always has an associated justification. This message modifies the inherit-from slot of the target and the inherit-to slot of the argument description.

**:you-are-same** Similar to the you-are message but sets up a same relation between the target description and the argument description. A justification is also included.

**:you-exclude** This message is sent as a result of the ***exclusive*** statement. The result is that the exclusion slot of the target and the argument descriptions are modified. It also takes a justification.

**:fast-inherit-from** This message is sent to a description with a description argument and a viewpoint. It determines if there is a description lattice link between the target and argument description in the given viewpoint. This message just checks the lattice structure, it does not try to establish the inheritance if it isn't found.

**:check-sprites** Causes the description which receives the messages to determine if it can match any of the sprites it inherits.

**:install-sprite** The sprite which is the argument is installed in the description and is inherited to other relevant descriptions.

**:disseminate** Sets the disseminate status for the description to true. This is discussed further below.

**:free-vars** Determines the free variables in the description target.

**:subst-vars** Substitutes values for variables in the target description. The argument is an environment, a pairing of variables and values. A new description is returned with the proper substitutions if any substitutions are made.

**:match** Attempts a match with the argument description. Returns an environment that is created by the match if a match is possible.

**:gen-or-spec-descr** Generalizes or specializes a description. The argument specifies which to do. Generalization and specialization is discussed further below. A generalized or specialized description is returned.

## 7.3 Description Internment and Installation

Internment of a description is the process of finding the unique description object that corresponds to a list representation of the description. The first question to consider is "Why do internment at all?" An alternative is to create a new Flavor object for every list representation of a description presented to the system. The short answer to the question is that internment saves a lot of work that would otherwise have to be done. Consider the two descriptions:

```
(an Assignment-Proposal (with Goal goal-1)
                        (with Actor officer-3))

(an Assignment-Proposal (with Actor officer-3)
                        (with Goal goal-1))
```

By the axiom of commutativity of attributions these two descriptions are same. If an attempt is made to find all the descriptions that inherit from a description and there isn't a unique object for a description then a search must be made to locate all the description objects that are ***same***. Further problems arise, for example assume the following assertions and goals:

```
(assert (exclusive (an Assignment-Officer) (a Placement-Officer)))

(assert (is Smith (an Assignment-Officer)))

(show (lnot (is Smith (a Placement-Officer))))
```

A considerable amount of search is necessary in the case where multiple objects per description are allowed. A possible proof method that one would like to use is to determine if **Smith** inherits from anything that is exclusive with **(*a* Placement-officer)**, if so then it can be concluded that **Smith** does not inherit from **(*a* Placement-Officer)**. The problem is that all the **Smith** objects must be found, all the objects that inherit to any of the **Smith** objects must be found and then all the exclusions of those objects must be found and lastly **(*a* Placement-Officer)** must be tested for membership in the set of exclusions. This process leads to excessive search and messy algorithms.

The internment process consists of the following steps:

1. Normalize the list representation or ex-rep of the description. This involves ordering commutative elements.

2. Hash the ex-rep and look it up in a hash table. If there is an entry it is the desired description object.

3. If no entry exits then an object must be created. During creation of the description object the internment process is applied recursively to any constituent descriptions.

4. When the description and all its constituent descriptions are created the description object is installed in the description lattice.

### 7.3.1 Normalization of Descriptions

The first step of internment is to normalize the list representation of a description. The intent is to map a set of syntactic objects to a single description object. Normalization involves two steps:

1. Any description objects in the list representation of a description are replaced by their list representation

2. All commutative parts of a description are ordered.

The set of syntactic objects obtained by permuting commutative parts of a description form an equivalence class that is mapped to the description object. Note that a limitation to current list representation of descriptions is that circular structures of description objects are not possible. Thus there is no way for the description of an attribute of an instance description to be the instance description itself. This problem is circumvented by using the ***same*** relation and atomic descriptions.

### 7.3.2 Recursive Internment

In step 3 of the description of internment above it was mentioned that the internment process is applied recursively when a description needs to be created. This guarantees that any constituent descriptions of a description that don't exist will be created. The interned descriptions will be included as parts of the description being created. Thus there is sharing of structure between descriptions that have the same constituent parts. This saves space and is in line with the philosophy that there should be only one description object per equivalence class of syntactic descriptions. There is no problem with the sharing of descriptions since once created a description never changes. If a attribute value, for example, of an instance description is changed then a new description is created. As the description lattice is grown descriptions may acquire new objects that they inherit to or from but the parts of a description never change.

Another benefit of the recursive internment of description is that a description may be represented by any combination of list structure and description objects. This is often useful in Lisp code when one builds descriptions out of description objects and Lisp's list building functions. This requires that when a description representation is normalized so it can be looked up in the hash table that the constituent description objects be replaced with their list representation.

### 7.3.3 Description Installation

The process of *installing* a description entails linking the description to other descriptions in the description lattice via its **Inherit-To**, **Inherit-From**, and **Same** links. When a description is created the only information available to use for the installation process is the structure of the description itself and the description lattice. Thus the monotinicity axioms as described in section 5.4 are used.

The motivation behind installation is to establish inheritance links that are easy to deduce at installation time and that will save work later when the descriptions are used. Installation is a type of antecedent reasoning and as such involves tradeoffs. At one extreme a minimal amount of work could be done and descriptions could be installed under ***Something*** and over ***Nothing***. This would result in a flat database and would leave a lot of structuring work to be done during reasoning processes. The other extreme would be to do a lot of work searching for descriptions a particular description inherits from and to and setting up the links; the risk here is that a lot of work will be done that will never be used. The actual choice made is influenced by the Sprite invocation scheme described in the next section. Sprites use the inheritance network to discover potential descriptions that may match the Sprite's pattern. The approach taken is to install descriptions in such a way that Sprites will inherit to descriptions that they will potentially match. The actual installation scheme involves complicated issues involving Sprite invocation. The description of the installation scheme is delayed until after the description of Sprites in the next section.

## 7.4 Sprites

The implementation of Sprites in Omega builds on work on parallel problem solving methods by Kornfeld [Kornfeld 79]. Kornfeld describes the use of Sprites and Sponsors that avoid the bottlenecks and possible synchronization problems of previous pattern directed invocation system in a parallel environment. Kornfeld has developed a way implementing sprite using point-to-point communication in which some of the semantics of assertions can be embedded in specialized procedures. The extension to Kornfeld's work in this dissertation is to integrate Sprites into Omega's description lattice. In the following sections we will describe the structure of a Sprite, what conditions trigger Sprites and how Sprites are installed and inherited in the description lattice.

Sprites are implemented as Flavor objects. As shown below in figure 7-7 a Sprite object has 5 instance variables.

```
(Defflavor Sprite
 (Pattern        ;The Sprite's pattern
  Body           ;Function to execute when Sprite fires.
  Environment    ;The environment the Sprite will run in
  Justification  ;The Sprite's justification.
  Name)          ;The name of this Sprite.
  ())
```

**Figure 7-7:** The Structure of a Sprite

The **Pattern** is a description, usually containing variables, that is matched against descriptions in the description lattice. The **body** is a Lisp function that is executed in the **environment** augmented by any bindings as a result of the pattern match. The **justification** of a Sprite is a description which is interpreted to be the reason the sprite was created. Shown below is the syntax used for creating sprites.

```
(when <sprite-name> <sprite-pattern> <sprite-justification>
      <sprite-body>)
```

**Figure 7-8:** Defining a Sprite

When the above form is evaluated the following actions occur:

1. The function corresponding to the sprites body is defined.
2. The sprite object is created.
3. The sprite is installed in the description lattice.

One thing to note is that Sprites are always associated with a description. An interesting subject for future work is a unification of Sprites with descriptions. Thus sprites would be implemented by giving behaviors to descriptions.

One unusual feature of Omega's Sprites is that they are named, this a difference from most other pattern invocation systems such as PLANNER, [Hewitt 69], AMORD [de Kleer, Doyle, Steele, Sussman 77], ETHER [Kornfeld 79] etc. This was found useful for incrementally modifying the behavior of a system without having to restart the system. The problem is that if Sprites are not named then once a Sprite is installed there is no way to modify its behavior. New Sprites can be added but the old one is still there and will still fire when it matches a pattern. The only recourse is to restart the system from scratch. When Sprites are named they are accessible and their bodies may be modified. The change is not retroactive but will be in effect for any future executions of the sprite.

In previous systems, when an error was found in a pattern-invoked procedure the error was corrected, the knowledge base cleared, and the entire system restarted. This of course will not be possible with large knowledge based systems that reason about many activities at once. The naming of sprites is only an interim solution. What is desired is the ability to access sprites by description and to replace a sprite by switching to a new viewpoint in which the new sprite replaces the old sprite. This will be accomplished by using the sprites justification.

### 7.4.1 Pattern Matching

Sprite pattern matching is very simple; it is a structural match between the pattern and description. Only two descriptions are interpreted during the match--these are the **Var-Descr** (variable) and the **Which-Is-Descr** (qualified variable). The definition of matching is:

**Definition 7-1:** Two description match if and only if any of the following conditions hold:

1. They are the same description object or,

2. they are of the same type and their constituent parts match, or

3. a variable description (**Var-Descr**) that is not previously bound matches any description binding the description to the variable, or

4. a qualified variable (**Which-is-Descr**) that is not previously bound matches any description if the description inherits from the qualification, or

5. a variable description, qualified or not, that is previously bound only matches if the description being matched is the identical one the variable is bound to.

For example the following descriptions match producing the environment shown.

*The pattern:*
**(*a* ≡C (*with* Goal ≡G)**
**(*with* Performer Assignment-Officer-5))**
*Will match the description:*
**(*an* Assignment-Proposal-Step (*with* Goal Make-Proposal-32)**
**(*with* Performer Assignment-Officer-5))**
*Producing the environment:*

| | | |
|---|---|---|
| **≡C** | *bound to* | **Assignment-Proposal-Step** |
| **≡G** | *bound to* | **Make-Proposal-32** |

If a variable appears more than once in pattern then it will match a description only if the variable's corresponding descriptions are identical:

```
The description:
(an Order-Form (with unique Bill-To-Address ≡Add)
               (with unique Deliver-To-Address ≡Add)
Will match the description:
(an Order-Form (with unique Bill-To-Address Address-78)
               (with unique Deliver-To-Address Address-78))
Producing the environment:
    ≡Add              bound to          Address-78
```

A qualified variable matches a description if the description inherits from the qualification, for example:

```
The pattern:
(an Invoice (with unique Date (which is ≡D
                                 (a Date
                                    (with Earlier-Date 7/21/81)))))
Matches the description:
(an Invoice (with unique Date Date-17)
If
(is Date-17 (a Date (with Earlier-Date 7/21/81)))
Producing the environment:
≡D bound to          Date-17
```

Currently the the matching process does not attempt to prove that descriptions inherit from a qualifying description. Matching could easily do this and would provide an easy way for the matching process to establish goals in order to establish matches. The reasons that this was not done are first, as was mentioned at the beginning of the chapter the mechanisms of the implementation were kept as simple as possible, and second, there are efficiency considerations that may be a problem.

### 7.4.2 Installation of Sprites

The first question is "Where should Sprites be placed in the description lattice?" Intuitively the answer is to replace all variables in the sprite pattern with ***Something*** and install the Sprite at this point in the lattice. In the case of qualified variables, they would be replaced with their qualification. Thus a Sprite with a pattern of the form:

```
(an Officer (with unique Due-to-Roll-Date ≡Dt)
            (with unique Duty-Station
                           (which is ≡DS
                                 (a Duty-Station
                                  (with unique Location Pacific)))))
```

would be placed on the description:

**(*an* Officer (*with unique* Due-to-Roll-Date *Something*)**
**(*with unique* Duty-Station (*a* Duty-Station**
**(*with unique* Location Pacific))))**

This works fine since any description that could match the pattern of the sprite inherits from the description obtained by *generalizing* the pattern. Note that in the case that a variable occurs more than once in a Sprite pattern, this method of generalization will let the Sprite inherit to description that it will not match, eg, those descriptions where the places where the variable occurs don't all have the same description. There is a tradeoff between 1) the amount of effort that goes into finding a place for the Sprite to hang and insuring that the Sprite inherits to the descriptions that it can match and 2) between the efficiency considerations of making sure the Sprite is compared with as few descriptions that it won't match as possible. For example, it is functionally correct to place every Sprite on the most general description ***Something***. But this results in intolerable inefficiencies when the description lattice gets large.

The scheme for generalizing description patterns works fine until the patterns involve negations. Consider the following Sprite pattern:

**(*dnot* ≡Var)**

The Sprite corresponding to this pattern will be placed on the description:

**(*dnot Something*)** *Which is **same** with **Nothing***

The Sprite will inherit only to ***Nothing*** which is not right. In addition to generalization a *specialization* operation is needed. Thus the result of generalizing a description negation is that the negation will specialize its argument. In the example above the resulting description will be:

**(*dnot Nothing*)** *Which is **same** with **Something***

This same problem occurs in the case of inheritance with the ***is*** description. By the Monotinicity axiom for the ***is*** description in section 5.4 the proper generalization of the pattern:

**(*is* Invoice-16 ≡X)**

is the pattern

**(*is***
**Invoice-16 *Nothing*)**

## 7.4.3 Description-Based Sprite Invocation

The above approach has the weakness that it depends heavily on the inheritance hierarchy in conjunction with the monotinicity axioms. If a description is not installed correctly in the inheritance hierarchy then there is a chance that a sprite that would fire on the description will not be inherited to the description. Consider the following example:

```
(a Customer-Account (with unique Account-Number ≡A)
                    (with unique Balance-Due ≡B which is
                                            (an Amount
                                               (with Lesser-Amount $100.00))))
```

This pattern matches all customer accounts that have a balance due of more that $100.00. After generalizing the above pattern, the sprite will be installed on the following description:

```
(a Customer-Account (with unique Account-Number Something)
                    (with unique Balance-Due (an Amount
                                            (with Lesser-Amount $100.00))))
```

The problem is to insure that all descriptions that indeed inherit from this one are installed below it so they will inherit the sprite. In general this is difficult; it requires a great deal of effort when the a description is installed into the inheritance lattice to find all descriptions that the description being installed might inherit to and from. Note that this scheme is based on the syntactic structure of the description. Our solution is to take advantage of the semantic structure of the inheritance network as expressed by the inheritance links that are not a result of the monotinicity axioms and at the same time to give the user some control over where sprites are placed.

Giving the user control over where sprites are placed is important because automatic placement of sprites cannot work sufficiently well in all circumstances. Consider the following argument. Suppose a policy on sprite placement, in conjunction with a description installation policy, is chosen. The sprite is placed at some location in the hierarchy, say on description **d**. The description **d** must be fairly general because otherwise an inordinate amount of work would be required to install new descriptions in the inheritance hierarchy. Now suppose the hierarchy below **d** is highly developed, ie, many descriptions are installed below **d**. The sprite becomes a source of inefficiency because it must be compared with many descriptions that it won't match. The problem is that placement of sprites is not sensitive to how many descriptions there are below a possible point of sprite placement in the hierarchy. To add mechanisms that would automatically move sprites to more specific installation points when they are being compared with too many descriptions that they don't match is an excessively complicated approach.

As an alternative we give the user control over where sprites are to be placed. In this way the placement of sprites can be controlled in accordance with the semantics of the application. Suppose in the above example the user has described each account that is to be considered for retrieval as:

```
(an Account-Under-Consideration)
```

Extending the use of the ***which is*** description the pattern for the sprite would be:

```
(a Customer-Account (with unique Account-Number ≡A)
                    (with unique Balance-Due ≡B which is
                                              (an Amount
                                                 (with Lesser-Amount $100.00))))
which is
(an Account-Under-Consideration)
```

The sprite is then installed on the description **(*an* Account-Under-Consideration)** because this is what the outermost ***which is*** description generalizes to and inherits to a controlled number of descriptions that it might match. Note that in the above example when a Customer Account is described as under consideration the account balance would also be described as being over $100.00 if appropriate.

## 7.4.4 The Conditions for Firing of Sprites

Although a Sprite will inherit to all descriptions below the description it is attached to, the matching of a pattern is not sufficient to cause the Sprite to fire. The Sprite's pattern must match the description and the description must have been *Disseminated.* Within the implementation dissemination is accomplished by sending a disseminate message to the description. This facility is not available at the user interface. The only way for a user of the description system to cause a description to be disseminated is via the ***assert*** or ***show*** statements. The reason for this is explained fully in section 6.4 but intuitively the reason is because it is desirable to represent a statement in Omega without the mere existence of the statement implying that the statement is true.

## 7.4.5 The Efficiency of Description-Based Sprite Invocation

An advantage of semantic invocation of sprites is that the number of descriptions a Sprite must be compared against can be controlled. This situation can be contrasted to pattern directed invocations systems such as Micro-Planner [Sussman, Winograd, Charniak 70], CONNIVER [Sussman 72], AMORD [de Kleer, Doyle, Steele, Sussman 77] and others. Typically in these pattern directed invocation systems facts are stored in a large database which is implemented via some kind of discrimination network. When a new fact is added to the system it must be compared against all the active sprites; when a new sprite is added to the system, its pattern must be matched against every fact in the system. Adding a new fact is comparatively inexpensive since there are usually many more facts than sprites. However adding a new sprite is expensive; the time complexity of accessing the discrimination network to discover matching facts is O(log N) where N is the size of the database. The problem is that as the size of the database grows to infinity the ratio of the amount of time spent doing useful work to the time spent in the database accessing facts goes to 0. Thus, since practical applications very often involve large numbers of facts these system become impractical.

With semantic invocation of sprites accessing of facts for pattern matching does not involve an O(log N) operation. The number of descriptions involved in a pattern matching is not the entire database but only those descriptions to which a sprite inherits. If a disseminated description is added to the system it must be matched with the sprites that inherit to it. Thus adding a general description is less expensive that a more specific one because it inherits fewer sprites. If a sprite is added to the system then it must be matched with all disseminated descriptions it inherits to. This can be expensive if the sprite pattern is a general one or not so expensive, if a sprite pattern is more specific. In practice the more general sprites are added less often while specific sprites are added more often.

## 7.5 Resource Control - Sponsors

For resource control in Omega we have adopted the use of Sponsors as developed by Kornfeld [Kornfeld 79]. The description of Sponsors here is brief, for further information, the reader is referred to Kornfeld's work.[8] Sponsors control how much processing power a particular goal is given. When a Sponsor is created for a particular goal it is given a quanta of processing power--an amount of computational time that can be expended working on a goal. When the quanta is used the sponsor must ask for more quanta to continue work on a particular goal. The sponsor does this by making assertions as to how much quanta it has used.

[8]Kornfeld uses the term *activity* instead of the term *sponsor* in his work.

```
(Defflavor Sponsor
 ((Events Nil)            ; The events that are to run under this Sponsor.
  (State Quiescent)       ; The Sponsor's state.
  (Local-Proportion 0.)   ; Proportion of quanta for local node.
  (Local-Quanta 0.)       ; The current quanta allocated to this Sponsor.
  (Subtree-Quanta 0.)     ; The current quanta allocated to the remainder of subtree.
  (Sub-Sponsors NIL)      ; the sub-sponsors, pairs of proportions and sponsors.
  (Super-Sponsor NIL))    ; the super-sponsor.
  ())
```

Figure 7-9: The Structure of a Sponsor

The structure of a sponsor is shown above. A Sponsor has a list of events; the computational time incurred by the execution of these events is charged against the sponsor's quanta. There are three distinguished states that are interesting when discussing Sponsors:

1. Active - The sponsor has events that can be executed.

2. Quiescent - The sponsor has no events that it can execute. A sponsor leaves the quiescent state when it is either stifled or becomes active be receiving events to execute.

3. Stifled - The Sponsored activity has been halted. Once a sponsor is stifled it cannot be un-stifled; stifled is a terminal state. A sponsor is stifled when its goal is achieved or when it has been shown that its goal is unachievable.

Sponsors are organized into trees. A sub-sponsor is a sponsor that is working on a goal that is a prerequisite goal for its super-sponsor. When a sponsor is given a quanta it divides it up, (usually in equal parts) among itself and its sub-sponsors. If a sponsor cannot use all the quanta it is given, the sponsor returns the unused quanta to its super-sponsor to be divided and distributed. When a sponsor is stifled all sub-sponsors to that sponsor are also stifled.

## 7.6 The User Interface

There are various features of the Omega implementation that enable it to be used with relative ease on the Lisp Machine. The major problems are to map the representation of a description into its description object in the Lisp Machine environment and to make assertions, goals and sprites.

## 7.6.1 Internment of Descriptions from within Lisp

The way description objects are found is that the description type is defined as a function that interns the description. For example:

> Given the following lisp form for evaluation:
> **(A 'QUALIFIED-OFFICER)**
> A is a function which performs the following message transmission:

```
(SEND *SELECTED-OMEGA*
      ':INTERN-DESCR '(A QUALIFIED-OFFICER))
```

The function defined on the symbol A constructs the list representation of the instance description and sends this list to the description system (which is in the global variable ***SELECTED-OMEGA***) as the argument of an **:INTERN-DESCR** message. This is true with all the description types e.g. **WITH, LAND, DAND**, etc. A more complicated example is the following:

```
(A 'QUALIFIED-OFFICER (WITH 'BILLET (A 'BILLET)))
```

When this form is evaluated first the arguments will be evaluated. In this case the description object of **(A 'BILLET)** is retrieved as in the above example. Then the **WITH** function is executed with the arguments of **BILLET** and **(*a* Billet)**.[9] The first being a lisp symbol and the second a description object. The **WITH** function will send the following internment message:

```
(SEND *SELECTED-OMEGA*
      ':INTERN-DESCR (LIST 'WITH 'BILLET (a Billet)))
```

Note that the list contains both symbols and description objects. The normalization process will replace all description objects with their list representation and then look up the description object in a hash table using the list as a key. If the item exists it is returned otherwise it is created, put in the hash table and returned. Thus the A function will execute with the arguments **QUALIFIED-OFFICER** and the description object **(*with* Billet (*a* Billet))**. The A function will send the following message:

```
(SEND *SELECTED-OMEGA*
      ':INTERN-DESCR (LIST 'A 'QUALIFIED-OFFICER (with Billet (a Billet))))
```

Thus any of the elements of a description, e.g., concept, may be something that evaluates to either a description object or a list that represents a description. This is why the elements of a description

[9] In the following discussion description objects are written as **(*a* Billet)** as opposed to **(A 'BILLET)** which is the lisp form that returns the description object when it is evaluated.

need to be quoted. The internment process normalizes the representation of the description and finds or creates and returns the appropriate description object.

The above scheme works for most description types with the exception of atomic descriptions that are not part of a description (in the context of a description, a symbol is interpreted to be the representation of a description object). A distinction must be made between lisp symbols and atomic descriptions. This is done with a reader macro:

The form: **δJUSTIFICATION-6** expands to:
**(ATOMIC 'JUSTIFICATION-6)**

The symbol **ATOMIC** is defined as a function with sends the following internment message:

**(SEND *SELECTED-OMEGA* ':INTERN-ATOMIC-DESCR 'JUSTIFICATION-6)**

Note that through the use of the **ATOMIC** function the description system may describe objects in the lisp world. For example, a sponsor is a lisp object, suppose the symbol **TOP-SPONSOR** is bound to a sponsor then the form:

**(ATOMIC TOP-SPONSOR)**

Will return an atomic description object that has the sponsor stored in its ex-rep slot.

## 7.6.2 Making Assertions and Establishing Goals

Two lisp functions exist to make assertions and establish goals. To make an assertion the following function is used:

**(ASSERT <statement> <viewpoint> <justification-type>)**

When making assertions the statement and the viewpoint must be given, the justification type may optionally be given, if it is not given it defaults to **USER**. The above results in the creation and dissemination of a description as described in section 6.4. Goals are established in a similar manner:

**(SHOW <statement> <viewpoint> <sponsor> <justification-type>)**

When establishing goals the sponsor for the establishment of the goal is also included. The evaluation of the above form results in the creation and dissemination of a goal description as described in section 6.4.

## 7.6.3 Sprites for Assertions and Goals

Some sprite macros for matching assertions and goals also exist. These expand into the appropriate sprites. They allow the user to specify only the parts of the sprite that are of interest to the user: The first one is for matching against asserted statements and is of the form:

```
(WHEN-ASSERTED <sprite-name> <sprite-just>                ;Sprite Information
    <statement> <statement-just> <viewpoint>          ;Pattern Matched Elements
    <body>)
```

The statement, statement-just, and viewpoint elements are used in the pattern matching of the assertion description. The sprite-name and sprite-just are used in the creation of the sprite and controlling what viewpoints the sprite is known in. The sprites that match goals have a similar structure:

```
(WHEN-GOAL <sprite-name> <sprite-just>                    ;Sprint Information
    <goal-statement> <goal-just> <viewpoint> <sponsor>       ;Pattern Elements
    <body>)
```

As in the **WHEN-ASSERTED** sprite the goal-statement, goal-just, and viewpoint as well as the sponsor are used in the matching of goal description. The sprite-name and the sprite-just are used in the creation of the sprite and in controlling in what viewpoints the sprite is active.

## 7.6.4 Defining New Statement Types

Often it is necessary to define a statement in terms of other statements. For example the Individual predicate is an abbreviation for a complex statement. There are several macros that the user can use to define new statement types and thus new flavor objects. Suppose a user would like to define a new Omega statement **Procedure-Goal-Achieved.** This would be done by using the following form:

```
(Def-Unary-Predicate <Flavor-Name> <Descr-Keyword> <Print-String>)
```

The Descr-Keyword is the symbol used to define the function for retrieving the description's flavor object and the Print-String is used for pretty printing of the object. Thus in our example we would evaluate:

```
(Def-Unary-Predicate P-G-A-Stmt Procedure-Goal-Achieved "Procedure Goal Achieved")
```

Once this is done sprites are written that implement the behavior of the statement. For example sprites are written that implement antecedent and consequent reasoning in the cases that the predicate is asserted or is posted as a goal. In a similar manner binary relations, binary symmetric relations and multi-argument statements (such as ***land*** and ***lor***) can be defined.

# Chapter Eight

# Supporting Office Work

Omega provides a uniform framework within which to implement tools to support an office worker's problem solving. This has the benefit that different tools may cooperate easily in achieving the goals of particular office tasks. In this chapter we present some examples of how Omega can know about the work being performed and how Omega can help achieve the goals of the work.

Knowledge is embedded in the form of descriptions about objects in the system and the relationships between these objects. Some office systems have taken the stand that forms are the basic element of the system, an attempt is then made to represent everything in the system using forms. We view descriptions as the basic element of the system. Since the knowledgebase is represented using Omega's description lattice data does not have to be cast in a rigid form as it does in traditional data processing applications. The consequence is that office tasks may be reasoned about more on an individual basis.

Among some of the functions that traditional office forms provide that descriptions also provide are:

- **Storage of information** as in records.
- **Transfer of information** as in messages.
- **Display of information** in an abstracted and structured manner.
- **Accumulation and modification of information** as the form is used by individuals in the accomplishment of their tasks.

However, descriptions provide much greater functionality than an automated forms flow system. Descriptions are a very general facility; not only do they provide the functions that forms-based systems have as shown above but they also are the basis for Omega's reasoning machinery. Descriptions provide:

- **A means for error checking** of information in an office system.
- **A basis for retrieval** of stored information.

- A means by which the **structure of the application and organizational domains** of an office system are specified.

- **Viewpoints** by which change and inconsistent states may be reasoned about.

This chapter consists of detailed examples of the ideas that have been developed in earlier chapters. The main ideas we will address will be the following:

- **Problem Solving Support** - Use of the problem solving support paradigm in helping office workers in their tasks.

- **Goals** - The use of goals to describe office work. How these goal descriptions can help office workers in the performance of their tasks.

- **Organizational Structure** - The use of knowledge about an office worker's formal and informal relationships to his or her fellow workers.

- **Contradiction Handling** - examples of Omega's contradiction handling capabilities in dealing with real work knowledge.

## 8.1 Describing Office Work in Terms of Goals

As an example we show how part of the officer transfer process described in the introduction can be described in terms of goals and how this description can help an Assignment Officers in their work. The following description focuses on the internal mechanism that Omega uses to reason about a particular domain. We do not describe the user interface with which the user would make assertions or post goals or the way that Omega would present the results of its reasoning processes to the user. Our goal is to first get the underlying mechanism working right and then to work on a user interface for those mechanisms.

### 8.1.1 Posting a Goal

Shown below in figure 8-1 is the top level goal for a particular assignment proposal. The goal is to show that **OFFICER-6** and **BILLET-17** form a reasonable assignment proposal. This goal may have been posted by the assignment officer because he or she wanted to establish that the officer-billet pair formed a reasonable proposal or the goal may be part of a query that is trying to determine all the reasonable proposals for a particular group of officers and billets.

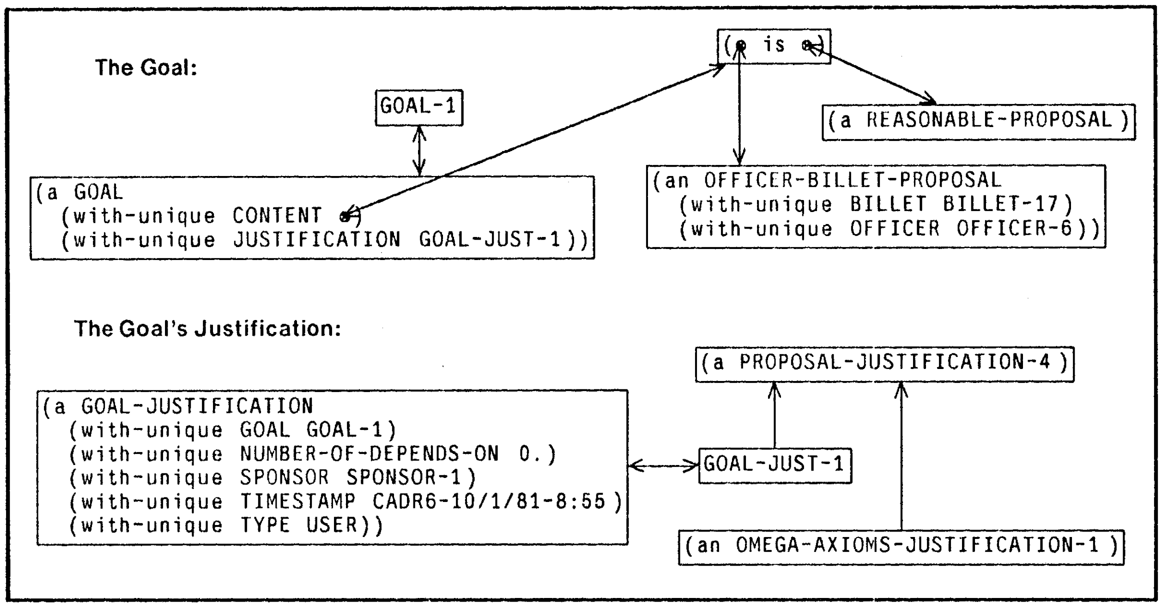

The Goal:
GOAL-1
( is )
(a REASONABLE-PROPOSAL )
(a GOAL
(with-unique CONTENT )
(with-unique JUSTIFICATION GOAL-JUST-1))
(an OFFICER-BILLET-PROPOSAL
(with-unique BILLET BILLET-17)
(with-unique OFFICER OFFICER-6))
The Goal's Justification:
(a PROPOSAL-JUSTIFICATION-4 )
(a GOAL-JUSTIFICATION
(with-unique GOAL GOAL-1)
(with-unique NUMBER-OF-DEPENDS-ON 0.)
(with-unique SPONSOR SPONSOR-1)
(with-unique TIMESTAMP CADR6-10/1/81-8:55 )
(with-unique TYPE USER))
GOAL-JUST-1
(an OMEGA-AXIOMS-JUSTIFICATION-1 )

**Figure 8-1:** The Assignment Proposal Goal

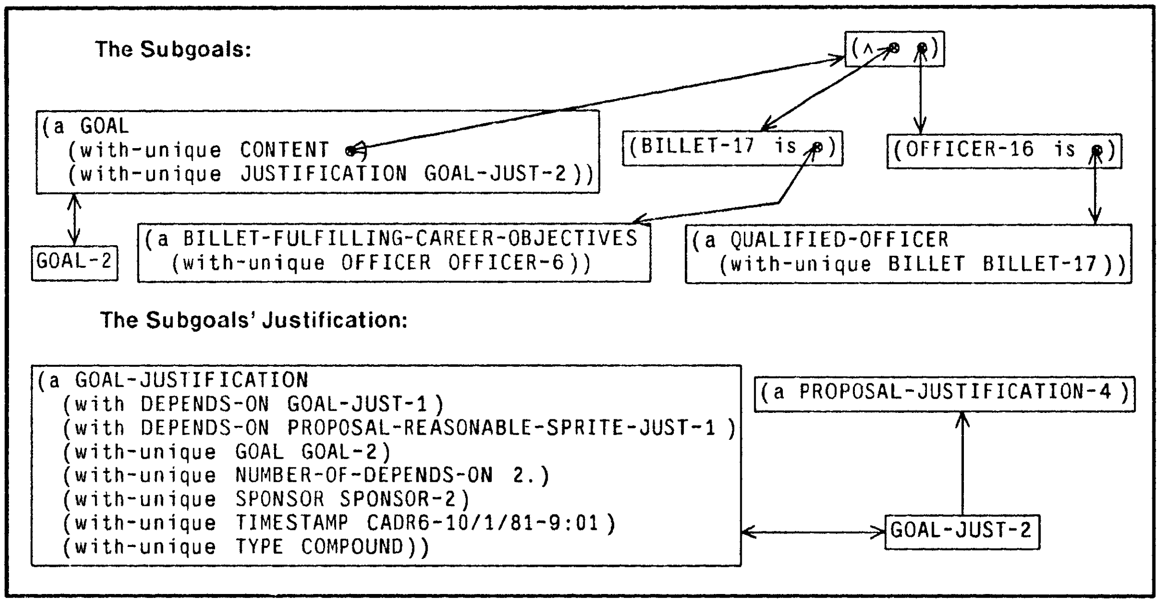
Now suppose that Omega has been told the following about what constitutes a reasonable proposal:

**(⇒ (∧ ≡B *is* (*a* Billet-Fulfilling-Career-Objectives**
**(*with unique* Officer ≡O))**
**≡O *is* (*a* Qualified-Officer**
**(*with unique* Billet ≡B)))**
**(*is* (*an* Officer-Billet-Proposal**
**(*with unique* Billet ≡B)**
**(*with unique* Officer ≡O))**
**(*a* Reasonable-Proposal)))**

This implication states that an Officer-Billet proposal is reasonable if the officer is a qualified officer for the particular billet and if the billet fits in with the officer's career objectives. If and when Omega decides that a particular assignment is reasonable is only according to the definition Omega has concerning what it takes to be a reasonable proposal. The above goal would be used as a filter to pick out the most obvious characteristics of the proposed assignment. The Assignment officer may look at a proposal that Omega has judged reasonable and reject it because of some criteria that Omega does not know about.

## 8.1.2 Posting of Subgoals

The assertion of the above rule creates several sprites,[10] one of which looks for a goal that matches the consequent of the implication. If the sprite fires it posts the antecedent of the implication as a goal. The sprite then creates a second sprite that watches to see if the antecedent is asserted. When the antecedent is asserted the second sprite fires and asserts the consequent. Thus as a result of the above implication and the goal in figure 8-1 the following subgoals will be posted.



The Subgoals:
(∧ ● ●)
(a GOAL (with-unique CONTENT ●) (with-unique JUSTIFICATION GOAL-JUST-2))
(BILLET-17 is ●)
(OFFICER-16 is ●)
GOAL-2
(a BILLET-FULFILLING-CAREER-OBJECTIVES (with-unique OFFICER OFFICER-6))
(a QUALIFIED-OFFICER (with-unique BILLET BILLET-17))
The Subgoals' Justification:
(a GOAL-JUSTIFICATION (with DEPENDS-ON GOAL-JUST-1) (with DEPENDS-ON PROPOSAL-REASONABLE-SPRITE-JUST-1) (with-unique GOAL GOAL-2) (with-unique NUMBER-OF-DEPENDS-ON 2.) (with-unique SPONSOR SPONSOR-2) (with-unique TIMESTAMP CADR6-10/1/81-9:01) (with-unique TYPE COMPOUND))
(a PROPOSAL-JUSTIFICATION-4)
GOAL-JUST-2

**Figure 8-2:** The Assignment Proposal Subgoals

Notice that the justification for this subgoal contains a new sponsor **SPONSOR-2**. The sprite that created the new subgoal also created a new sponsor for the processing that attempts to establish the subgoal. The reason for this is so that when the subgoal is achieved (or shown to be unachievable) the subgoal can be stifled without affecting the processing of the supergoal. In addition the sprite also linked the subgoal to the goal by setting up the following *is* relation:

**GOAL-JUST-1 *is* (*a* Goal-Justification (*with* Depended-on-by GOAL-JUST-2))**

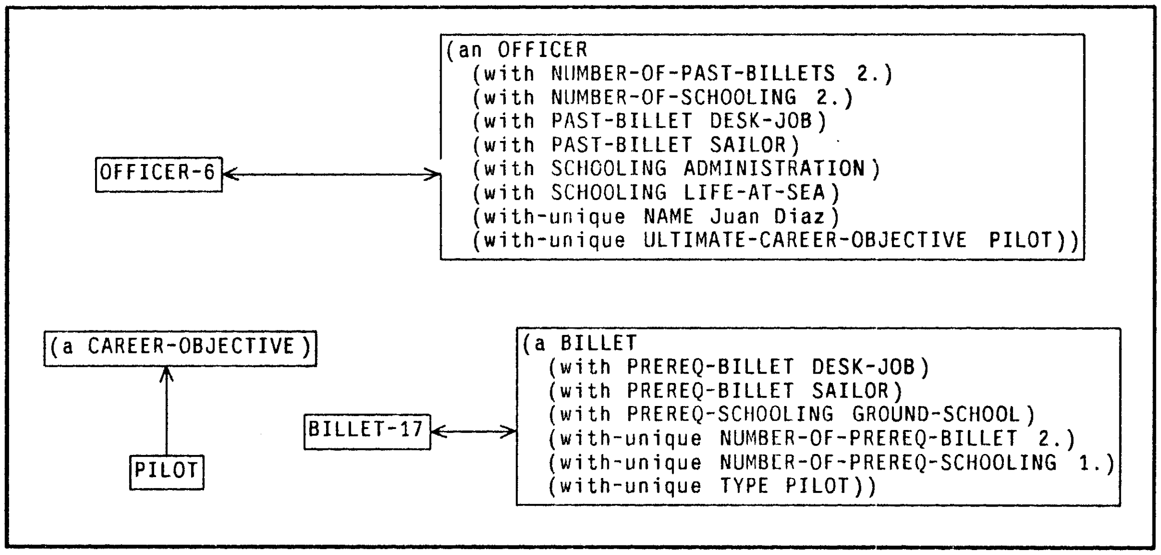
This enables analysis of the reasoning when, for example, a goal cannot be achieved or is shown to be

[10]A sprite watches for the assertion of an implication. When the sprite fires on such an assertion it creates 4 sprites corresponding to the 4 ways the implication can be used. These correspond to the antecedent and consequent reasoning of the implication and its contrapositive.

unachievable because of the failure of some subgoal. It is also useful when Omega explains how it has achieved a goal.

A conjunctive goal as in the diagram above is handled in the following fashion, a sprite will notice that there is a conjunctive goal. The sprite will fire and post the two conjuncts as goals. In addition the sprite will create additional sprites that watch for the assertion of each of the conjuncts or negation of either conjunct. When both conjuncts are asserted the conjunction is asserted; if either conjunct is negated the negation of the conjunction is asserted.

Suppose the following knowledge is stored in the description lattice with relevance to the goal shown above. Note that for brevity we do not include the assertions and justifications in this diagram, we just illustrate the *is* relations directly. The reader will note that the officer fulfills the billet prerequisites for past billets but not those for schooling. We describe how Omega discovers this.



```
(an OFFICER
  (with NUMBER-OF-PAST-BILLETS 2.)
  (with NUMBER-OF-SCHOOLING 2.)
  (with PAST-BILLET DESK-JOB)
  (with PAST-BILLET SAILOR)
  (with SCHOOLING ADMINISTRATION)
  (with SCHOOLING LIFE-AT-SEA)
  (with-unique NAME Juan Diaz)
  (with-unique ULTIMATE-CAREER-OBJECTIVE PILOT))
```

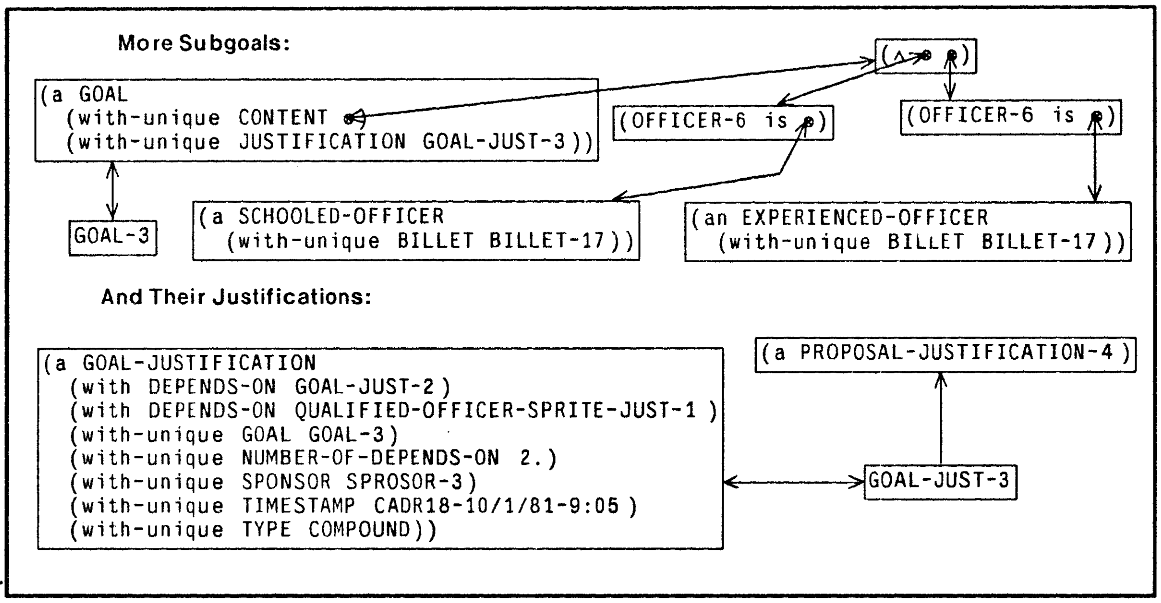
```
(a BILLET
  (with PREREQ-BILLET DESK-JOB)
  (with PREREQ-BILLET SAILOR)
  (with PREREQ-SCHOOLING GROUND-SCHOOL)
  (with-unique NUMBER-OF-PREREQ-BILLET 2.)
  (with-unique NUMBER-OF-PREREQ-SCHOOLING 1.)
  (with-unique TYPE PILOT))
```

**Figure 8-3:** Some Officer and Billet Knowledge

In this discussion we concern ourselves with how Omega shows that **OFFICER-16** is a qualified officer. The method used to show that **BILLET-16** fulfills the officer's career objectives follows in a similar manner. Omega has been given the following equivalence concerning qualified officers.

(⇔ (∧ ≡O *is* (*an* Experienced-Officer
(*with unique* Billet ≡B))
≡O *is* (*a* Schooled-Officer
(*with unique* Billet ≡B)))
(*is* ≡O (*a* Qualified-Officer
(*with unique* Billet ≡B))))

As in the previous implication, when this equivalence is asserted sprites are created that watch for goals that match the either the left or right halves of the equivalence. When a sprite fires after matching one half of the equivalence as a goal it posts the other half as a goal. In addition sprites are created that watch for the assertion or negation of either side of the equivalence. Thus when an assertion or negation of one side is made, the assertion or negation of the other side is made. Thus we have the following subgoals posted with a new sponsor:



More Subgoals:
(a GOAL (with-unique CONTENT ●) (with-unique JUSTIFICATION GOAL-JUST-3))
(∧ ● ●)
(OFFICER-6 is ●)
(OFFICER-6 is ●)
GOAL-3
(a SCHOOLED-OFFICER (with-unique BILLET BILLET-17))
(an EXPERIENCED-OFFICER (with-unique BILLET BILLET-17))
And Their Justifications:
(a GOAL-JUSTIFICATION (with DEPENDS-ON GOAL-JUST-2) (with DEPENDS-ON QUALIFIED-OFFICER-SPRITE-JUST-1) (with-unique GOAL GOAL-3) (with-unique NUMBER-OF-DEPENDS-ON 2.) (with-unique SPONSOR SPROSOR-3) (with-unique TIMESTAMP CADR18-10/1/81-9:05) (with-unique TYPE COMPOUND))
(a PROPOSAL-JUSTIFICATION-4)
GOAL-JUST-3

**Figure 8-4:** Subgoals to Establish Qualified Officer Status

Omega has been told the following concerning what it takes to be a Experienced Officer.

```
(⇔ (for all ≡P
          (⇒ ≡B is (a Billet
                          (with unique Prereq-Billet ≡P))
               ≡O is (an Officer
                          (with Past-Billet ≡P))))
     (is ≡O (an Experienced-Officer
               (with unique Billet ≡B))))
```

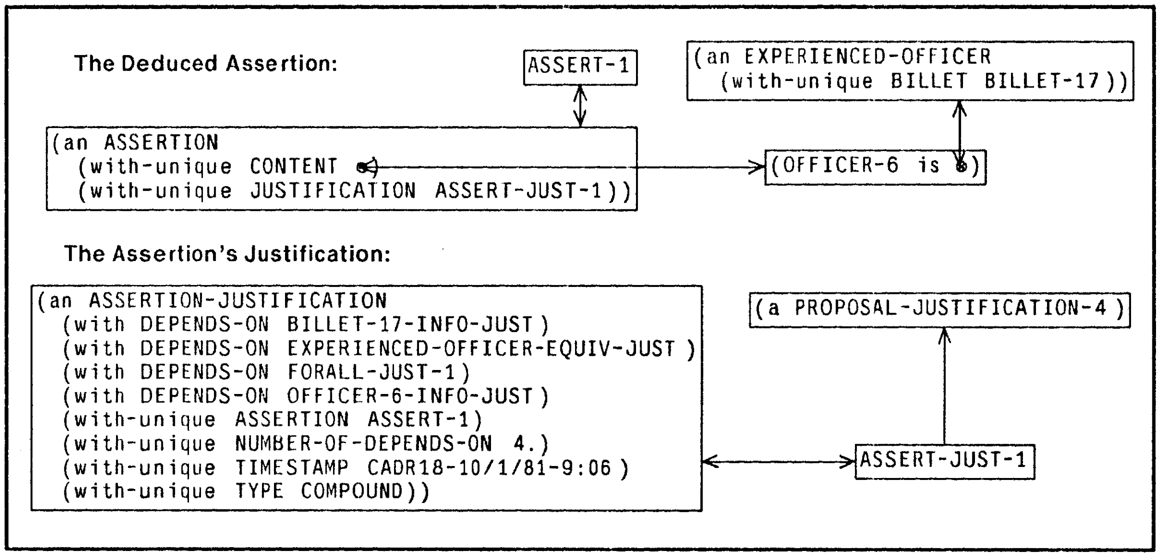
These goal are more difficult to achieve. This rule states that if it is true that for every prerequisite of a billet the prerequisite is a past billet for an officer then the officer is an experienced officer for the billet and vice-versa. As with the previous equivalence the right half of the statement will be posted as a goal when the left half is posted as a goal. Since this new goal involves a universal quantification some knowledge of the domain over which the variable ranges is necessary. This is the purpose of the NUMBER-OF-PREREQ-BILLETS and the NUMBER-OF-PAST-BILLETS attribute descriptions.

## 8.1.3 A Subgoal is Established

The method used to prove the universally quantified statement is to first retrieve the number of prereq billets and the number of past billets via sprites. The general approach is to insure that all the prerequisites are past billets; this means we must retrieve all the prerequisite billets. We know when we have retrieved them all by the NUMBER-OF-PREREQ-BILLETS number. Once they are all retrieved we check to see that each is a past billet. If each is a past billet then the universally quantified statement is asserted. If one prerequisite is not a past billet (which we can know since we know how many there are) then the negation of the statement is asserted. If there is not enough information to determine the truth or falsity of the statement the sprites remain waiting for additional information. Once the necessary information is known, if the sponsor of the sprites is still active, the statement or its negation will be asserted.

The reason that the NUMBER-OF-PAST-BILLETS attribute is necessary is so that Omega can know when to stop looking for billets. Without the number stated explicitly Omega cannot conclude that an officer has 2 past billets only because that is all the information that is stored explicitly in the description system. For example, it may be possible to prove the existence of more billets than are explicitly known about. Without explicitly stating the number of past billets the question on whether all billets are known or not is undecidable. This is an example of how Omega's goals of monotinicity and assimilation of new information affect the way Omega's reasoning processes.

In our example the sprites, using the information that appears in figure 8-3, will conclude that the officer is an experienced officer and will make the assertion shown below.



```
(an EXPERIENCED-OFFICER
  (with-unique BILLET BILLET-17))
```

```
(an ASSERTION
  (with-unique CONTENT •)
  (with-unique JUSTIFICATION ASSERT-JUST-1))
```

```
(OFFICER-6 is •)
```

```
(an ASSERTION-JUSTIFICATION
  (with DEPENDS-ON BILLET-17-INFO-JUST)
  (with DEPENDS-ON EXPERIENCED-OFFICER-EQUIV-JUST )
  (with DEPENDS-ON FORALL-JUST-1)
  (with DEPENDS-ON OFFICER-6-INFO-JUST)
  (with-unique ASSERTION ASSERT-1)
  (with-unique NUMBER-OF-DEPENDS-ON 4.)
  (with-unique TIMESTAMP CADR18-10/1/81-9:06 )
  (with-unique TYPE COMPOUND))
```

**Figure 8-5:** The Experienced Officer Assertion

Notice that this assertion depends on the information shown in figure 8-3 BILLET-17-INFO-JUST, OFFICER-6-INFO-JUST, on the justification for the universally quantified statement, FORALL-JUST-1, and on the justification for the equivalence statement EXPERIENCED-OFFICER-EQUIV-JUST. In particular it does not depend on any of the goals that were posted in the process of achieving the goal; as pointed out in [de Kleer, Doyle, Steele, Sussman 77] this would be a mistake since we do not want the truth or falsity of an assertion to depend on interest (as indicated by posted goals) in achieving the assertion.

Thus we have one of the conjuncts of figure 8-4 established.

### 8.1.4 A Subgoal is Refuted

The attempt to establishing the truth of the second conjunct follows in a similar manner. In this case the following rule is used to try to establish that an officer is a Schooled Officer for a particular billet:

```
(⇔ (for all ≡S
        (⇒ ≡B is (a Billet
                    (with unique Prereq-Schooling ≡S))
           ≡O is (an Officer
                    (with Schooling ≡S))))
   (is ≡O (a Schooled-Officer
             (with unique Billet ≡B))))
```

The difference is that in this case the outcome is the negation of the posted goal; Omega will assert that:

```
¬ (OFFICER-6 is (a Schooled-Officer
                    (with unique Billet BILLET-17)))
```

The failure to establish this fact implies the failure to establish the conjunctive goal in the rule on page 111 and hence the negation of the conjunction will be asserted which results in the negation of the second half of the equivalence:

```
¬ (OFFICER-6 is (a Qualified-Officer
                    (with unique Billet BILLET-17)))
```

We have been able to propagate back the fact that **OFFICER-6** was not a Schooled Officer because we had been using equivalences in our reasoning. When we get to our original implication, shown again below, we can go no further.

```
(⇒ (∧ ≡B is (a Billet-Fulfilling-Career-Objectives
                (with unique Officer ≡O))
      ≡O is (a Qualified-Officer
                (with unique Billet ≡B)))
   (is (an Officer-Billet-Proposal
          (with unique Billet ≡B)
          (with unique Officer ≡O))
       (a Reasonable-Proposal)))
```

This rule may be only one way that a proposal can be shown to be reasonable. There may be other rules that can possibly achieve the goal.

At this point the question is: how can we know when a goal cannot be achieved and how do we notify the user. One approach is the following. Suppose there are only 3 conditions under which a proposal may be judged reasonable. The the following rule could be used:

```
(⇔ (V r1 r2 r3)
    (is (an Officer-Billet-Proposal
            (with unique Billet ≡B)
            (with unique Officer ≡O))
        (a Reasonable-Proposal)))
```

Thus Omega can know that when all of **r1**, **r2**, and **r3** fail then the goal cannot be established. This approach has two undesirable consequences. First, if the Assignment Officer asserts that a particular proposal is reasonable then Omega can conclude that one of r1, r2, or r3 is true which in fact may not be the case. There may be some other criteria that the Assignment Officer has used to judge a proposal as reasonable. The second problem is what to do when another criteria for judging a proposal reasonable is to be told to Omega. This would mean that the above rule would have to be contradicted and a new viewpoint would have to be constructed with an new equivalence rule with 4 criteria for judging a proposal reasonable.

## 8.1.5 Using Sponsors to Reason About Reasoning

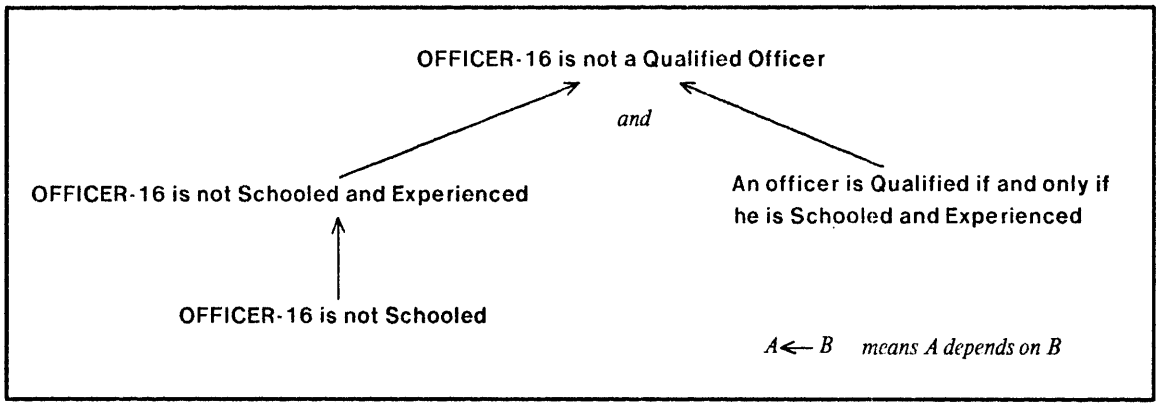
A superior approach is to use information concerning the sponsor of a particular goal. As was described in section 7.5, a sponsor is given a quanta with which to accomplish a goal. When the sponsor uses all its quanta it must ask for more to proceed. If a sponsor has quanta but can do no more work, i.e. it is quiescent, then it waits for additional work. The sponsor informs Omega about these events by making assertions. In our case the assertions will be simply the total quanta the sponsor has used. These assertions are made at two times, when the quanta allotted to the sponsor is exhausted or when the sponsor is quiescent.

Thus when a user posts a goal he or she will also specify the amount of quanta to be allocated to achieving the goal. When the quanta is used or no more of it can be used at a particular time then an assertion is made as to how much has been used. Note that if the assertion is made because the sponsor is quiescent at a particular time does not mean that no more can be used in the future. A new assertion, made from other sponsored activity, may once again enable work to be done on a particular goal. Thus in the case above when no more work can be done for a particular sponsor then the following is asserted.

```
Sponsor-1 is (a Quiescent-Sponsor
                 (with Exhausted-Quanta 4.3))
```

Note that this assertion is monotonically compatible with past assertions of this type. The assertion will trigger a sprite that was created at the time the sponsor was given its quanta for the particular goal. Again, at the time the sprite actually fires it may well be that the sponsor is no longer quiescent. The sprite may well check to see if the sponsor is quiescent or if the sponsor's goal has been established. If the sponsor isn't quiescent or the goal has been established the sprite may take no further action. If the sponsor is quiescent then it can examine the progress toward the goal. The progress toward the goal is analyzed by examining the **DEPENDED-ON-BY** attributes in the goal's justifications.

In this way Omega can determine what subgoals were posted for a goal and whether the goal or its negation was asserted. In our case it is determined that **OFFICER-6** was determined not to be a qualified officer. The following information can be extracted from Omega's descriptions.



OFFICER-16 is not a Qualified Officer
and
OFFICER-16 is not Schooled and Experienced
An officer is Qualified if and only if he is Schooled and Experienced
OFFICER-16 is not Schooled
A<— B means A depends on B

Figure 8-6: Why the Officer is not Qualified

Thus the user can see that the reason Omega has concluded that the officer is not qualified is because the officer is not Schooled. At this point the Assignment Officer may add the following assertion:

```
(⇒ (∧ ≡O is (an Experienced-Officer
                 (with unique Billet ≡B))
      ≡O is (a Schooled-or-Enrolled-Officer
                (with unique Billet ≡B))
      ≡B is (a Billet-Fulfilling-Career-Objectives
                (with unique Officer ≡O)))
    (is (an Officer-Billet-Proposal
          (with unique Billet ≡B)
          (with unique Officer ≡O))
        (a Reasonable-Proposal)))
```

This assertion says that if an officer is experienced, if the officer is Schooled or enrolled in school and if the billet satisfies the officer's career objectives then the proposal is a reasonable proposal. The officer would then go on to describe to Omega what it means for an officer to be schooled or enrolled in school for a particular billet.

## 8.2 Reasoning About Contradictions

In the previous section we have described how a user might interact with Omega when he or she is trying to achieve some goal and the goal cannot be achieved. The sponsors of a computation communicate with Omega and thus allowing Omega to reason in a limited but useful fashion about the progress in achieving a particular goal. In this section we describe how contradictions are handled when they arise in the course of achieving some goal. For example, contradictions can arise when a user makes an assumption that violates a system constraint.

In the following example we will continue the scenario from the previous section of this chapter. Now the Assignment Officer has judged a proposal as reasonable and must calculate travel expenses for the proposed reassignment. The contradiction will arise when the Assignment Officer assumes there is enough money in the current quarter's expense account to cover the reassignment. To begin the calculation of travel expenses the Assignment Officer posts the goal that the proposal be financially viable:

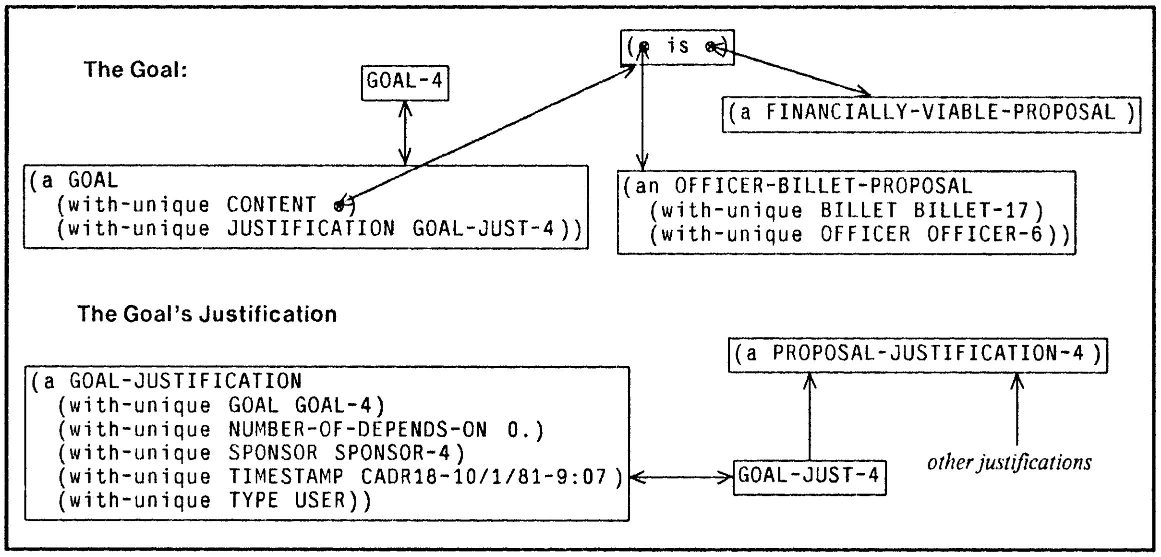

The Goal:
GOAL-4
(a GOAL
(with-unique CONTENT
(with-unique JUSTIFICATION GOAL-JUST-4))
( is )
(a FINANCIALLY-VIABLE-PROPOSAL )
(an OFFICER-BILLET-PROPOSAL
(with-unique BILLET BILLET-17)
(with-unique OFFICER OFFICER-6))
The Goal's Justification
(a GOAL-JUSTIFICATION
(with-unique GOAL GOAL-4)
(with-unique NUMBER-OF-DEPENDS-ON 0.)
(with-unique SPONSOR SPONSOR-4)
(with-unique TIMESTAMP CADR18-10/1/81-9:07 )
(with-unique TYPE USER))
(a PROPOSAL-JUSTIFICATION-4 )
GOAL-JUST-4
other justifications

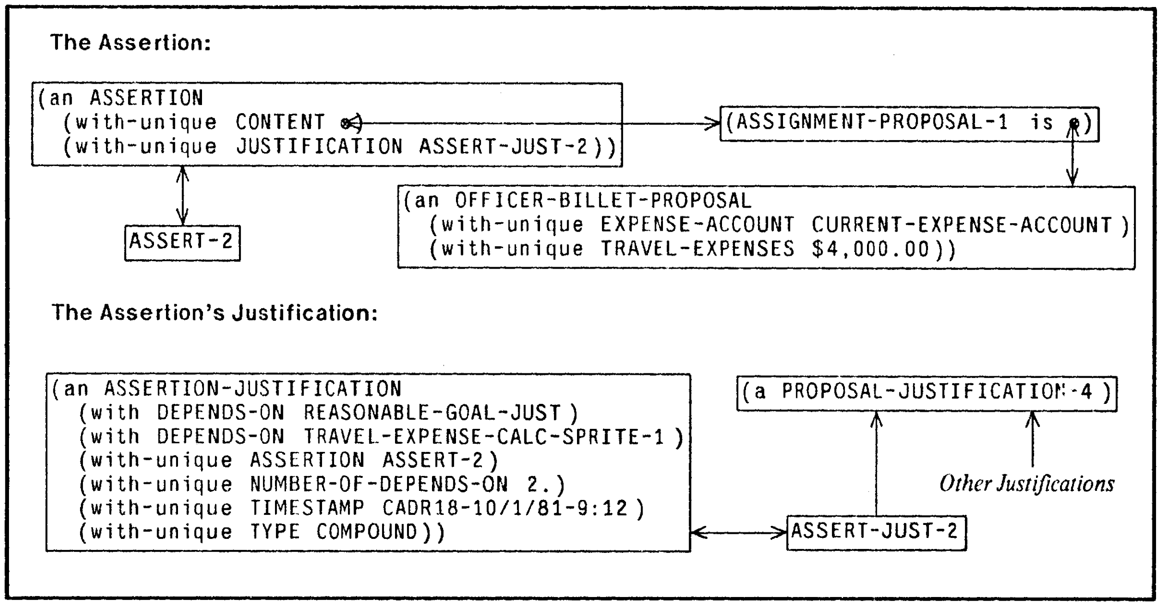
Figure 8-7: Representation of the Goal for Financial Viability

A sprite exists within Omega that watches for a goal of this sort. When the sprite fires on the goal it calculates the travel expenses for the proposed assignment and asserts this information. A abbreviated description of the sprite is shown below.

```
(when-goal Calc-Sprite-2 Travel-Expense-Sprite-Just-1          ;Name, Justification
           (is (an 'Officer-Billet-Proposal                       ;Goal to match
                   (with unique 'Officer ≡O)
                   (with unique 'Billet ≡B))
               (a 'Financially-Viable-Proposal))
           ≡G-JUST ≡VP ≡SPONSOR                                   ;Goal elements

           1. Calculate travel expenses,
           2. Use current expense account,
           3. Assert the travel expenses and expense account.)
```

The assertion the sprite makes with its justification is shown in the diagram below.


The Assertion:

```
(an ASSERTION
  (with-unique CONTENT ●)
  (with-unique JUSTIFICATION ASSERT-JUST-2))

(ASSIGNMENT-PROPOSAL-1 is ●)

ASSERT-2

(an OFFICER-BILLET-PROPOSAL
  (with-unique EXPENSE-ACCOUNT CURRENT-EXPENSE-ACCOUNT)
  (with-unique TRAVEL-EXPENSES $4,000.00))
```

The Assertion's Justification:

```
(an ASSERTION-JUSTIFICATION
  (with DEPENDS-ON REASONABLE-GOAL-JUST)
  (with DEPENDS-ON TRAVEL-EXPENSE-CALC-SPRITE-1)
  (with-unique ASSERTION ASSERT-2)
  (with-unique NUMBER-OF-DEPENDS-ON 2.)
  (with-unique TIMESTAMP CADR18-10/1/81-9:12)
  (with-unique TYPE COMPOUND))

(a PROPOSAL-JUSTIFICATION-4)

ASSERT-JUST-2
```

Figure 8-8: Travel Expense Assertion

The assignment proposal has been asserted to be *same* with the description **Assignment-Proposal-1** for brevity. The above assertion states that the assignment proposal will incur a cost of $4,000.00 from the current expense account for travel expenses. Now Omega uses the following rule to calculate the new balance on the expense account.

```
(⇒ (is ≡P
      (an Officer-Billet-Proposal
         (with unique Travel-Expenses ≡TE)
         (with unique Expense-Account
                  (an Expense-Account
                     (with unique Account-Number ≡AN)
                     (with unique Balance ≡B)))))
   (is (an Expense-Account (with unique Account-Number ≡AN))
      (an Expense-Account
         (with unique New-Balance (- ≡B ≡TE)))))[11]
```

Assume that the expense account has a balance of $1,000.00 and that the description of an expense account includes the following:

[11] Note that we have used the abbreviation (- A B) for the description (*a* Difference (*of* Minuend A) (*of* Subtrahend B))

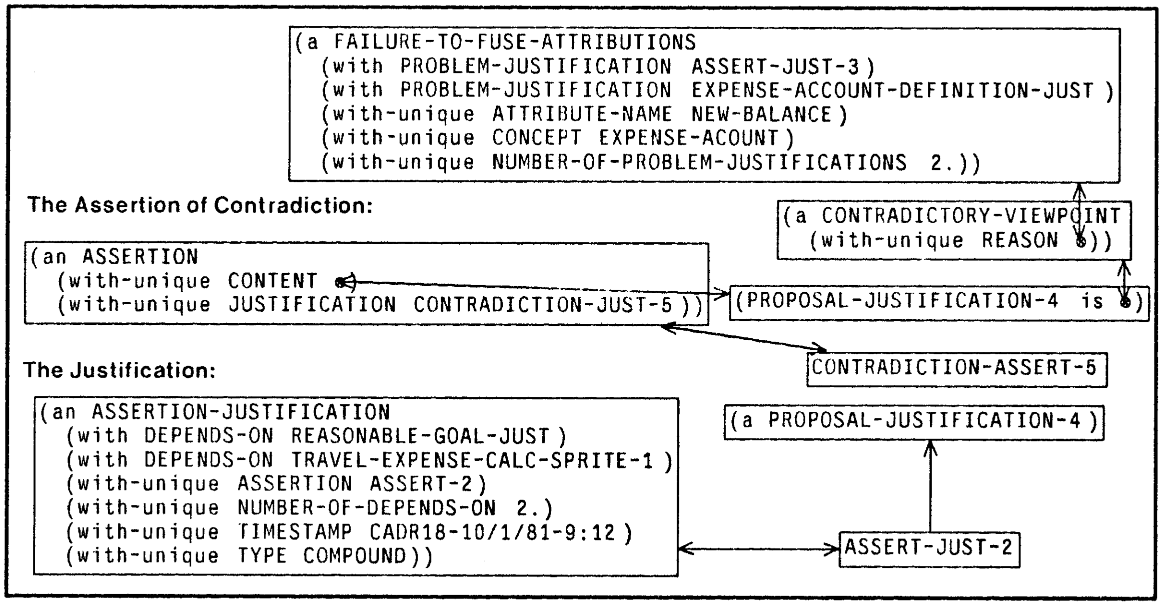
(*an* Expense-Account) *is* (*an* Expense-Account
(*with every* Balance (>= 0))
(*with every* New-Balance (>= 0)))

Where here we use the abbreviation (>= 0) for the description

(*a* Dollar-Amount (*with* Lesser-or-Equal-Amount 0))

.

When the rule that calculates the new balance fires it will assert that the new balance is $-3,000.00. This will be fused with the constraint that every balance and new balance be greater than or equal to 0. The attempt to fuse will fail, signalling a contradiction by making the following assertion:



```
(a FAILURE-TO-FUSE-ATTRIBUTIONS
  (with PROBLEM-JUSTIFICATION ASSERT-JUST-3)
  (with PROBLEM-JUSTIFICATION EXPENSE-ACCOUNT-DEFINITION-JUST)
  (with-unique ATTRIBUTE-NAME NEW-BALANCE)
  (with-unique CONCEPT EXPENSE-ACOUNT)
  (with-unique NUMBER-OF-PROBLEM-JUSTIFICATIONS 2.))

(a CONTRADICTORY-VIEWPOINT
  (with-unique REASON *))

(an ASSERTION
  (with-unique CONTENT *)
  (with-unique JUSTIFICATION CONTRADICTION-JUST-5))

(PROPOSAL-JUSTIFICATION-4 is *)

CONTRADICTION-ASSERT-5

(an ASSERTION-JUSTIFICATION
  (with DEPENDS-ON REASONABLE-GOAL-JUST)
  (with DEPENDS-ON TRAVEL-EXPENSE-CALC-SPRITE-1)
  (with-unique ASSERTION ASSERT-2)
  (with-unique NUMBER-OF-DEPENDS-ON 2.)
  (with-unique TIMESTAMP CADR18-10/1/81-9:12)
  (with-unique TYPE COMPOUND))

(a PROPOSAL-JUSTIFICATION-4)

ASSERT-JUST-2
```

**Figure 8-9:** Assertion of the Contradiction

In the above, we have assumed that the assertion which calculated the **NEW-BALANCE** has the justification **ASSERT-JUST-3**. A sprite will fire when the contradiction is asserted. The sprite will retrieve the justifications for the offending assertions. The sprite will analyze the assertions, retrieving the descriptions in the **NEW-BALANCE** attributions derive the following dependencies.

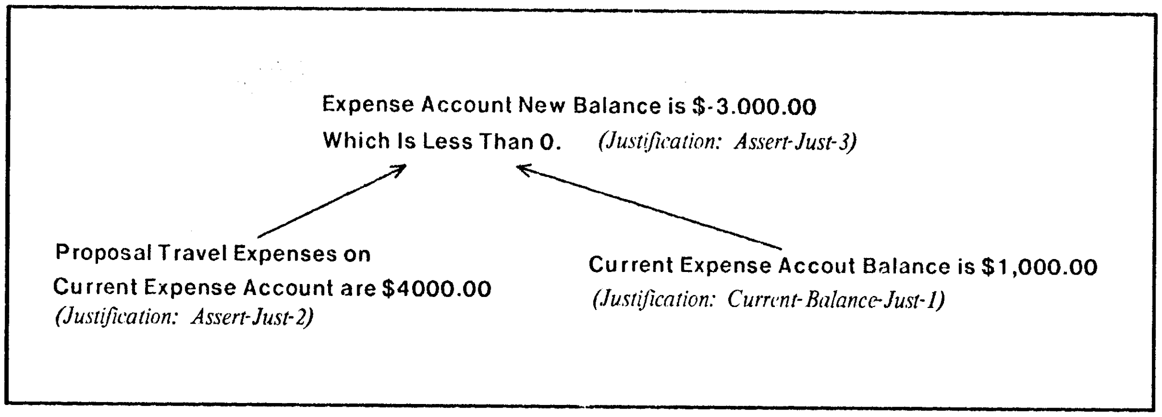

Expense Account New Balance is $-3.000.00
Which Is Less Than 0. (Justification: Assert-Just-3)
Proposal Travel Expenses on
Current Expense Account are $4000.00
(Justification: Assert-Just-2)
Current Expense Accout Balance is $1,000.00
(Justification: Current-Balance-Just-1)

**Figure 8-10:** Contradiction Dependencies

The possible actions at this point are various:

- The information could be presented to the user allowing him or her to make a decision about how to proceed.

- The problem, if it is frequent, could be recovered from automatically by using, the next quarter's account or some slush fund that exists for this purpose.

- The constraint the the new balance be non-negative could be retracted, perhaps notifying the user. In this way the user could proceed with the work and leave address the travel funds issue latter.

In other situations more variations arise. If the contradiction arises from contradictory assertions made by different sources then the reliability of the sources or the authority of the sources might be compared in order to decide which assertion to accept and which to reject.

## 8.3 Summary

We have presented Omega's viewpoint mechanism along with some examples of its use to describe change in an office. The viewpoint mechanism is useful for describing objects whose properties vary with time as well as a means to handle contradictions that arise during reasoning processes.

The viewpoint mechanism presented here is related to that in ETHER [Kornfeld 79] and to the layers of the PIE system [Goldstein 80]. Viewpoints are a powerful unifying mechanism which combine aspects of McCarthy's situational tags [McCarthy, Hayes 69] and the contexts of QA4 [Rulifson, Derksen, Waldinger 72]. They serve as a replacement for update and pusher-puller mechanisms.

# Chapter Nine

# Conclusion

This dissertation presents the beginnings of a theory of organizational behavior. We have pointed out the need for a theory that includes both organizational knowledge and application knowledge. The application knowledge is important because it is the explicit reason for the existence of the organization. The organizational knowledge is important because it describes how people function together to elicit the overall behavior of the organization. This dissertation has focused on a characterization of office work as a problem solving behavior and on the development of the description machinery necessary to describe activity in the organization (Omega and Viewpoints).

Our motivation for studying organizations arises from two concerns. The first is that new technology is being used in a growing number of ways in today's organizations but the question of how to best use that technology is not easily answered. Indeed, the use of technology such as word processors in typing pools has produced unforeseen and in some cases undesirable consequences. Our contention is that this reflects a basic lack of understanding of the organizational. The second concern is that organizations represent a good domain in which to develop and test new AI theories.

In section 3.2 on Office Semantics we presented several goals for Office Semantics. We have only begun to satisfy these goals in the limited scope of this work. Below are listed the goals, how this dissertation has contributed toward their achievement and directions for future work.

**Describing Organizational Behavior** - We have presented the description system Omega for describing organizational behavior. In the body of this dissertation are presented several examples of the use of Omega in describing organizational activity. The examples represent only a beginning in describing organizational activity. A direction for future research would be to use Omega in conjunction with a analysis methodology such as that described in [Sirbu, Schoichet, Kunin, Hammer 81] to perform field studies with the purpose of describing and analyzing actual organizational situations.

**Explanation of Organizational Structure** - This also has not been fully developed here but we feel that 3 ideas that we have developed will contribute significantly to an explanation of organizational behavior. These are 1) including the informal and formal social structure as well as the application structure in understanding the behavior of organizations, 2) the view of organizational work as being composed of problem solving and algorithmic tasks, 3) the description of organization work in terms of goals. The development of these three ideas into a coherent explanation of organizational structure is a promising direction for future research.

**Organizational Predictability** - The ability to predict the effects of change on organizations awaits further development of Office Semantics. Once more understanding of organizational structure is achieved predictions of the effects of organizational change can be done with more confidence.

**Character of Organizational Work** - We have described organizational work as goal oriented and thus requiring problem solving capabilities on the part of the organizational worker. Office work is characterized as a mix of problem solving and algorithmic tasks. Algorithmic tasks are those that can be automated while problem solving tasks are those that are performed by the office worker. We believe that technology is best used in supporting organizational workers in their problem solving rather than trying to automate the workers' tasks. We believe that problem solving in organizational work is an essential element of that work.

**Role of Technology** - We have developed the idea that the role of technology is to support organizational workers in their problem solving and algorithmic tasks. Much work remains to be done in developing distributed workstation systems based on the ideas presented here. We have addressed the issue of the underlying reasoning machinery which we believe is directly related to the organization's structure and function. These ideas must be implemented in hardware and software systems that can support organizational work.

This dissertation has emphasized the development of a knowledge embedding system that can be used to reason about organizational and application area knowledge. To this end we have developed Omega's Viewpoint mechanism. The Viewpoint mechanism contributes to the state of the art in reasoning systems by allowing a a system to reason about contradictions and, to a limited extent, to reason about the reasoning process itself.

In addition we have developed the idea of a description based sprite invocation system. This allows the designer of a knowledge base to control the scope of a sprite in terms of the semantics of the application and thus control the inefficiency of pattern directed invocation.

Many of the ideas relating to Omega and the Viewpoint mechanism in this dissertation offer opportunities for further developed. As was mentioned previously the work on Office Semantics is just a beginning. A great deal of work remains in the areas of studying and describing the structure of organizations. We have done little in developing the informal and formal social structure of organizations save to point out that it is a prerequisite to a successful theory of organizational behavior. With respect to Omega there are several areas of future research.

A **Distributed Omega** description language that will function in the presence of many workstations is necessary for the full utilization of Omega's potential in an organizational setting.

A **Graphic User Interface** to allow a user to interact with Omega in a graphical and intuitive fashion. This work has begun with Ciccarelli's Presenter for Omega but much remains to be done. The ability to integrate diverse sources of knowledge into Omega's knowledge base is needed. Omega must be able to manipulate and generate speech, color graphics and to use pointing devices other than the mouse.

Development of **Description Directed Invocation** facilities for controlling sprite inheritance. The basic mechanism is available for controlling the inheritance of sprites, the problem is that it is hard to use. For each application the knowledge base designer must explicitly control the inheritance of sprites. More automatic methods are needed that don't need to be custom tailored to each application.

Facilities to **Control Reasoning** which are more powerful than those presented here are possible. Facilities for detecting confusion, lack of knowledge, and inappropriateness of knowledge would be extremely useful in reasoning about real work situations.

A **Formal Semantics** for Omega with Viewpoints is needed beyond the consistency proof for a subset of Omega given in [Attardi, Simi 81]. It is necessary that the Omega system be shown to be consistent. If this is not the case then a user cannot have confidence in any conclusions the system derives.

This dissertation has addressed organizational issues and AI issues. Many of these issues are related and it is the author's conclusion that these two domains are natural companions for study. Not only does insight into an issue in one domain often arise from considering the issue in the other but many new ideas arise from the consideration of the two fields together.

# Chapter Ten

# References

[Attardi, Barber, Simi 80]
Attardi, G., Barber, G. and Simi, M.
Towards an Integrated Office Work Station.
*Automazione e Strumentazione* (3), March, 1980.

[Attardi, Simi 81]
Attardi, G. and Simi, M.
Semantics of Inheritance and Attributions in the Description System Omega.
In *Proceedings of IJCAI 81*. IJCAI, Vancouver, B. C., Canada, August, 1981.

[Bailey, Gerlach, McAfee, Whinston 81]
Bailey, A. D., Gerlach, J., McAfee, R. P., Whinston, A. B.
Internal Accounting Controls in the Office of the Future.
*Computer* 14(5), May, 1981.

[Barber, Hewitt 80]
Barber, G. R. and Hewitt, C. E.
Research in Office Semantics.
In *Expert Systems: State of the Art*, pages 4/3-4/12. Infotech, April, 1980.

[Barber, Hewitt 81a]
Barber, G. R. and Hewitt, C. E.
Research in Workstation Network Semantics.
*IEEE Transactions on Software Engineering* , (forthcoming), 1981.

[Barber, Hewitt 81b]
Barber, G. R. and Hewitt, C. E.
Research in Office Semantics.
In Michie, D., editor, *Introductory Readings in Expert Systems*. Gordon and Breach, London, 1981.

[Barber 80]
Barber, G. R.
Reasoning About Change in Knowledgeable Office Systems.
In *First Annual National Conference on Artificial Intelligence*. AAAI, August, 1980.

[Barber 81a]
Barber, G. R.
Embedding Knowledge in a Workstation.
In *Proceedings of the INRIA International Workshop on Office Information Systems.* October, 1981.

[Barber 81b]
Barber, G. R.
*Record of the Workshop on Research in Office Semantics.*
AI Memo 602, MIT, March, 1981.

[Bobrow, Winograd 77]
Bobrow D. G. and Winograd, T.
An Overview of KRL-0, a Knowledge Representation Language.
*Cognitive Science* 1(1), 1977.

[Borning 77]
Borning A.
ThingLab -- An Object-Oriented System for Building Simulations Using Constraints.
In *Proceedings of IJCAI-77.* IJCAI, August, 1977.

[Browner, Chibnik. Crawley, Newman, Sonafrank 79]
Browner, Chibnik, Crawley, Newman, and Sonafrank.
*Report on a Summer Research Project: A Behavioral View of Office Work.*
Technical Report, Xerox PARC, 1979.

[Buneman, Morgan, Zisman 77]
Buneman, O. , H. Morgan and M. Zisman.
Display Facilities for Decision Support.
*Data Base*, Winter, 1977.

[Carmody, Gross, Nelson, Rice, Van Dam 69]
Carmody, S., Gross, W., Nelson, T. H., Rice, D., Van Dam, A.
*Pertinent Concepts in Computer Graphics.*
University of Illinois, 1969, chapter A Hypertext Editing System for the S/360.

[Dahl, Myhrhaug, Nygaard 70]
Dahl O. J., Myhrhaug B., and Nygaard K.
*Simula Common Base Language.*
Technical Report S-22, Norwegian Computing Center, October, 1970.

[Davis 76]
Davis, R.
*Applications of Meta Level Knowledge to the Construction, Maintainance and Use of Large Knowledge bases.*
Memo AIM-283, Stanford AI Lab, July, 1976.

[Davis 80]
Davis, R.
*Meta-Rules: Reasoning About Control.*
Memo 576, MIT AI Lan, March, 1980.

[de Jong, Zloof 77]
de Jong S. P. and Zloof M. M.
The System for Business Automation (SBA): Programming Language.
*Communications of the ACM* 20(6), June, 1977.

[de Jong 80]
de Jong S. P.
SBA: A System for Office Automation.
In *Proceedings of the 1980 IFIP Congress.* Tokyo, 1980.

[de Kleer, Doyle, Steele, Sussman 77]
de Kleer, J., Doyle, J., Steele, G. L., and Sussman, G. J.
AMORD: Explicit Control of Reasoning.
In *Proceedings of the Symposium on Artificial Intellignece and Programming Languagues.* SIGART, Rochester, N.Y., August, 1977.

[Deliyanni, Kowalski 79]
Deliyanni, A., Kowalski, R. A.
Logic and Semantic Networks.
*Communications of the ACM* 22(3), March, 1979.

[Doyle 77]
Doyle, J.
*Truth Maintenance Systems for Problem Solving.*
AI-TR 419, MIT AI Lab, May, 1977.

[Ellis, Nutt 80]
Ellis, C. A. and Nutt, G. J.
Computer Science and Office information Systems.
*Computing Surveys* 12(1), March, 1980.

[Ellis 79]
Ellis, C. A.
Information Control Nets: A Mathematical Model of Office Information Flow.
In *Proceedings of the Conference on Simulation, Modeling and Measurement of Computer Systems*, pages 225-240. ACM, August, 1979.

[Engel, Groppuso, Lowenstein, Traub 79]
Engel G. H., Groppuso J., Lowenstein R. A., and Traub W. G.
An Office Communications System.
*IBM Systems Journal* 18(3), 1979.

[Fahlman 77]
Fahlman, S. E.
*NETL: A System for Representing and Using Real-World Knowledge.*
MIT Press, Cambridge, Mass., 1977.

[Fein 74]
Fein, M.
Job Enrichment: A Reevaluation.
*Sloan Management Review*, Winter, 1974.

[Fikes, Henderson 80]
Fikes, R. E. and Henderson, D. A.
On Supporting the Use of Procedures in Office Work.
In *Proceedings of the First Annual AAAI Conference.* American Association for Artificial Intelligence, August, 1980.

[Fikes 80a]
Fikes, R. E.
Odyssey: A Knowledge-Based Assistant.
To appear in Artificial Intelligence.

[Fikes 80b]
Fikes, R. E.
On Supporting the Use of Procedures in Office Work.
Presentation at the Workshop on Research in Office Semantics, Chatham Bar, Massachusetts.

[Garfinkel, Sacks 70]
Garfinkel, H. and H. Sacks.
On Formal Structures of Practical Actions.
In J. Mckinney and E. Tiryakian, editors, *Theoretical Sociology*, chapter 13, pages 337-366. Appleton-Century-Crofts, 1970.

[Goldstein, Roberts 77]
Goldstein, I. P. and Roberts, R.B.
NUDGE, a Knowledge-Based Scheduling Program.
*Proceedings of the Fifth International Joint Conference on Artificial Intelligence.*

[Goldstein 80]
Goldstein, Ira.
PIE: A Network-Based Personal Information Environment.
In *Proceedings of the First Annual AAAI Conference.* American Association for Artificial Intelligence, August, 1980.

[Gorry, Scott Morton 71]
Gorry, G. A., and Scott Morton M. S.
A Framework for Management Information Systems.
*Sloan Management Review*, Fall, 1971.

[Hammer, Howe, Kruskal, Wladawsky 77]
Hammer, M., Howe, G., Kruskal V., and Wladawsky I.
A Very High Level Programming Language for Data Processing Applications.
*Communications of the ACM* , November, 1977.

[Hewitt, Attardi, Lieberman 79]
Hewitt C., Attardi G., and Lieberman H.
Specifying and Proving Properties of Guardians for Distributed Systems.
In *Proceedings of the Conference on Semantics of Concurrent Computation.* INRIA, Evian, France, July, 1979.

[Hewitt, Attardi, Simi 80]
Hewitt, C., Attardi, G., and Simi, M.
*Knowledge Embedding with a Description System.*
AI Memo, MIT, August, 1980.

[Hewitt 69]
Hewitt C. E.
PLANNER: A Language for Proving Theorems in Robots.
In *Proceedings of IJCAI-69.* IJCAI, Washington D. C., May, 1969.

[Hewitt 79]
Hewitt C. E.
*Design of the APIARY for VLSI Support of Knowledge-Based Systems.*
AI Lab Working Paper 186, MIT, May, 1979.

[Hiltz, Turoff 78]
Hiltz, S. R. and Turoff, M.
*The Network Nation, Human Communication via Computer.*
Addison-Wesley Publishing Company, 1978.

[Ingalls 78]
Ingalls D.
The Smalltalk-76 Programming System, Design and Implementation.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages.* ACM, Tucson, Arizona, January, 1978.

[Kahn 81]
Kahn, K.
*Uniform--A Language Based Upon Unification which Unifies (much of) Lisp, Prolog, and Act 1.*
Technical Report, University of Uppsala, March, 1981.

[Katz, Kahn 78]
Katz, D. and Kahn, R.
*The Social Psychology of Organizations.*
John Wiley and Sons, 1978.

[Kornfeld, Hewitt 81]
Kornfeld, W. A. and Hewitt, C.
The Scientific Community Metaphor.
*IEEE Transactions on Systems, Man, and Cybernetics* SMC-11(1), January, 1981.

[Kornfeld 79]
Kornfeld, W.
*Using Parallel Processing for Problem Solving.*
AI Memo 561, MIT, December, 1979.

[Kornfeld 82]
Kornfeld, W.
*Concepts in Parallel Problem Solving.*
PhD thesis, Massachusetts Institute of Technology, 1982.

[Lieberman, Hewitt 80]
Lieberman, H. and Hewitt C. E.
*A Session with TINKER.*
AI Memo 577, MIT, April, 1980.

[Liskov, Snyder, Atkinson, Chaffert 77]
Liskov B., Snyder A., Atkinson R., and Chaffert C.
Abstraction Mechanism in CLU.
*Communications of the ACM* 20(8), August, 1977.

[March, Simon 63]
March J. G. and Simon H. A.
*Organizations.*
John Wiley & Sons, Inc., 1963.

[McCarthy, Hayes 69]
McCarthy, J. and Hayes, P. J.
Some Philosophical Problems from the Standpoint of Artificial Intelligence.
In *Machine Intelligence 4*, pages 463-502. Edinburgh University Press, 1969.

[McDermott, Doyle 79]
McDermott, D. and Doyle, J.
*Non-Monotonic Logic I.*
AI Memo 486b, MIT, July, 1979.

[Metcalfe, Boggs 76]
Metcalfe, R. M. and Boggs, D. R.
Ethernet: Distributed Packet Switching for Local Computer Networks.
*Communications of the ACM* 19(7), July, 1976.

[Minsky 77]
Minsky, M.
Plain Talk about Neurodevelopmental Epistemology.
In *Proceedings of IJCAI 5.* IJCAI, Cambridge, Mass., August, 1977.

[Naffah 81]
Naffah, N.
Distributed Office Systems in Practice.
In *Local Networks and Distributed Office Systems.* London, 1981.

[Newell, Simon 72]
Newell, A. and Simon, H. A.
*Human Problem Solving.*
Prentice Hall, Englewood Cliffs, N.J., 1972.

[Nutt, Ricci 81]
Nutt, G. and Ricci, P. A.
Quinault: An Office Modeling System.
*Computer*, May, 1981.

[Quine 63]
Quine, W. V. O.
*From a Logical Point of View.*
Harper & Row, 1963.

[Rulifson, Derksen, Waldinger 72]
Rulifson, J., Derksen, J. and Waldinger, R.
*QA4: A Procedural Calculus for Intuitive Reasoning.*
Artificial Intelligence Center Technical Note 73, Stanford Research Institute, November, 1972.

[Sandewall 79]
Sandewall E.
*A Description Language and Pilot-System Executive for Information-transport System.*
Report LiTH-MAT-R-79-23, Informatics Laboratory, Linkoping University, November, 1979.

[Simon 77]
Simon, H.
*The New Science of Management Decision.*
Prentice-Hall Inc., 1977.

[Sirbu, Schoichet, Kunin, Hammer 81]
Sirbu, M., Schoichet, S., Kunin, J. and Hammer, M.
*OAM: An Office Analysis Methodology.*
Office Automation Group Memo OAM-016, MIT Laboratory for Computer Science, January, 1981.

[Steele, Sussman 78]
Steele G. L. and Sussman G. J.
*Constraints.*
AI Memo 502, MIT, November, 1978.

[Steele 80]
Steele, G. L.
*The Definition and Implementation of a Computer Programming Language Based on Constraints.*
AI TR 595, MIT AI Lab, August, 1980.

[Steels 79]
Steels, L.
*Reasoning Modeled as a Society of Communicating Experts.*
Technical Report 542, MIT AI Lab, June, 1979.

[Suchman 79]
Suchman, L.
*Office Procedures as Practical Action: A Case Study.*
Technical Report, XEROX PARC, September, 1979.

[Sussman, Winograd, Charniak 70]
Sussman, G. J., Winograd, T., and Charniak, E.
*MICRO-PLANNER Reference Manual.*
AI Memo 203, MIT AI Lab, 1970.

[Sussman 72]
Sussman, G. J.
From PLANNER to CONNIVER - a Genetic Approach.
In *Proceedings of the AFIPS Fall Joint Computer Conference.* AFIPS, 1972.

[Tarski ]
Tarski, A.
The Semantic Conception of Truth.
*Philosophy and Phenomenological Research 4* :341-375, .

[Taylor 47]
Taylor, F. W.
*Scientific Management.*
New York, 1947.

[Tsichritzis, Hudyma 80]
Tsichritzis, D. and Hudyma, R.
*A Summary of Office Automation Research at the University of Toronto.*
Technical Report, University of Toronto, 1980.

[Tsichritzis 81]
Tsichritzis, D.
*Omega Alpha.*
Technical Report CRSG-127, University of Toronto Computer Systems Research Group, March, 1981.

[Walton 79]
Walton, R. E.
Work Innovations in the United States.
*Harvard Business Review*, July-August, 1979.

[Warren, Pereira 77]
Warren, D. H. D. and Pereira, L. M.
PROLOG - The Language and its Implementation Compared with Lisp.
In *Prooceedings of the Symposium on Artificial Intelligence and Programming Languages.* SIGART, Rochester, N.Y., August, 1977.

[Weinreb, Moon 80]
Weinreb, D. and Moon D.
*Flavors: Message Passing in the Lisp Machine.*
AI Memo 602, MIT, November, 1980.

[Weinreb, Moon 81]
Weinreb, D. and Moon D.
*LISP Machine Manual.*
MIT, 1981.

[Winograd 71]
Winograd T.
*Procedures as a Representation for Data in a Computer Program for Understanding Natural Language.*
MAC TR 83, MIT, 1971.

[Wynn 79]
Wynn, E.
*Office Conversation as an Information Medium.*
PhD thesis, Department of Anthropology, University of California, Berkeley, 1979.

[Zisman 77]
Zisman, M. D.
*Representation, Specification and Automation of Office Procedures.*
PhD thesis, Wharton School, University of Pennsylvania, 1977.

[Zloof 75]
Zloof, M.
Query by Example.
In *Proceedings of the NCC*, pages 431-438. AFIPS, 1975.