2

RADC-TR-89-383
Final Technical Report
April 1990

NEW GENERATION KNOWLEDGE SYSTEM DEVELOPMENT TOOLS

Intellicorp, Inc.

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order No. 5292

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-89-383 has been reviewed and is approved for publication.

APPROVED: Michael L. McHale

MICHAEL L. MCHALE
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ,JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Igor G. Plonisch

IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

NEW GENERATION KNOWLEDGE SYSTEM DEVELOPMENT TOOLS

Robert E. Filman
Conrad Bock
Roy Feldman
Joshua Singer
Richard Treitel

Contractor: Intellicorp, Inc.
Contract Number: F30602-85-C-0065
Effective Date of Contract: 12 March 1985
Contract Expiration Date: 30 June 1989
Short Title of Work: Prototype (B) Expert Systems Development Tool
Program Code Number: XE20
Period of Work Covered: Mar 85 - Jun 89

Principal Investigator: Robert E. Filman
Phone: (415) 965-5500

RADC Project Engineer: Michael L. McHale
Phone: (315) 330-3564

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Michael L. McHale, RADC (COES), Griffiss AFB NY 13441-5700 under Contract F30602-85-C-0065.

REPORT DOCUMENTATION PAGE	Form Approved OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE April 1990	3. REPORT TYPE AND DATES COVERED Final Mar 85 - Jun 89

4. TITLE AND SUBTITLE

NEW GENERATION KNOWLEDGE SYSTEM DEVELOPMENT TOOLS

5. FUNDING NUMBERS

C - F30602-85-C-0065
PE - 62301E
PR - E292
TA - 00
WU - 01

6. AUTHOR(S)

Robert E. Filman, Conrad Bock, Roy Feldman,
Joshua Singer, Richard Treitel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Intellicorp, Inc.
1975 El Camino Real West
Mountain View CA 94040

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research
Projects Agency
1400 Wilson Blvd
Arlington VA 22209

Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RADC-TR-89-383

11. SUPPLEMENTARY NOTES

RADC Project Engineer: Michael L. McHale

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report details the results of Intellicorp's activities during its participation in the Expert Systems Technology Base of the DARPA-sponsored Strategic Computing Program. Outlined is a software environment that provides knowledge system developers an integrated collection of "power tools" that support and accelerate their work. These tools include facilities for knowledge acquisition, knowledge representation, reasoning and user interface construction.

14. SUBJECT TERMS

Artificial Intelligence
Expert Systems
Blackboard Systems
Truth Maintenance
Multi-Agent problem solving

15. NUMBER OF PAGES

126

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Contents

Chapter 1

Introduction

This is the final report of the results of IntelliCorp's activities during it's participation in the Expert Systems Technology Base of the DARPA-sponsored Strategic Computing Program (SCP). Knowledge system developers now have available to them well engineered software environments (e.g., IntelliCorp's Knowledge Engineering Environment^{TM} (KEE^{TM}))[1] that provide integrated collections of "power tools" that support and accelerate their work. These tools include facilities for knowledge acquisition, knowledge representation, reasoning, and user interface construction. A fundamental goal of the SCP knowledge systems technology base is to produce a new generation of such tool environments that make knowledge system development easier and faster, reduce the expertise in artificial intelligence techniques needed to develop such systems, and broaden the class of problems to which the technology can be effectively applied.

IntelliCorp's goals in the SCP are:

- To provide the program's contractors a common tool base for developing both knowledge system component technology and applications, and
- To evolve that tool base into a new generation knowledge system development environment containing both enhanced versions of currently available tools and new tools that expand the range of application problems which can be effectively addressed.

Our methodology is to encourage use of the KEE system by SCP contractors as a common tool base and to evolve the new generation system from that base.

1.1 The New Generation Tool

The focus of the evolution toward the new generaion is on tools for building knowledge systems that do *model-based reasoning.* That is, we are developing tools for building and using structural models of application domains that can be used for performing multiple tasks (e.g., diagnosis, process control, situation assessment, simulation, training). Such multi-task models and their associated reasoning facilities can be contrasted with the collections

[1] Knowledge Engineering Environment, KEE, KEEworlds, and TellAndAsk are trademarks of IntelliCorp, Inc.

of task-specific production rules that characterize first generation expert systems. Our work involves extending the kinds of domain knowledge that can be modeled, the problem solving power available for processing that knowledge, the formal understanding of such extensions, and the size and architectural structure of the models. Our methodology is to use the tools in the KEE system as a base and to integrate the new tools we develop into that system environment.

In Phase 1 of our project we emphasized strengthening the object-description language to increase its expressive power and to clarify its semantic basis. The primary result of that research is the OPUS system. OPUS is a KEE-like system characterized by (1) recursive slots—the ability to describe objects at multiple levels, (2) relations—a facility for declaratively expressing binary relations, (3) continuously active deduction rules, (4) semantically sound inheritance mechanisms, (5) a context mechanism based on DeKleer's ATMS (KEEworlds), (6) extensions to the ATMS mechanisms to handle non-monotonic reasoning, and (7) abstract units—object-oriented extensions to the KEE core to permit the compact representation of objects.

1.2 Integration and Transition

OPUS uses the system architecture and primary components of the currently available KEE system. Because of the evolutionary approach we are taking in the development of OPUS and the attention we are giving to upward compatibility, component technology developed in the KEE environment by other contractors will be easy to integrate into the new system. In addition, we are building the new system using object-oriented programming and a modular, "open system" architecture that facilitates access to the system's internal structures and incorporation of new components.

We have supported the integration efforts of the knowledge systems technology base contractors by making KEE systems available to them via grants or academic discounts, and by consulting with them regarding interface requirements of KEE system modules.

Since the new generation system is an evolution of the current KEE system, application contractors who are using KEE in their projects can transition to the new generation technology with a minimum of effort. Prototype component technology modules developed by us and other contractors using the KEE system will transition directly to applications contractors who are using KEE. In addition, the system modules which we develop in this project will serve as prototypes for modules that may be included in future commercial releases of the KEE system and/or other IntelliCorp products that run in the KEE environment. Thus, IntelliCorp's productization activities can make available to applications contractors who are using the KEE system fully engineered and supported versions of much of the technology developed in our project as part of their standard KEE license agreements. This scenario has already been realized by the productization of the KEEworlds, ATMS, and virtual unit facilities, and is being realized in the Pilot's Associate program, for example, where the KEE system is currently being used by the team headed by Lockheed, and in the Air-Land Battle-Management program.

IntelliCorp has also provided consulting support to bidders for DARPA-sponsored SCP applications contracts (e.g.. CASES) and to Lockheed on their work on verifying and validating expert systems. The consulting is focused on informing the bidder about and determining the appropriate role in the proposed project for the tools and technology being developed by IntelliCorp and the other DARPA knowledge systems contractors.

1.3 Report Overview

In Phase 2 we have extended our work in three primary directions: improved representational and reasoning facilities, improved facilities for developing knowledge based systems, and experiments on the use of concurrency in knowledge-based systems. In our interim report we discussed the design of a distributed version of the OPUS system, the composite mechanism for compactly and precisely expressing the relationships within and between objects which themselves have objects as parts, and a formal representation of non-normal defaults and non-monotonic reasoning. This, our final report, is roughly grouped into three components—mechanisms for knowledge representation and acquisition, tools for the developers of knowledge-based systems, and explorations in concurrent programming.

We deal with knowledge representation in Chapter 2, where we discuss the Assertion Nets (ANets) language, a language developed as a tool for knowledge acquisition. ANets focuses on elicitation of object structure from experts and on aiding the translation of expert procedural knowledge to more general, pattern-directed form.

In Chapters 3 and 4 we turn to tools to aid the programmer of a knowledge based system. Chapter 3 describes a system for static analysis of the calling, binding, and assignment patterns of object-oriented systems. This is a particularly difficult task, as the calling structure of such programs are dynamically determined at run time. In Chapter 4 we present a graphic knowledge-base browser that seeks to deal effectively with the problems of getting a proper overview of a large knowledge base and displaying the relations within that knowledge base.

Our work on concurrency has two components. The first of these is an exploration of the idea that knowledge-based systems are (often) best expressed as a collection of concurrent, interacting *knowledge sources* or *agents*. This idea first reached concrete expression in blackboard systems [21,53], and has seen further realization in proposals such as Contract Nets [61] and the Scientific Community Metaphor [41]. The scientific question this research addresses is to what extent systems can profitably be built of interacting agents organized around economic or other principles. In Chapter 5 we describe a blackboard facility for KEE-like systems. Chapter 6 describes some experiments in multiagent problem solving—we programmed a system or interacting, competing agents to a job-shop scheduling task and compared their behavior under different control algorithms.

The second thread in our work on concurrency is research on genuinely concurrent versions knowledge systems and knowledge system algorithms. For example, our work on the factory scheduling system has led to the implementation of concurrency control mechanisms for KEE and OPUS; our interim report discussed OPUS protocols for loosely-coupled distributed systems. In our concluding chapter, Chapter 7, switch to the opposite architectural extreme, presenting an algorithm for rule-based inference on the Connection Machine

1.4 Publications

Our contract has been a particularly fertile source of scientific and technological results. We have had impact on not only the SCP technological base and the commercial knowledge-based systems tools market but also on the scientific community. We list below those papers that have already been published based on our work under this contract.

1. Bock, C., Filman, R., Morris, P., and Treitel, R., "Next-Generation Knowledge-System Tools," Proceedings of the 1989 AAAI Spring Symposium on Knowledge System Development Tools and Languages, Stanford, California, 1989, pp. 1-5.

2. Fikes, R., P. Morris, and R. Nado, "Use of Truth Maintenance in Automatic Planning," In DARPA Knowledge-Based Planning Workshop. Austin, Texas, Dec., 1987. Also in Proc. NASA AI Forum, Palo Alto, CA, Nov. 1987.

3. Filman, R. E., "Reasoning with Worlds and Truth Maintenance in Frame-Centered Integrated Systems," 1989 ACM Computer Science Conference, Louisville, Kentucky, 1989 pp. 2-3 (invited talk).

4. Filman, R. E. "Architectures for Distributed Problem Solving Systems," Distributed Artificial Intelligence Workshop, Sea Ranch, California, 1985, pp. 191-192.

5. Filman, R. E., "Retrofitting objects," ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA-87), Orlando, Florida, October 1987, pp. 342-353.

6. Filman, R. E., "Reasoning with worlds and truth maintenance in a knowledge-based programming environment," *Comm. ACM, v. 31*, no. 4, April 1988. pp. 382-401.

7. Morris, P.H., and R.A. Nado, "Representing actions with an assumption-based truth maintenance system", *Fifth National Conference on Artificicl Intelligence*, 1986, pp. 13-17.

8. Morris, P.H., "A truth maintenance based approach to the frame problem', *Proc. Workshop on the Frame Problem in Artificial Intelligence*, Los Altos: Morgan Kaufmann 1987.

9. Morris, P.H., "Curing Anomalous Extensions," *Sixth National Conference on Artificial Intelligence*, 1987, pp. 437-442. (Best Paper Award in Knowledge Representation).

10. Morris, Paul H., "The Anomalous Extension Problem in Default Reasoning," *Artificial Intelligence, vol. 35*, 1988, pp. 383-401.

11. Nado, R., and Fikes, R., "Semantically sound inheritance for a formally defined frame language with defaults," *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, July, 1987, pp. 443-448.

12. Nardi, B. A., and Paulson, E. A., "Multiple worlds with truth maintenance in AI applications," *Proceedings of the European Conference on Artificial Intelligence,* 1986, pp. 437–444.

13. Treitel, R., "Parallel rule-based inference on the Connection Machine," IJCAI Parallel Algorithms for Machine Intelligence and Pattern Recognition Workshop, 1989.

Chapter 2

Eliciting Rules From Procedures

Assertion Nets (ANets) is a Planner-like language that elicits object structure from experts as they transform their procedural knowledge into and rules. It bridges the gap between experts, who prefer to think procedurally at first, and expert systems shells, which require rules and object taxonomies. The expert will find the transition from rules to procedures easy in ANets, because it requires few syntactic manipulations. In addition, it highlights the additional information needed when a procedure's inputs are restricted to the match information provided in the invocation of a rule. This information suggests the structure of the objects needed to support rules.

This chapter is broken into three sections. The first compares two major techniques for interactive knowledge acquisition. The second describes ANets as an example of one of those techniques. The third compares ANets to other approaches to the integration of rules and procedures.

2.1 Interactive Knowledge Acquisition

Knowledge-based systems are a programming technique adopted in the face of the complexity of real world problems. Real world problems are almost always NP-complete, which, it will be remembered, are those that can be solved in reasonable time only with knowledge about where the solution lies in the search space [28]. NP-complete problems overwhelmed with combinatorial explosion earlier "weak methods," such as hill climbing and means-ends analysis. Knowledge-based system shells, on the other hand, are designed to allow experts to enter knowledge necessary to prune the search space. Emerging standard components of such shells are object orientation (frames), pattern direction (rules), and context mechanisms [23,51]. These facilities remain uncommitted about expert knowledge they contain, so that experts have as much leeway as possible.

The distance that expert system shells keep from a priori speculations about knowledge also makes them difficult for experts to use. Almost all knowledge-based systems are created with the help of a knowledge engineer who is more familiar with the shell than the expert. Knowledge engineering is a long and cumbersome process of bridging the distance between expert and shell. The goal of knowledge acquisition tools is to automate the role of the knowledge engineer.

2.1.1 Techniques for reducing the distance between experts and expert system shells

One technique for automating knowledge engineering is to train experts in the use of expert system shells. This can be very successful given experts who are motivated and articulate. It has the advantage of producing a very good system, but it can take quite a long time.

A second technique is to optimize a shell for the particular type of domain knowledge and problem solving methods the expert uses. For example, the technique of *heuristic classification* assumes the expert is performing a diagnosis by categorizing input data or symptoms, heuristically mapping to a categorization of diagnoses, and finally narrowing the diagnosis to a particular problem or disease [11]. Categorizing knowledge into input data, heuristic mapping, and solution taxonomies is an example of *knowledge level analysis* [52]. This analysis abstracts from the particular implementation of knowledge in rules or frames to give the method that the expert is using to do a certain task. Shells that acquire knowledge for a particular task through an interface designed with a knowledge level analysis are called *task and method specific* expert systems.

Most knowledge acquisition systems recently built are task and method specific. A diagram from [9] shows some systems as mappings between various tasks and problem solving methods used to do them (see figure 2.1). MORE acquires the knowledge necessary to do a diagnosis using the heuristic classification method. For example, it will prompt the expert for more symptoms or more specialized symptoms when a disease is only weakly tied to its symptoms; or it will ask the expert for symptoms causally closer to a hypothesis if no symptom is strongly linked to it [39]. SALT acquires knowledge needed to do a backtracking search. Il it detects a loop in the domain rules, it will ask for default values to break the loop. It prompts the expert for the fixes when constraint violations occur at a point in the search space [47]. ETS prompts the user for attributes that will distinguish elements of a solution space, yielding a classification of the elements [8].

One disadvantage of task and method specific systems lies in paucity of problem solving methods that are currently understood. Clancey, aided by an articulate domain expert, managed in a few years of work to isolate the heuristic classification method. Skeletal or variant synthesis modifies existing solutions to fit a problem. Propose and revise or backtracking search attempts to intelligently generate solutions, test them against some evaluation criteria, and modify the solution based on the criticisms. Attempts to understand domain knowledge have been unsuccessful enough to remind us of the difficulty of NP-complete problems.

A second disadvantage of task and method specific systems is that they must have a good model of expert problem solving to keep the expert from being frustrated. The expert is shielded not only from the implementation of the method, but is also prevented from suggesting any work-arounds that might be necessary when the system is not solving the problem correctly. These systems are frequently closed even further by interacting with the expert through a machine-controlled dialog.

A third disadvantage of task and method specific knowledge acquisition, at least as it is practiced, is that the problem solving methods address heuristic solutions rather than causal or "deep" models of the domain. For example, Clancey's heuristic classification method

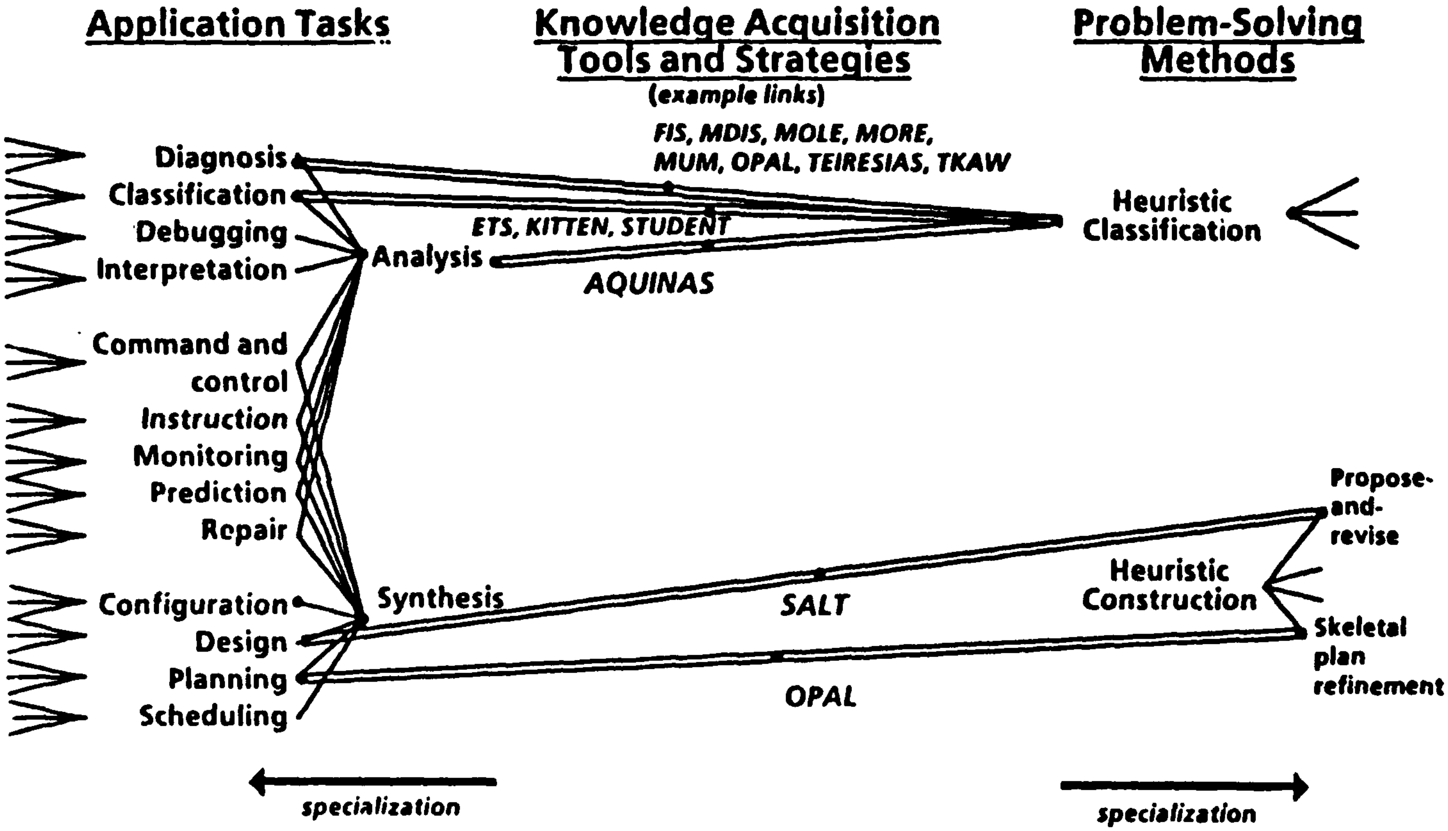


Figure 2.1: Interactive Knowledge Acquisition Tools

maps symptoms to diagnoses without any underlying model of the system being diagnosed. Without a causal model the system can only have limited predication and explanation capability. Large amounts of expert initiative are associated with creating and using causal models (simulation, modification, etc) [43], but are restricted in task and method specific systems.

A third technique for automating knowledge acquisition is to continue improving expert system shells with facilities that make them easier to program and understand. Among these might be representation of processes and structured objects; program analysis that discerns otherwise obscure system connections, such as data usage in a program and the run time direction of indirect calls; and wide spectrum languages that fit expert styles of programming. This *extended knowledge-based* approach adheres to the idea that NP-completeness should be handled with large amounts of domain knowledge. It is best, in this view, not to second-guess the expert, but rather provide tools that improve expressiveness of the expert system shell.

An advantage of the extended knowledge-based approach is that tools can be incrementally suggested by experts. It allows for an open system that can include an accessible causal model. A disadvantage is that the expert must become proficient at using an artificial language.

2.2 Assertion Nets

One of the gaps between experts and expert system shells is that experts are so enmeshed in their problem solving context that that they have difficulty translating their experience into the shell's encoding: objects and rules. Their expression tends to be procedural and they will even try to build procedures using rules [69]. They also have difficulty determining which elements of their domain should be objects and which attributes.

We might not bother the expert with this translation from procedures to rules and objects except for the modularity and reusability gained in doing so. Procedural solutions that an expert gives to each particular problem are not as useful as general principles and categories that would be applicable to new problems. Rules and object taxonomies are appropriate for context-independent information. Rules are invoked indirectly by the patterns that they presume and conclude. Object taxonomies are useful for many problems different from the ones in which they are developed.

Assertion Nets (ANets) is a wide spectrum language in which procedures and rules can be represented uniformly. The expert can begin by encoding procedures and then gracefully transform these into rules. Remarkably, this transformation yields suggestions for the expert about object structure.

2.2.1 Transforming a procedure to rules and objects in anets

Rules separate queries and changes to the database into different parts of the rule (premises and conclusions). Procedures, however, allow database interactions to be freely intermixed. From this point of view, rules are a special case of procedures. ANets is a rule-like language in which a series of queries and assertions are executed. with bindings from any query being passed on to the queries and assertions following it. For example, the procedure below staffs of a film production. The first query in the procedure retrieves the producer for the film. If a producer exists for the film, it is passed along along to the second query; otherwise, the procedure halts with a failure the way a rule would. The second query determines the level of funding for the film. The third statement is an assertion recording the funding level. The next two sets of four statements determine the director and the cinematographer of the film. The procedure is responsible for selecting the director and the director is responsible for selecting the cinematographer. Each set of four statements ends with a cut, which like Prolog's cut insures that the bindings up to that point are not pursued any further [12]. In this case, they insure that only one director and one cinematographer are selected.

```
STAFF-FILM (?film)
   Query: (the producer of ?film is ?producer)
   Query: (?producer gives ?funding to ?film)
   Add:   (the funding of ?film is ?funding)
   Query: (the occupation of ?director is director)
   Query: (the availability of ?director is available)
   Query: (an acceptable-hirees of ?producer is ?director)
   Cut
   Add:   (the director of ?film is ?director)
   Query: (the occupation of ?cinematographer
                                 is cinematographer)
   Query: (the availability of ?cinematographer is available)
   Query: (an acceptable-hirees of ?director
                                       is ?cinematographer)
   Cut
   Add:   (the cinematographer of ?film is ?cinematographer)
```

Readers may recognize this as very similar to Micro-Planner's ThProg [70]. The only addition so far is the cut construct. We leave other procedural constructs such as loops and conditionals to another section and proceed now to pattern direction.

The first step to creating rules from an ANets procedure is to extract subroutines. The last eight statements of the procedure above have been extracted into a subroutine that does hiring. In the argument list the hiree is not bound when the procedure is called, but is bound when the procedure returns and is available to the calling procedure.

```
STAFF-FILM (?film)
   Query: (the producer of ?film is ?producer)
   Query: (?producer gives ?funding to ?film)
   Add:   (the funding of ?film is ?funding)
   Query: (call hire director ?director ?film ?producer)
   Query: (call hire cinematographer ?cinematographer ?film
                                        ?director)
```

```
HIRE (?position ?hiree ?project ?hirer)
   Query: (the occupation of ?hiree is ?position)
   Query: (availability of ?hiree is available)
   Query: (an acceptable-hirees of ?hirer is ?hiree)
   Cut
   Add:   (the ?position of ?project is ?hiree)
```

The next step is to turn the subroutines into pattern directed procedures (rules). The arguments are removed from the hiring procedure below, because the information available to a rule must only arrive through the matching of one of its queries or assertions. In this case, the last statement, an assertion, must be matched in a backward chaining way to the queries in the caller. The procedure calls in the Staff-Film procedure become queries asking for a director and cinematographer for the film. The using-nets keyword simply means that the query should be answered using pattern invoked ANets rather than a database lookup. The information now flowing into the rule is only the film and the position sought. The output is the director or cinematographer.

Finally, and most importantly, we note that the restrictions imposed by the pattern direction suggest requirements on the object structure, as shown in the two capitalized lines of the hiring rule. The position that is responsible for hiring must be derived from the film and position sought. The position sought must be represented by an object that has an attribute giving the hiring position. For example, director must be an object with an attribute, called, say, Hiring-Position, having a value of Producer. The notion of hiring positions as attributes of occupation was only implicit in the original procedure. With ANets and the transition from procedures to rules, this implicit object information has now been made explicit.

```
STAFF-FILM (?film)
   Query: (the producer of ?film is ?producer)
   Query: (?producer gives ?funding to ?film)
   Add:   (the funding of ?film is ?funding)
   Query: (the director of ?film is ?director)
                                     :using-nets
   Query: (the cinematographer of ?film is ?cinematographer)
                                     :using-nets


HIRE ()
   Query: (the occupation of ?hiree is ?position)
   Query: (availability of ?hiree is available)
   Query: (THE HIRING-POSITION OF ?POSITION
                                  IS ?HIRING-POSITION)
   Query: (THE ?HIRING-POSITION OF ?FILM IS ?HIRER)
   Query: (an acceptable-hirees of ?hirer is ?hiree)
   Cut
```

```
Add:   (the ?position of ?project is ?hiree)
```

We can see that context independence achieved in rules has the price of using additional object representation to retrieve the context of the rule. The object structure makes explicit the representational and heuristic content of the original procedure. This object representation may be useful in other rules; for example, the hiring position may also be the position that sets salaries. When a representation is too burdensome or too infrequently used, the original procedures can be left as is. ANets have the additional advantage over rules that the expert can get something running quickly by writing procedures. The encoding is always in an executable state, because the procedures need only be transformed incrementally into rules. Finally, the original procedures can act as a test on the rule behavior.

2.2.2 Additional procedural constructs

In addition to sequential control and procedure calls, ANets provides conditionals and loops. We illustrate these with a net that directs an automated guided vehicle along a route through a factory to pick up parts, shown below. We use the programmer's notation because the procedure is so complicated. A loop terminates when any of its queries fail, so this loop terminates when the AGV has no more cells to visit, as recorded on its `cells-to-visit` attribute. The `:how-many` keyword simply requires the query to return only one binding. The queries on the arm length and the position of the AGV always succeed.

```
(defnet AGV (?agv)
  (loop
    (if (a cells-to-visit of ?agv is ?cell) :how-many 1)
    (if (the position of ?cell is ?cell-position))
    (if (the position of ?agv is ?agv-position))
    (if (the arm-length of ?agv is ?arm-length))

    (cond-true
      ((if (lisp (> (distance ?agv-position ?cell-position)
                    ?arm-length)))
       (retract (the position of ?agv is ?old-position))
       (add (the position of ?agv is ?cell-position))))

    (cond-true
      ((if (a parts of ?cell is ?part))
       (cond-true
           ((if (the status of ?part is defective))
            (add (the defective-store of ?agv is ?part)))
           ((if t)
            (add (the non-defective-store of ?agv is ?part))))
       (retract (a parts of ?cell is ?part))))
    (retract (a cells-to-visit of ?agv is ?cell))))
```

Conditionals are used to move the AGV closer to the machine it is currently visiting and to store the parts of the machine according to whether or not they are defective. The cond-true construct executes the statements after the first query if that query is true (the cond part of the name), but returns true whether or not the query is true (the true part of the name). The failure of the conditional will not halt the entire program, analogously to a standard conditional which can have all its branches fail without aborting its context. However, one difference is that no bindings may be passed out of a cond-true. If bindings must be passed out, then the cond construct must be used, which halts the entire program if all of its branches fail. This practice simply make more explicit the programming rule that requires the programmer to insure that some branch succeeds if bindings are needed.

The above procedure translated into rules, shown below, is very cumbersome. The facts AGV-Loop, Got-To-Cell, Get-Cell-Part, and Loop are introduced to insure rule order. They must be retracted before the rules are executed again to prevent the system from looping infinitely. One conditional is achieved by the second and third rules, and another by the fourth and fifth rules. Each conditional requires duplication of effort, in one case the comparison of the positions of the AGV and visited machine, and in the other case the query about defects. In ANets we do not introduce connecting facts between rules to maintain order, nor do we redundantly calculate for conditionals. In this example, if we compare the number of expressions in the assertion net with the number of premises and conclusions in the rules, we see a decrease in the amount of code by a factor of two and a half.

```
1)  IF    (AGV-loop)
          (a cells-to-visit of ?agv is ?cell) :how-many 1
          (the position of ?cell is ?cell-position)
          (the position of ?agv is ?agv-position)
          (the arm-length of ?agv is ?arm-length)
    THEN
          Retract (AGV-loop)
          Assert  (go-to-cell ?agv-position ?cell-position
                              ?arm-length ?cell)


2)  IF (go-to-cell ?agv-position ?cell-position
                   ?arm-length ?cell)
       (lisp (> (distance ?agv-position ?cell-position)
                ?arm-length))
    THEN
       Retract (go-to-cell ?agv-position ?cell-position
                           ?arm-length ?cell)
       Retract (the position of ?agv is ?old-position)
       Assert  (the position of ?agv is ?cell-position)
       Assert  (get-cell-part ?cell)


3)  IF   (go-to-cell ?agv-position ?cell-position
                    ?arm-length ?cell)
         (<= (distance ?agv-position ?cell-position)
             ?arm-length )
    THEN
         Retract (go-to-cell ?agv-position ?cell-position
                             ?arm-length ?cell)
         Assert  (get-cell-part ?cell)


4)  IF (get-cell-part ?cell)
       (a parts of ?cell is ?part)
       (not (the status of ?part is defective)
    THEN
       Retract (get-cell-part ?cell)
       Retract (a parts of ?cell is ?part)
       Assert  (the non-defective-store of ?agv is ?part)
       Assert  (loop ?cell)


5)  IF (get-cell-part ?cell)
       (a parts of ?cell is ?part)
       (the status of ?part is defective)
    THEN
       Retract (get-cell-part ?cell)
       Retract (a parts of ?cell is ?part)
       Assert  (the defective-store of ?agv is ?part)
       Assert  (loop ?cell)


6)  IF (loop ?cell)
    THEN
       Retract  (loop ?cell)
       Retract  (a cells-to-visit of ?agv is ?cell)
       Assert   (AGV-loop)
```

2.2.3 Variations of pattern direction the presence of procedural constructs

The procedural constructs of ANets have interactions with pattern direction that appear only in a hybrid system. For example, when a match is made into a procedure, the queries before the matched statement are set up as goals. This may include the queries in any of the callers of the procedures or any of their callers. It makes sense to do this in the case that a procedure is a special subprocedure of a more general one. However, if a procedure is used by other procedures as a utility, then the queries of its callers should not be set up as goals when matched. Such procedures can be executed independently of their calling environment and are marked with a `:top` declaration.[1]

If a match is made to a statement that follows a loop, then the loop must terminate at least once in order for processing to reach the statement. This means that pattern direction on the statement must include in the goals the failure of the conjunction of the queries in the loop.

If a match is made to a statement in the branch of a conditional, then the conditions of the previous branches must fail in order to reach that statement. So pattern direction in this case must include in the goals the failure of the conjunction of the previous conditions. Pattern matching to statements following a conditional must include in the goals the failure of the conjunction of all the branch conditions.

The queries after a `cond-true` or a `prog-true` (a sequential operator that always returns true) can ignore the queries inside the these protected constructs when setting up goals for pattern direction. However, queries inside the conditional or sequential must have queries before them set up as goals, just like any other query.

2.2.4 Future work

ANets as it stands cannot quite provide complete procedural control because the order in which queries and the query results are pursued is non-deterministic. This will present some barrier to the expert, who will probably prefer more determinism at first. We might imagine a RETE matcher applied to the goals and dependency direction to the bindings. These have not been implemented.

2.3 Related Work

Integration of procedural and pattern-directed languages began at least with Planner. Hewitt gives a definition of these processes:

> Explicit processing ... is the ability to specify and control actions down to the finest detail. [Procedural specification]

[1] Chaining in a rule system makes an analogous distinction. Rules that "call" a matched rule have their queries set up as goals. The queries in this case are not "top." In forward invocation, rules that call a matched rule are not invoked unless the system invokes backward chaining on the premises. These queries are "top."

> Implicit processing ... is the ability to specify the end result desired and not have to say very much about how it should be achieved. [Pattern-directed specification]

Procedural specification (explicit processing) is more commonly known as the control available in a programming languages like Algol or Pascal. It is realized with constructs such as sequences, conditionals, loops, and procedure calls. Processing in such systems is initiated by calling a specific procedure by a name unique to it. Pattern-directed specification (implicit processing) is usually associated with rule systems. Pattern-directed processing is initiated by asking or telling the system whether a statement is true. If the statement matches the premise or conclusion of a rule, then the rule will execute with variable substitutions made with the match bindings. The name of the rule itself need not be specified.

Hewitt suggests:

> Planner attempts to provide a formalism in which a problem solver can bridge the continuum between explicit and implicit processing.

This goal was motivated by work in robotics. On one hand, a robot must use pattern direction to make deductions from its own knowledge. On the other hand, it must efficiently execute strictly ordered procedures that achieve goals. Pattern direction for this latter task only introduces an inperspicuous and inefficient indirection. So Hewitt says:

> We aim for maximum flexibility so that whatever knowledge is available can be incorporated even if it is fragmentary and heuristic.

In the context of the discussion, we take him to mean that heuristic knowledge is procedural; and that procedural knowledge should be incorporable even if it is not rigorously deducible from other knowledge.

A subset of the language envisioned by Hewitt was implemented as Micro-Planner [70]. Winograd has the following example:

```
(deftheorem tc-cleartop
    (thconse (x y) (#cleartop $?x)
      go (thcond ((thgoal (#support $?x $_y))
                  (thgoal (#get-rid-of $?y)
                          (thuse tc-get-rid-of))
                  (thgo go))
                 ((thassert (#cleartop $?x))))))
```

The relevant aspect for us is that a procedural construct (thgo) is used with pattern-directed constructs like queries that invoke other rules (thgoal). Hewitt describes another construct called ThProg which operates identically to the sequential operator in ANets. These sorts of constructs were also used in languages such as QA4 [58].

However, research interest in mixtures of procedures and pattern changed course. Hewitt turned to a language in which the procedural aspects of deduction can be controlled.

For example, [36] contains an extended examination of the procedural aspects of deducing the statement

```
(and (Fallible ?x)
     (Greek ?x))
```

from the database of

```
(Human Turing)
(Human Socrates)
(Greek Socrates)
(Consequent (y) (Fallible ?y)
   <Goal (Human ?y)>)
```

The last fact is rule that concludes humanness from fallibility. In Planner, finding fallible Greeks is done with the program:

```
(ThProg (x)
  <Goal (Fallible ?x)>
  <Goal (Greek ?x)>)
```

It evaluates each expression, passing any bindings along to the evaluation of the next expression. The Goal statements are queries that may look for the fact on the database or use rules to get the answer. If it finds an answer, the bindings for ?x are substituted into the second goal, which in turn looks on the database or executes rules.

As Hayes points out [34], Planner requires a different rule for each way a logical implication can be used. For example, the rule for fallibility can be used to deduce that a person is not human (the contrapositive). In Planner this is:

```
(Consequent (y) (Not (Human ?y))
   <Goal (Not (Fallible ?y))>)
```

Nowadays, we say that Planner provides *production rules*, not logical implication. This property of Planner was an intentional reaction to the large search spaces generated by the indeterminant behavior of programs that operated on logical implication. Restraining indeterminant behavior is known as the *control* problem.

The control problem does not go away, however, by using production rules. The order in which to apply rules, match premises or conclusions, and pursue bindings from matching is still indeterminate. Much fruitful work has been done to provide mechanisms for control of production rules (Planner, Conniver [65], QA4, QLisp, aspects of InterLisp stack mechanism [5]). These approaches focus on techniques, such as possibilities lists and agendas, that suspend rule evaluation processes so that alternative rules and bindings may be selected. Suspended processes can be reinvoked if their alternatives prove unsatisfactory.

Suspended processes may be accompanied by their own data environments for storing their intermediate results. This line of research can be seen as culminating in Scheme [62,20], which though not directly addressing production rules, provides the primitive constructs for data environments and process suspension and continuation. We will call the above solution to the control problem the *deductive continuation-based approach.*

The deductive continuation-based approach achieves an intermixing of explicit (procedural) and implicit (pattern-directed) processing. If the user wants purely pattern-directed processing, no control information need be be specified. To more precisely specify the order of actions, the continuations that are not in order can be suspended permanently. In the extreme case of a strict ordering, all other orderings besides the one desired are suspended. Since the user must specify so many suspensions to achieve procedural control, deductive continuations do not provide "flexible" intermixing.

Let us look at an agenda-based example. A procedure expressed in rules means that a set of rules are executed in order. To order rules, one might first attempt to use a *connecting fact*: asserted in the conclusion of one rule, the connecting fact triggers another rule that has the fact as a premise. In the example below, three rules describe staffing a film production. The first rule gets the funding and asserts the `hire-director` fact, which triggers the second rule. The second rule finds a director (limited to one by `cut` keyword) and asserts the `hire-cinematographer` fact to trigger the third rule. The third rule hires a single cinematographer.

```
If (?film-producer gives ?money-amount for ?the-film)
Then
   (the funding of ?the-film is ?money-amount)
   (hire-director ?film-producer ?the-film)

If (hire-director ?film-producer ?the-film)
   (?director is a director)
   (?director is available)
   (?director is acceptable to ?film-producer) :cut
Then
   (the director of ?the-film is ?director)
   (hire-cinematographer ?film-producer ?the-film)

If (hire-cinematographer ?film-producer ?the-film)
   (?cinematographer is a cinematographer)
   (?cinematographer is available)
   (?cinematographer is acceptable to ?director)
                                            :cut
Then
   (?cinematographer is the cinematographer of ?the-film)
```

This seems to work until one considers using these rules with others. The connecting facts go on an agenda with facts from other rules. This means that the above procedure cannot be guaranteed to run without interruption; that is, the procedural control of the above rules has been compromised by the pattern direction of other rules.

We might solve the problem by instructing the agenda selection process to select only those items asserted by the above procedure for the duration of the procedure. Such instructions might be delivered by allowing the rules to modify the agenda selection function. The first rule must store the existing agenda selection function, concluding in the first rule that the agenda must operate in last-in-first-out mode. The last then restores the old agenda selection function. This still does not work if the procedure does not complete, as when a director cannot be found, because the agenda selector will not be restored. A form of "unwind-protect" must be constructed.

```
If (?film-producer gives ?money-amount for ?the-film)
   (the agenda-selection-function is
        ?old-agenda-selection-function)
Then
   (the old-agenda-selection-function
        ?old-agenda-selection-function)
   (the agenda-selection-function is LIFO)
   (the funding of ?the-film is ?money-amount)
   (hire-director ?film-producer ?the-film)

If (hire-cinematographer ?film-producer ?the-film)
   (?cinematographer is a cinematographer)
   (?cinematographer is available)
   (?cinematographer is acceptable to ?director) :how-many 1
   (the old-agenda-selection-function is
        ?old-agenda-selection-function)
Then
   (?cinematographer is the cinematographer of ?the-film)
   (the agenda-selection-function is
        ?old-agenda-selection-function)
```

The deductive continuation-based control style takes pattern-direction as the default processing mode. To get procedural control, a pattern-directed process must be controlled. The more procedural a program is, the more control is needed to override the default. Flexible integration requires that each end of the spectrum be default behaviors; that is, purely procedural or pattern-directed modes must be the simplest to express. It also requires a formalism that can be gracefully manipulated to move between the two modes.

We follow Hewitt in assuming that procedural specifications are a common way to express heuristic knowledge. For example, a domain expert frequently knows how to achieve a goal, without knowing why the procedure works. DeKleer counters this viewpoint by saying

> The procedural embedding of knowledge [referring to Hewitt's work] seems natural for capturing the knowledge of experts because ... For each specific problem

> he seems to follow a definite procedure with discrete steps and conditionals. In fact, an expert will often report that his behavior is controlled by a precompiled procedure. *One difficulty with this theory is ...* If one poses a new problem, differing only slightly from the one which we have previously observed and expert solve, he will explain his new solution as the result of a procedure differing in detail from the previous one. It really seems that the procedure is created on the fly from a more flexible knowledge base. [16]

When the above model of the problem solving process is used as a model of knowledge acquisition it appears that knowledge acquisition is done by accumulating small "chunks" of knowledge in rules. In actuality, an expert begins recording knowledge with procedures, since these come most naturally. It only becomes clear afterwards how the procedures can be separated into rules.

In ANets we do not attempt to model the actual process of expert problem solving, but rather provide a language in which experts can elucidate and record their own knowledge. Since ANets encompasses both procedures and production rules on a continuum, heuristic domain knowledge can be encoded as procedures, but also can be gracefully dissected into separate rules as the domain is formalized. Finally, we have discovered that this methodology guides the creation of object structure.

2.4 Conclusion

We have described a wide spectrum language, *Assertion nets, that covers both procedures* and rules. We have discovered that the transition from procedures to rules suggests object structure due to the restriction that pattern direction places on the input information. ANets fits more closely to experts tendency to express themselves more procedurally, while at the same time easing the burden of creating rules and object structure. ANets is a much needed step in bridging the gap between expert system shells and experts.

Chapter 3

Object-Centered Program Analysis

Object-centered languages utilize powerful forms of abstraction to reduce the cost of maintenance, modification, and extension of large software systems. Programmers can make changes more quickly with these systems when the change affects only one abstraction at a time. However, program changes that involve multiple abstractions are more difficult to perform because the protocols and dependencies between the abstractions must be maintained. Advances in programming environments and compilers in the past have been made in response to the challenges created by new abstraction mechanisms. Language-specific programming environments can help a programmer cope with issues of complexity and efficiency resulting from increased abstraction. A language-specific environment contains extensive knowledge of the semantics of the language and can automatically analyze programs to derive additional knowledge. Object-centered languages in particular are characterized by late binding of procedure calls. Analyzers for late binding can form the base of a programming environment that supports inter-abstraction changes. This chapter describes an analyzer and its use in making inter-abstraction changes in an object-centered language.

3.1 Inter-abstraction Modifications

It is well known in software engineering that maintenance, modification, and extension of a large software system comprises the majority of the effort in its life cycle. One software expert puts the figure of maintenance alone as high as 80% of programming cost [48]. One reason is that the scale of most military and industrial systems combined with programmer turnover require programmers to climb a steep learning curve on any particular program before making a change to it. Abstraction, including modularization and reusable code, help flatten the learning curve because the programmer can focus on parts of the system or use other abstractions without rewriting them. In other words, abstraction introduces "joints" into a program that allow the programmer to refer to unchanged abstractions by their names or interface instead of details of their internal mechanism.

For example, the central algorithm of a program always operates on some input data. To facilitate modifications, the data acquisition should be abstracted into a module that puts its results into memory locations that are used by the central algorithm. Each module refers to the other only through these common memory locations. In this way, the data acquisition and algorithm can be changed more independently of each other. The programmer can change the data acquisition from a user keyboard interaction to a automatic remote sensor without editing the delicate central algorithm.

Complete independence is not achievable, however, because a change in the central algorithm may require a change in the input data, which in turn requires a change to the data acquisition module. It is this sort dependence that makes large systems with many abstractions difficult to change. Multi-abstraction changes are in fact hampered by the boundaries between them. For example, if the data acquisition and central algorithm were combined, then a change in the algorithm that requires more data can be immediately accompanied by a change that retrieves the necessary data. With the code separated into modules, the programmer must find the data acquisition and change it in a way that must be coordinated with the change in the central algorithm. In summary, abstractions are useful when changes can be made in only one of them, but they become an increasing burden when changes are made across many of them.

Fortunately, approaches to the tradeoffs inherent in abstraction have been around almost as long as abstraction itself. One of the first uses of abstraction was the indirection in assemblers that allowed the programmer to assign names to memory locations and use those names in instructions operating on the memory locations. In this way, the memory locations can be moved without changing the rest of the code. Programmers soon found out, however, that debugging large amounts of assembly code without knowing the memory locations of the variables can be an enormous chore. To find the value of a variable, the programmer must search through all the code for the assignment of the variable, then access the memory location. Early debugging environments (DDT, RAID) changed assemblers to save the variable assignment tables after assembly. The environment provided easy access to these tables so that the contents of the variables could be retrieved on demand.

The type of environment necessary to support abstraction varies with the type of abstraction. For example. Fortran extended the abstraction of variables by assigning their memory locations automatically. A simple assembler link table will not handle this. Debugging environments in Fortran require the allocator to leave tables for the variable locations and provide easy access to variable contents.

3.2 Object-centered Programming

Object-centered programming languages have introduced new forms of abstraction that increase the demand for support tools to bridge abstractions. The features that will concern us are inheritance hierarchies, message passing, access oriented programming, and pattern-directed programming. The first of these, *inheritance* of structure and functionality over *hierarchies*. reduces the effort of defining new records by inheriting to the new records the structure and function from existing ones. A record such as `vehicles` may have subclass records of `automobiles` and `trucks`, inheriting fields like `engine` to each subclass. However, if programmers unfamiliar with this particular hierarchy attempt to change the name of a field in automobiles, they will find that the old named field is still inherited instead of being renamed. In a large hierarchy, the programmer must search through a possibly large set of ancestors to find the record in which the field is first introduced.

An object-centered program can operate on new kinds of records without changes, because the inherited fields can contain new programs invoked by sending *messages* to the fields. For example, in a factory simulation, maintenance of various machines in a factory is different according to the type of machine, but a single message called maintain can invoke the operation for each type of machine. When a new type of machine is added with its own maintenance routine, existing programs sending the maintain message still work. However, it is difficult to tell what code is executed by the message, because the messages are most frequently sent to variables instead of constants.

Object-centered systems also allow functionality to be initiated when fields are accessed can be added without changing the programs that access those fields (*access oriented programming* or *demons*). For example, a machine may have a field called Part.To.Process which has a demon that processes a part when it is put in the field. Unfortunately, a programmer or compiler looking at a program that changes the field cannot tell that the flow of control enters the demon. The programmer is confused when operations happen that are not explicitly in the code.

Another feature of object-centered languages, *pattern-directed programming*, allows very general patterns across many records to be detected and initiate changes to the records. Since the resulting changes can invoke other pattern-directed programs (*rule chaining*), they can be incrementally added without changing each other; in other words, pattern-directed programs are like ordinary programs broken up into separate objects that can be combined independently. For example, a rule for machine maintenance like

```
IF   (and (?machine is in class machine)
          (the work-load of ?machine is heavy))

THEN (oil-frequency of ?machine is often))
```

can be added to the collection of rules without changing the others. If any rules have oil-frequency in their *if* parts, then those rules are invoked when the *then* part of the above rule is recorded. For example, this second rule

```
IF   (and (?machine is in class machine)
          (oil-frequency of ?machine is often)
          (output-parts  of ?machine are ?output-parts))

THEN (wash-amount of ?output-parts is alot))
```

chains on the first. The cost of the modularity is that the connections between rules are not explicit, so the overall behavior is hard to understand and is slow to execute. For example, the programmer must manually search for the second rule to see what chains on the first and must repeat this process through the entire rule set. The rule interpreter must do the same thing.

Patterns in rules are usually expressed independently of the data structures that the patterns match against. For example, a rule which uses a clause `(the work-load of a-machine is ?work-load)` might look for a match on the variable in the `work-load` field of `a-machine` record, or it might look for a match for the `a-machine` key in the `work-load` hash table, or the `work-load` entry in the `a-machine` association list, etc. The user is not burdened with choosing a data structure when trying to express a fact or pattern. On the other hand, the programmer cannot tell from looking at a data structure what rules apply to it. A programmer must search through all the rules to find the answer. If the first rule of the previous paragraph (relating the oiling frequency for a machine to its work load) was accessible to the programmer from the `oil-frequency` field of any machine record, then the maintenance effort would be greatly reduced. The fields and their interaction through the rule-based programs then appear in one presentation, rather than separated by minutes or even hours of search, during which time the programmer has lost the mental context in which he/she was working.

All the above features of object-centered abstractions have one aspect in common: the data structures and behavior of the program are not completely determined until run time. The class hierarchy and hence the record fields change while prototyping a system and sometime even during execution. Messages and demons may invoke procedures on objects which are input data to the program. One rule may call any of a number of others, depending on the run time results of matching. These and additional dynamically determined behavior as is found in standard programs is called *late binding.*

3.3 Program Analysis and Partial Evaluation

The core of support tools for object-centered programming is a *program analyzer* that examines the source code of object-centered programs to determine the way abstractions interact. Since a program analyzer must understand what a particular program is doing, it is must operate on the code in the same way as an interpreter would if the code were being executed. For example, when the analyzer is given a `case` statement that branches on a variable must pursue all of the cases, binding the variable for each case and keeping the analyses of each branches separate from the others.

Furthermore, the late binding characteristic of object-centered languages means that the run time environment must be simulated to a certain degree. For example, the indirection in a message can only be resolved by knowing the function that implements the message. Perhaps the object receiving the message is part of the input data of the program, whereupon only some information provided external to the source code, the class of the object for instance, can determine the function. An analyzer that operates on a piece of code like an interpreter does, but with only some the information that would be available at run time, is called a *partial interpreter* or *partial evaluator.*[1]

[1] The functional programming and Lisp communities refer to interpreters as *evaluators*, because they evaluate the value of an expression.

3.3.1 Partial evaluation in standard programs

Partial evaluation is useful for analyzing ordinary programs as well as object-centered programs. A partial evaluation of a case statement branching on an input datum might be able to eliminate some case if something were known about the datum. For example, in the case statements below, if x were known to be of material iron, then all the other branches could be ignored in the analysis.

```
(defun testfn (testarg)
    (case (material-of testarg)
       (steel ...)
       (iron ...)
       (plastic ..)))
```

A partial evaluator that utilizes information about inputs must be capable not only of handling incomplete input data [4], but also incompletely specified input data. The partial evaluator then is a knowledge base and inference engine about the state of variables and records at every point in the program.

Partial information is best provided by analyzing the call to a function with some description of the arguments. For example, the following might be a way that an analyzer is called.

```
(analyze '(testfn ((material-of ?x iron))))
```

The analyzer is given a function call with each argument replaced by a list of statements describing the argument in a predicate logic formalism. The variable ?x by convention refers to the value of the argument. In this case, the first argument to testfn is described by one statement saying that the material-of function (predicate) will return iron when called on the argument.

Partial information may be specified for one function, but passed along to others:

```
(defun testfn (testarg)
  (case (material-of testarg)
    (steel (subtestfn testarg))
    (iron ...)
    (plastic ...)))

(defun subtestfn (subtestarg)
    ...
    (case (material-of subtestarg)
      (brass ...)
      (aluminum ...))
    ...
    )
```

The case statement in subtestfn can be reduced to zero or one branch because the analyzer can detect that the information about variable testarg can be passed along to subtestarg.

Partial evaluation of conditionals, including case statements must analyze the code following the condition in the context of that condition. In the conditional statement below, the call to hardness-of may be able to be turned into a constant because in testarg is known to be made of plastic.

```
(defun testfn (testarg)
    (cond
      ((equal (material-of testarg) 'plastic)
       ...
       (hardness-of testarg))))
```

The code following the condition also includes code following the entire conditional statement.

```
(defun testfn (testarg)
    (let (hardness)
      (cond
        ((equal (material-of testarg) 'plastic)
         ...
         (setq hardness (hardness-of testarg)))
        (...))
      (acceptable hardness)))
```

The call to acceptable may be reduced to a constant *when analyzing the first conditional branch*, because hardness-of call has been reduced to a constant there. In general, the code following a conditional statement must be analyzed once for every conditional branch, to simulate the run time effect of having taken that branch. This process is called *unfolding thϵ conditional.*

3.3.2 Partial evaluation in object-centered programs

Suppose we have the class structure and function shown below and then we perform the analysis in third expression.

```
                     MATERIALS
                      /    \
          LIGHT-MATERIALS   HEAVY-MATERIALS


(defun testfn (testarg)
    ...
    (message testarg 'msgfield)
    ...
```

```
        (change-field testarg 'testfield 'newvalue))

    (analyze '(testfn ((in-class ?x light-materials))))
```

Here the indirection of the message can be resolved to a direct function call with the partial information given if it is assumed that the functions implementing messages are not changing in the instances of the classes. Similarly, for demons, the change to the **testfield** field can directly call the active value associated with it under similar assumptions.

Partial evaluation applied to pattern-directed programs provides a powerful way to detect rule connectivity. The partial evaluator matches *if* and *then* parts without complete execution of the rules. Consider for example the previously mentioned rules concerning the oiling frequency of a machine and the amount to wash its output. A partial evaluator can "execute" the premises of the first rule, making the assumptions about the class and work load necessary for the rule to succeed; then execute the conclusion, remembering that the `oil-frequency` is now **often**; then loop through other rules for ones that have premises satisfied by the assumptions and conclusions from previous rules; eventually the second rule is found and a link in the connectivity graph is made between the two rules.

Another feature of some class hierarchies which can facilitate partial evaluation is that the class of values allowable in a record field is known (*value restriction*). This corresponds in traditional programs to a type declaration. Any value put in the field is restricted to being member of the class specified in the value restriction. So if a program retrieves a value from a field and sends a message to it, the value restriction limits the flow of control to the procedure implementing the message in the class of the value restriction. For example, the machines in a certain plant may all be made by certain manufacturer. When the controller for the plant sends a message to a machine, the procedure that implements the message may be restricted according to the manufacturer of the machine.

The type information provided by value restrictions can also specify the classes of records that are moved from one field to another (*data flow*), so if a field is always filled with values that are in a subclass of the field's value restriction, then flow of control is narrowed even further. For example, a record for a machine with a field for the part on which it operates may have a very general type like **objects**. In a particular program, however, it may be that only members of the **metal-objects** class are sent to the machine for operation. If a program analyzer can detect this fact, then the code for any messages sent to the objects can be significantly restricted.

A similar behavior is true of *control data flow*, where a function of a record field is used as a condition on a branch that affects another record field, the value restriction information may be enough to tell whether the branch should be pursued. For example, if a machine operates on wood or metal, but in different ways, the operation will have a conditional testing the material of the part on which it is operating. If the part is always retrieved from a bin that only has wooden parts, then the value restriction on the contents of the bin will say this. The program analyzer can detect the value restriction and eliminate one of the branches of the conditional.

Rule connectivity analysis is also enhanced by using class hierarchies. Consider for example the following two rules about nuclear reactors, the first of which tells when the integrity of a vessel is challenged and the second of which tells when a steam generator is ruptured.

```
;; Vessel integrity
IF (and (?vessel is in class containment-vessels)
        (the radiation of ?vessel is high)
        (the pressure of ?vessel is high)

THEN >>> (the integrity of ?vessel is challenged) <<<

;; Generator rupture
IF (and (the primary-cooling-system of ?reactor
            is ?cooling-system)
   >>> (the integrity of ?cooling-system is challenged) <<<
        (the steam-generator of ?reactor
             is ?steam-generator)
        (the level of is ?steam-generator is increasing)

THEN (a current-state of ?reactor
             is steam-generator-tube-rupture)))
```

Since the conclusion of the vessel integrity rule unifies with the second premise of the generator rupture rule, it seems possible that the application of the former rule could cause the application of the latter rule. However, suppose there is sufficient information in the knowledge base about reactors to determine that the value of ?vessel in the vessel integrity rule could never be the value of ?cooling-system in the generator rupture rule. For example, the primary-cooling-system field may only exist in members of of the class reactors, and the primary cooling system of a reactor may have its values restricted to members of class cooling-systems. The hierarchy may also contain the constraint that the classes containment-vessels and cooling-systems are mutually exclusive. Thus, since tne value of ?vessel in the vessel integrity rule must be a member of class containment-vessels, it can never be the value of ?cooling-system in the generator-rupture rule. Essentially, the information in the knowledge base has allowed the analyzer to determine that the integrity field in the first rule does not apply to the same class of records as the integrity field in the second rule; in other words, the rules apply to different parts of the class hierarchy. The above result helps the programmer understand the function of the rules by reducing their connectivity.

In all of these cases of partial evaluation. we need to assume that the value restrictions and procedures implementing messages do not change in the members of classes. This is almost always true. More thorny is that the class information may be more general than a leaf in the class hierarchy. In this case, the information about the procedures implementing a message is a disjunction. The only effect is to blur somewhat the accuracy of the partial evaluation. In a well structured knowledge base, these effects should be minimal.

3.3.3 Other work in program analysis for software environments

Although there are a significant number of experimental software systems that incorporate some form of automatic code analysis (particularly systems addressing program synthesis and intelligent programming instruction), the only widely used programming environment that supports extensive code analysis is the Interlisp environment. The code analysis module is called MasterScope; a program which accomplishes for the Interlisp environment many of the goals we desire in a object-centered environment.

Masterscope contains a static code analyzer which understands Lisp semantics. The code analyzer extracts knowledge about a set of Interlisp functions, variables, data structures descriptions and stores it in a database. Masterscope also provides a query language for asking questions about the usage relationships among the components of a previously analyzed system. Queries in Masterscope are used in conjunction with other tools in the environment such as the editor, thus providing the programmer some intelligent assistance in the development and maintenance of programs.

Masterscope can answer questions such as "what functions call function x?" or "what functions use special variable v?" The program is *analyzed* by reading the code just as a programmer sees it. So, for example, it reads the definition of function x and sees a call to function y, whereupon it records that information in a database. When the user asks "what functions call function y?" Masterscope uses the database records to find the answer. Analysis of this sort is called *lexical* or *static* analysis.

Masterscope cannot handle the late binding constructs of object-centered programs. Concerning the class hierarchy, when an analyzer is asked to find the functions that change an automobile, it should automatically find the functions that change a coupe or a sedan. Since objects in the class hierarchy also include methods (programs), a masterscope should also be able to find the functions that send messages to coupes or sedans by asking about functions that send messages to automobiles.

Another problem for lexical analysis is that messages in an object oriented system are rarely sent to a constant object; for example, if a message is sent to an automobile, it is usually not sent to one that is specified directly in the code (lexically), like `joe's-automobile`. Usually the message is sent to a variable x that may or may not be bound to `joe's-automobile`. This means that lexical analysis will not work well in an object-oriented setting. However, if the variable has been bound to `joe's-automobile` earlier in the program, a partial evaluation is able to note that fact and analyze the message accordingly. More commonly, the variable will be bound to some instance of a class, like `automobiles`, and the message sent to the variable. Partial evaluation can find that a message has been sent to an instance of the class of `automobiles`, even though it does not know which instance it is. If we assume instances have the same methods as their classes, then the control flow can be narrowed to one procedure call.[2]

[2] The example assumes that the class structure (that is, which classes are subclasses of which others) does

The problem lexical analysis has with messages, it also has with access oriented programming. Suppose we have demon that calls in fire trucks when a building is on fire. A lexical analyzer like Masterscope cannot answer a question like "what functions start the fire trucks going?" without regard to whether it is a house or factory (subclasses of buildings) that is getting set on fire.

The only information a masterscope usually has about a class is a constant field name either in a message or data access. The field name heuristically restricts the class of the message or demon to those classes that have the field. However, this class can be an arbitrarily high object in the hierarchy, even the class that contains all objects. In the average case, the class will be somewhere in the middle, whereupon the number of methods or demons will usually be large.

Language-specific environments have been developed for object-oriented languages such as SmallTalk. However, none of them include a code analyzer constructed along the lines of MasterScope. Object-oriented programs are more difficult to analyze then purely functional Lisp programs because the late binding of procedure calls introduces ambiguity that cannot be resolved by a purely syntactic inspection. Our contention is that partial evaluation is necessary to resolve the ambiguities introduced by the late binding of procedure calls.

3.4 An Example of Program Analysis

Analysis integrated into a programming environment should proceed as seamlessly as possible and results should be easily accessible. Of the many ways this might be done, we have selected one that is most useful for day-to-day programming style. We will take as an example, a program that simulates the automated manufacture of a machined part. First the user selects a top level function the arguments of which can be partially specified. The following function instructs an object representing a bin of parts to process its contents.

```
(defun FACTORY-EXAMPLE (part-bin)
  (unitmsg part-bin 'process-parts!))
```

some information about input Next, the user decides what the partial information will be. In this case, the user makes a new class of bins that contain only steel parts:[3]

```
Partial-parts
  Subclass-of: Steel-parts
```

not change while the program is running. If it does, then the masterscope may need to record the relevant class information at the time the change is made. For example, if an object is believed to be a sedan at the time the message is sent, but then later it is believed to be a truck, then the masterscope may need to record that (what was believed to be) a sedan and an automobile was messaged. On the other hand, it may be that the user wants to assume that whatever class an instance ends up in at the end of the program is the right one, whereupon the analyzer has more of a problem on its hands. If the instance is determinable, then the class structure extant at the end of the program can be used; for example, when asked about messages to automobiles, the analyzer can find the functions that message any of the instances of automobiles at the end of the program. However, if the instance is not determinable, then the question cannot be answered, since there is no telling in which class the instance ended up.

[3] Our notation for objects is to use the left most indentation for the object name, the next level for the attributes, and the next for attributes of the attributes

```
Partial-part-bins
  Subclass-of: Part-bins
  Parts
    Value Restriction: Partial-parts
```

The user now invokes the analyzer passing it a call to the top level function, which in turn is passed a partial specification of the argument

```
(analyze-program
    '(factory-example '((in-class ?x partial-part-bins)))))
```

The result is a database of analysis information that is used by the editing interface to answer user queries. The user editing the `factory-example` function above can place the editing cursor on the unitmsg expression, type a command, and the following function will be displayed for editing:[4]

```
(defun PROCESS-BIN-PARTS (bin)
  (mapc (function process-one-part-instruction)
        (get.values bin 'parts)))
```

From here the user might execute a command to see if the unit operation will trigger an active value. In this case, it does not. They may then proceed through standard editing facilities to display the definition of `process-one-part-instruction`.

```
(defun PROCESS-ONE-PART-INSTRUCTION (part)
   (let (instruction-operation machine)
     (setq instruction-operation
           (get.value (pop (get.value part 'instructions))
                      'operation))
     (cond
       ((setq machine
              (some (function process-part-instruction-machine-p)
                    (unit.descendants 'machines 'member)))
        (put.value part 'position
                   (get.value machine 'position))
        (put.value machine 'part part))))
```

The user now asks again which unit accesses trigger active values and this time the last `put.value` is highlighted using a bold face font. This deduction is made by the analyzer with data flow analysis beginning at the partial specification of the argument to

[4] For the remainder of the example, we present the code without explanation to recreate for the reader the situation in which a programmer is trying to understand programs they have not written or have forgotten.

In the example code, `put.value` and `get.value` are modifications and accesses of fields, respectively. **Unitmsg** sends a message specified by its second arguments to the object specified by its first argument. The other arguments are passed as arguments to the message.

factory-example through process-bin-parts to find the partial specification of the argument to process-one-part-instruction. The parts of the bin are known to be in class partial-parts, so this is passed to process-one-part-instruction.

The user can now move the cursor to the highlit put.value call and ask for the definition of the invoked function the same way they ask for it in a message or direct procedure.

```
    (defun PROCESS-PART-AV (machine new-parts old-parts )
      (mapc (function process-part)
            new-parts))
```

From here, again using standard facilities, the user can display the process-part's definition:

```
    (defun PROCESS-PART (new-machine part)
      (unitmsg machine 'operate!
                       (pop (get.value new-part 'instructions))
                       new-part))
```

Again we have a message which the user can investigate. In this case, three functions can implement the message, of which we will show one:

```
(defun DRILL-PART (drilling-machine instruction part)
  (let (part-hardness
        (drill-cache (get.value drilling-machine 'tool-cache))
        drill-class drill-size drill)
    (setq part-hardness (get.value (get.value part 'material)
                                   'hardness))
    (setq drill-class
          (cond ((eq part-hardness 'high) 'heavy-duty-drills)
                ((eq part-hardness 'medium) 'ordinary-drills)
                ((eq part-hardness 'low) 'light-weight-drills)))
    (setq drill-size (get.value part 'size))
    (cond ((setq drill (unitmsg drill-cache 'reserve-tool!
                                drill-class drill-size))
           (put.value drilling-machine 'speed
                      (unitmsg drill 'speed part-hardness))
           (drill-part-sub drilling-machine)
           (unitmsg drill-cache 'cache-tool! drill)
           t))))
```

The user can now ask about active values, of which there are none, and about which conditional branches are guaranteed to succeed. In this case, the partial information limits the parts to being made of steel, which has a hardness of high. The analyzer can trace this through the function calls, unit operations, and value restrictions as before. The first branch of the first conditional is highlit.

The user may then proceed to look at the functions that implement the **reserve.tool!** and **speed!** messages. The second of these has two possibilities for all drills, one for heavy duty drills and one for ordinary and light weight drills. In this case, the user will find that the analyzer has already deduced the type of drill from the result of the conditional, it can display one message instead of two. The partial analysis limits the number of paths the program may take given the partial evaluation.

With more investigation, the user will discover that the analyzer had evaluated the **reserve.tool!** message to determine the drill class. It found the function implementing this message on the value restriction of the **tool-cache!** attribute of drills. This function and its subfunction turned out to be

```
(defun RESERVE-TOOL (cache required-tool-class
                     required-tool-size)
  (let ((reserved-tool (some (function reserve-tool-1)
                             (get.values tool-cache 'tools))))
    (remove.value tool-cache 'tools reserved-tool)
    reserved-tool))

(defun RESERVE-TOOL-1 (tool)
  (cond ((unit.descendant.p tool required-tool-class 'member)
         (cond ((> (get.value tool 'size) required-tool-size)
                tool)))))
```

The analyzer knows that any value returned from the **some** function must be a non-nil value of the function given, in this case **reserve-tool-1**. The function **reserve-tool-1** can only return non-nil if the tool passed in is a member of the required class, whereupon it returns the tool. The analyzer can then deduce that the **drill** variable in **drill-part** must be in the drill class returned by the conditional.

The user can continue investigating the behavior of the program under various circumstances of partial information. Partial information is essential for object-centered systems, because the late binding allows input data to alter the flow of control. For example, an object-oriented graphics system may exhibit faulty behavior when working with a certain class of user-defined objects. A inter-abstraction editor is essential to discover and repair the program. Given the understanding gained from such an editor, the user can make modifications with greater reliability.

We have given an example of the integration of program analysis into a object-oriented programming environment. New forms of indirection, as object-orientation is now, have always been followed by environments that cope with the difficulties they pose. Our claim is that such an environment is a natural step in the development of object-centered systems.

3.5 Future Opportunities for Program Analysis

Program analysis can be integrated into a program environment in other ways than resolving late binding issues. Two we present here are extending the notion of closures in CommonLisp and cognitive-load reduction in displays of control flow.

3.5.1 Closures and program analysis

An analysis of a program gives the data that it might use, or in the case of a programs with inputs fully specified, exactly the data that it will use. Generally, a program uses only a subset of data, even without all the inputs specified. Program analysis, however, does not restrict programs to any proper subset of data; it just reports the subset it does use. This facility is partially provided by CommmonLisp *closures.*

The code below shows a closure around the functions fn1 and fn2 that allows them use the data a and b. Similarly, for fn3 and fn4 using data c and d. All the functions are allowed to use e. No other functions can access the data except as specified.

```
(let (e)
  (let (a b)
     (defun fn1 (x y) ...)
     (defun fn2 (x y) ...))

  (let (c d)
     (defun fn3 (x y) ...)
     (defun fn4 (x y) ...)) )
```

Program analyzers can give function and data relationships not expressible in closures. Suppose another variable h is accessible only to fn2 and fn3. This closure conflicts with the others. The nature of the notation does not allow this to be expressed, even if it was implementable. The problem with program analyzers currently is that they do not allow the user to express these enforceable relationships as they can with closures. One might image an analyzer that allowed some relationships to be marked as enforced. For example, the user might indicate that the functions fn3 and fn4 that currently access variable h are to be the only ones that access variable h. We call these *generalized closures.*

One of the most difficult problems in programming is coping with side effects that causally link procedures through global data structures. If an analyzer qua scoping language existed, the global data structures could be generalized variables in generalized closures that specify which procedures communicate through with global variables. Such closures would limit the interaction between functions through global structures to those explicitly indicated. In prototyping mode, where such restrictions are cumbersome [2], the user could use the analyzer in non-closure mode of operation.

3.5.2 Graphic display of program control flow

Processes that are called for effect should be distinguished from those called for value. Process called for effect tend to be those on the "main line" of sequential processing through the program. Processes called for value are usually those that answer questions for the sequential process. A typical piece of code has the form

```
(defun test (a b)
   (let (c d)
(effect-fn-1 a)
(setq c (value-fn-2))
(effect-fn-2 c b)
...))
```

An analyzer that can distinguish functions that are called in valueless positions, such as effect-fn-2, from those called in valued positions, such as value-fn-2, can feed that information to a graphic presentation mechanism. The graphics can display more prominently the main lines of the program, with the functions called for value less prominently or called up on user request. This technique gives the most orienting information first. Once the user understands the principle sequential processes in a program, the detail of functions called for value can be understood within that framework.

Imperative languages (like, for example, Ada) recognizes and enforces the difference between subprograms that return values (functions) and those that do not (procedures). However, they does not distinguish those with side effects from those without. Such a language, if it existed, might be substituted for an analyzer. However, a language that does not enforce these kinds of distinctions is better for prototyping. A good prototyping language can reduce the cost of programming errors considerably [2]. Analysis may provide a bridge from prototyping languages to languages that enforce stronger distinctions by extracting those distinctions from prototyped code.

3.6 Related Work

We briefly survey some of the work relevant to our approach to improving programming environments for object-centered languages. Research in several areas is relevant, including work in interactive programming environments, program synthesis, partial evaluation, object-centered programming, and type inference.

3.6.1 Object-centered programming

The roots of object-centered programming go back to work on SIMULA [15]. The object-centered paradigm has been popular for simulation, systems programming, graphics and user interfaces. *The first complete object-centered language was* SMALLTALK[30].

Two ideas that are fundamental to the object-centered approach are message passing and inheritance. The message passing paradigm has been explored in depth by work in ACTORS [37]. Issues of inheritance have been addressed in FLAVORS, a object-centered language embedded in LISP, as well as variety of knowledge representation languages which include KRL [7], KEE [23], UNITS [63], and LOOPS [6].

An LISP language extension called GLISP was developed by Gordon Novak at U. Texas [55] that supports object-centered programming. Its features include data abstraction, message passing, and an object system with property inheritance. The data type facility provides some type checking capability for GLISP programs. GLISP also extends the LISP compiler with a symbolic optimizer that does function specialization based on knowledge derived by its type inference engine.

A more recent effort to include object-centered programming within LISP, which is close to becoming a official part of Common Lisp, is CLOS [17]. Unlike GLISP and other object centered languages that are in LISP, CLOS unifies message-sending and function-calling syntax. This has the advantage of supporting the incremental conversion of existing Lisp software systems written in a functional style to a object-centered implementation.

3.6.2 Partial evaluation

Partial evaluation is a program manipulation technique which transforms programs with partially specified inputs. The goal of a partial evaluator is to perform as much of the computation in a program with some inputs being undefined or only partially defined. Partial evaluation has primarily been used as a program optimization technique.

The first major work in the area of partial evaluation was done by Futumura who used the technique in his experiments with compiler compilers. Additional work on using partial evaluation as a symbolic optimization tool was done by Beckman and others at Linkoping University [4]. They were the first to advocate use of a partial evaluator as a programming tool in conjunction with data-driven programming techniques. Other work in partial evaluation for program optimization includes that of Kahn [38] and Novak [55].

3.6.3 Type inference

Strongly related to partial evaluation is another program analysis technique called type inference. It consists of obtaining, where possible, a description of the set of values that the result of a computation could have, using knowledge about the meanings of the operators used in the computation. For example, if an addition is performed on two positive numbers, the result will be positive (if a subtraction is performed instead, no such assurance could be given). Such information is valuable because it can enable a compiler to omit code that would otherwise be needed to check whether the value is in a certain set, or which of several sets it is in (e.g., to find out whether a number is positive or negative). The best known programming language which extensively uses type inference is ML [50]. The powerful type inference mechanism used in ML achieves many of the advantages associated with strongly typed languages without requiring the programmer to explicitly declare the type of every variable in a procedure. ML has demonstrated that most type information can be inferred by analyzing the calling relationships among the entire set of functions. The powerful type inference mechanism makes it possible for ML to be compiled very efficiently [10].

3.6.4 Programming environments

The roots of modern programming environments which support a wide range of programming activities can be traced to Interlisp [66,67]. The idea of building a rich environment specialized for a particular language was developed by AI researchers who wanted a set of powerful tools to aid them in their experimental programming. Work on SMALLTALK [29] also had a tremendous practical influence on the design of programming environments. The widespread use of graphics in conjunction with bit-mapped displays, the use of the mouse and menu user interfaces, display centered editors, intelligent program formatters, sophisticated trace and debugging packages, and incremental compilation are all innovations attributable to the InterLisp and SMALLTALK programming environments.

An important theoretical contribution to the design of programming environments was the work of Phillips [56]. His approach to programming environment design is based on uniform closure, the property that any environment object can access any other object in a uniform manner, and self-description, the property that all components of the environment have descriptions of their organization and structure that are accessible from within the environment. The particular environment he describes in his paper that has both of these properties is CHI, a programming environment designed for research in program synthesis.

3.6.5 Program Synthesis

Systems that perform program synthesis are of interest because they incorporate program analysis and they address issues of how to represent knowledge about programs. Early work in the area of program synthesis began with research by Green on the automatic derivation of programs from logic specifications [31]. Utilizing a theorem prover, it demonstrated powerful techniques for analysis of very high level programs.

The SETL project, developed at New York University [59], involved the design of a high level language (SETL) based on set data types. It has a data refinement sublanguage [18] that supports datatype implementation directives. The SETL compiler was extended to do automatic selection of data structures [27] from a fixed library based on the analysis of set *intersections and data flow analysis.*

GIST [1] is a specification language that can express temporal constraints on the behavior of systems. Research prototypes have been able to compile parts of the language [22]. The AP5 system [13], developed at ISI, is an extension to Lisp that can compile efficient procedures that access a relation database, which itself can be implemented using a variety of data structures selected by the developer.

The PSI system [32] is the most ambitious system to date to synthesize efficient programs. Its goal was to develop an architecture that combined a variety of automatic programming techniques, including program transformation, efficiency analysis, and natural language specification. PSI was composed of independent knowledge-based experts. They included a program model builder (PMB) which develops an abstract program model through a dialogue with the user [49], a coding expert (PECOS) which takes the program model and refines it into a set of implementations using a catalogue of transformation rules [3], and finally an efficiency expert (LIBRA) which acts as a consultant to PECOS to aid it in making design decisions [40]. The problem area of PSI is symbolic computation, and it has been successfully used to solve problems in the areas of list manipulation and sorting.

The CHI system [33] has built on top of PSI technology, in particular it has borrowed heavily from the program construction component of PSI. However, CHI differs substantially from PSI in its architecture. Instead of having autonomous experts, CHI uses a homogeneous collection of tools sharing common knowledge bases. CHI also differs from its predecessor by its use of a more humanly readable wide-spectrum language, called V, that covers both high-level program specification and programming knowledge. Both declarative and procedural statements are allowed, as well as facts about program efficiency. CHI has been successfully used to reimplement the system's rule compiler. The primary goal of the CHI research is to develop a self-described programming environment that supports a variety of software development activities.

3.7 Conclusion

We have described how program analysis can alleviate problems characteristic of programming in object-centered systems. These systems gain their flexibility and modularity by increased use of late binding of procedure calls. Late binding, like other forms of indirection introduced for ease of programming, requires a support environment that allows programmers to bridge the gap it introduces between the lexical representation of a program and its actual behavior. Late binding differs from other forms of indirection in that it requires knowledge of input data to resolve. Using this information requires in turn an analyzer that can execute programs with only partial input data. When such an analyzer is integrated into a programming environment it can clarify the control flow across the object-centered modules. A programmer otherwise restricted to making changes within one module can with such an environment make changes that require inter-modular coordination.

Chapter 4

The Knowledge Base Browser

In this chapter we describe the Knowledge Base Browser that we implemented as an experimental alternative to the standard tools for displaying the contents of knowledge bases. It attempts to ameliorate the problems associated with getting an overview of (part of) a very large knowledge base, and to deal effectively with the need to display binary relationships other than class-subclass and class-member.

4.1 Purpose: displaying large KBs

Developers of large knowledge-based systems cannot be at their most productive unless they can see their work on the screen. Displaying a large knowledge base (KB) in such a way that all the relationships between its objects are visible is expensive both in computation and in screen space; and while CPU cycles are rapidly becoming cheaper and more plentiful, pixels are not. The acuity of human vision and the size of offices also constrain the amount of information that can usefully and conveniently be displayed. Presentations that make heavy demands on the viewer's eyesight are, in any case, not conducive to easy or rapid apprehension of the overall structure of what is being displayed. Presentations that cover many square feet do not lend themselves to the correlation of details from different parts of the KB.

4.1.1 Managing the display

A browser for large KBs must therefore assist its users in two ways:

- managing the set of objects covered by the display
- managing the level of detail at which this set is displayed.

Users will often be interested in a particular area of a KB, namely some set of objects (probably covered by a union of classes) and the relationships between them. A browser must allow the user to confine the display to such an area, and must not require that the user first display the whole KB before selecting an area. Incremental, exploratory browsing of the KB must also be supported by allowing the user interactively to augment the area shown (and, conversely, to reduce it if some of it turns out not to be interesting).

Given an area of interest, users will not always want to see every object and relationship within this area. They will want to abstract some details away, replacing a large set of viewed items by a generalization of all of them or by an example. Thus, instead of showing twenty facts of the form "object A_i is related to object B_j by relation R", they may prefer to see that R has domain A and range B, with the individual objects not shown. Or they may want to see only A_3 related to B_8, with the domain and range not shown. For that matter, they may want to hide some relationships or objects altogether.

4.1.2 No implicit knowledge

We distinguish between a simple display facility and a tool for answering questions that require several objects or facts to be correlated, such as "Into how many equivalence classes does relation R divide class A?" While one can imagine a purely graphical interface that allows such questions to be asked and answered, it would be significantly more complex than one that simply allowed users to see selected subsets of the explicit knowledge in the KB. The browser we have built makes it possible for the user to find the answers to some such questions, but multi-step processes are involved.

4.1.3 No direct manipulation

We also have not attempted to build an "active" browser in which changes made to the graphical representation cause corresponding changes in the underlying KB. However, the browser allows access from a shown unit or slot to the commands provided by the rest of the Opus interface, such as displaying or deleting the object. The browser display is updated to reflect the side-effects, if any, of such commands.

4.2 Design

There is necessarily some overlap between the concepts of specifying a part of the KB and specifying a level of detail at which that part should be shown, since both of these boil down to specifying subsets of the knowledge. A KB is not quite like a map of a country, where one can specify an area of coverage in cartographical coordinates and independently give the size of the smallest feature of the terrain that should be shown. A user with an object-centered mind-set would define an "area" of a KB to be a set of classes and/or member units, and the "level of detail" to mean which members of each class, and what relationships between them, are shown. A more relation-oriented user might prefer to define an "area" by means of a set of relations, and the detail would consist of decisions to show or hide individual relationships and the objects they relate.

We have therefore made no hard-and-fast distinction between these two concepts in our browser; the same mouse clicks and menus can be thought of as adding and subtracting detail, or expanding and reducing the scope of the knowledge shown. We have, though, indulged our object-centered bias in some ways, as will be seen shortly.

4.2.1 Exploration

The foundation of our design is that the user is expected to know the name of some object, class, or relationship in which they are interested. They need not know its exact name (it can be chosen from a menu), but somehow a starting point must be found. The browser then displays this item and perhaps others connected to it (if a relation has more than one domain, it will ask which of these are to be shown). The user can then expand the scope of what is shown by selecting an item and asking to see more items directly connected to it. They can also ask the browser to remove an item (or several related items) from view.

The notion of "connected to" used when expanding the browser's display is fairly broad. A unit is connected to all of its direct subclasses, superclasses, or members, and to every class of which it is (even indirectly) a member; if it has no direct members, it is connected to its indirect members (i.e. the members of its subclass (direct and indirect)). A relation is connected to its domains and ranges, and to units that are connected to them. At present a unit is connected to relations whose domains include it, but not to those whose ranges include it, because they will not show up as slot names on the unit. We recognize that it is often useful to know what units are linked to a given unit; there is no fundamental reason why such information could not be available in the browser.

An additional option is available for relation links: a specific relationship can be removed and replaced with the corresponding general relation (linking its domain to its range), or a general relation can be expanded to show its specific instances.

In general, when several items of a given kind are connected to a selected item, the user will be offered a menu of them and asked which ones to add to (or remove from) the browser display. Two exceptions to this deserve note. When a unit is selected and the user asks to see a class of which it is a member, at most one class can be selected. This restriction stems from the way in which class membership is shown, as described below. Also, the options for showing subclasses or members of a class include showing all of them, some selected from a menu, or a few chosen arbitrarily by the browser.

If there is only one permissible item of the kind requested, the browser will usually display or remove it without putting up a one-line menu.

The consequences of removing an item from the display vary somewhat. If a unit is removed, all relation links to or from it vanish (and re-appear if the unit is ever added back to the display), but removal of a relation never affects units. When removing a class, the user can tell the browser to also remove everything in it, but the default is to remove only the class, leaving its descendants (if any were shown) still visible.

4.2.2 Filtering

After a certain amount of exploration, a user may become aware of subsets of the knowledge in the KB which are not of interest, and would like the browser to cease offering to display units or relations from these subsets. To allow "sticky" user preferences to be remembered by the browser, we introduce the concept of a *filter*. This is simply a predicate, remembered by the browser, with which it tests every unit and relationship before displaying them or offering them to the user. In fact there are two filters, one for units (including classes), and one for relations. Both filters can be incrementally updated, by selecting an object, class, or relationship to be filtered in or out; filters can also be removed if necessary. We expect that users will want to start with no filters (i.e. everything permitted), and incrementally expand both the display and the filters, filtering out irrelevant subsets as they become apparent.

Unit filter

The filter for units is essentially an expression that evaluates to a set of units, though for brevity we have used `AND`, `OR` and `NOT` rather than `INTERSECTION`, `UNION` and so on. A unit name stands either for the singleton set of that unit, or, if the unit is a class, for the set of the unit and all its descendants in the inheritance hierarchy. Thus the filter `(AND CARS (NOT (OR TOYS POLICE-CARS)))` admits vehicles that most people would feel comfortable riding in. It does not admit anything that is not a car or class of cars; a likelier filter for browsing a road traffic knowledge base might be `(OR ROADS PARKING-LOTS (AND CARS (NOT (OR TOYS POLICE-CARS))) (AND DRIVERS (NOT ALCOHOLICS)))`.

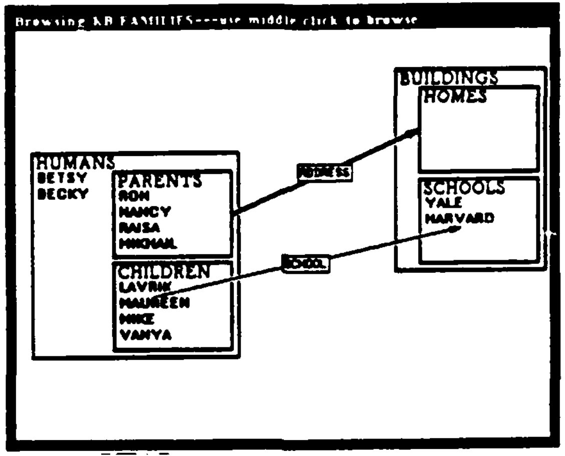
Relation filter

The relation filter specifies a set of relationships in similar fashion to the unit filter; its basic elements consist of a named relation and two unit filters, of which the first is used to delineate a subset of the relation's domain (or subsets of its domains) and the second does likewise for its range(s). A specific relationship will only be shown if the unit at one end of it is permitted by the domain filter and the other one is permitted by the range filter. Both these units must also be permitted by the current unit filter. Thus a relation filter might include `OWNS from TEENAGERS to HOT-RODS`, but under the unit filter given above, ownership of a hot-rod by an alcoholic teenager would still not be shown. The relation name can be null, in which case the filter element covers any relationship between any two units in the domain and range filters given; so if `Any relation from DRIVERS to PARKING-LOTS` were included, this could allow instances of a driver parking in a parking lot and instances of a driver owning a parking lot to be shown.

4.3 Presentation

A good principle of graphic design is that different kinds of information be conveyed in visually different ways. It is also helpful if the visual notation has some correspondence to the user's "natural" or intuitive model of the information being shown. Accordingly, we have decided to represent the containment of subclasses and members within classes by spatial inclusion, following the widely accepted Venn diagram notation used for sets; in keeping with the customary style of drawings of entity-relationship models, relationships that hold between units are represented by labelled lines linking them.

To further distinguish instances of a relationship that hold between two units from the existence of a relation with a domain and a range, we use thin lines for the former and thick lines for the latter. This has the intuitive significance that a thick line represents the (potential) existence of a large number of thin lines. Eventually, color-coded lines will be useful to emphasize certain units, classes, and/or relationships without causing the others to disappear.



In this picture, we see the classes of HUMANS and BUILDINGS, each with two subclasses. Some direct members of HUMANS are also shown. One of the CHILDREN is depicted as going to HARVARD; since no members of HOMES are shown, the only information about the ADDRESS relation is that it links PARENTS to HOMES.

Intersecting classes are hard to represent effectively in this notation, especially if more than two classes intersect. Given the graphical resources available, we have not tried to show intersections.

The position of units within class boxes, and the relative position of units connected by relation links, has been given no significance in our display notation; we reserve this "dimension" of information for the users of the browser, who may well wish to align certain units or to move important subclasses or members to the top of an enclosing class. While auto-layout algorithms can be designed that attempt to minimize line crossings and other undesirable features of a diagram, they cannot read the user's mind. (Attempts to do this by computer have rarely produred satisfactory results.)

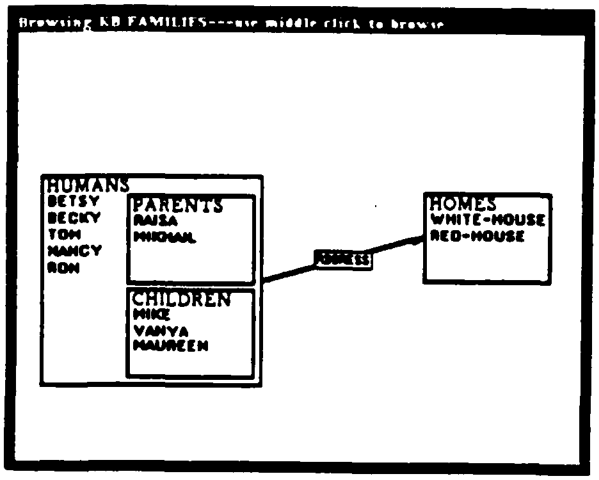
4.4 Implementation

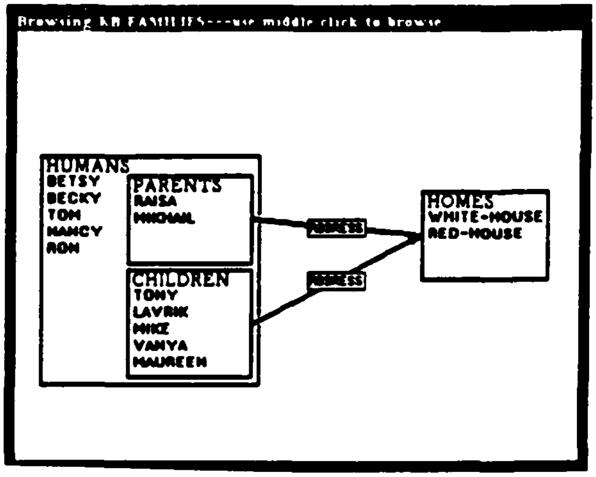
The browser prototype is implemented using the KEEpictures™facility which Opus inherits from KEE™; this provides most of the graphical functionality needed for the browser, including rectangular boxes for units and classes, and labelled lines of defined thickness. When boxes are moved, KEEpictures ensures that spatial relationships between them and their associated subboxes and links are preserved. We have added a primitive layout algorithm, in which boxes start with a small default size, then expand as more units appear inside them, and contract if units disappear; this is important for conserving real estate on the screen so that the user can explore a KB in comfort.

Ideally, the relationships shown by the browser would be instances of relations as defined using the Opus relation facilities. But for additional flexibility, we allow any slot whose valueclass is a class of units to be treated as a relation.

4.5 Example

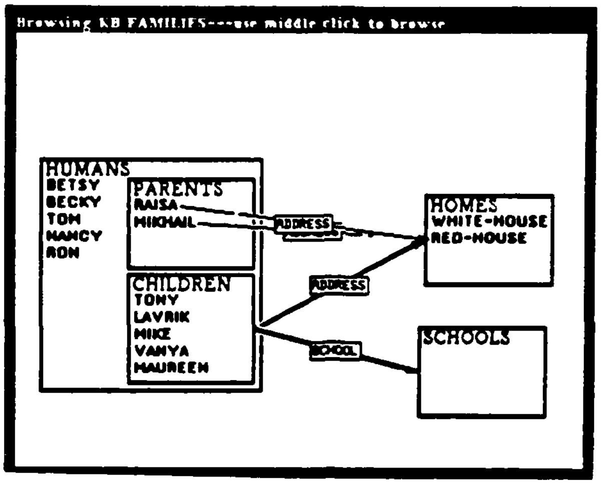
There follow some pictures excerpted from a session of interaction with the browser. In the first picture, the user has displayed the class of HUMANS, two subclasses, some members of each of these, and the class HOMES: HUMANS is linked by the ADDRESS relation to HOMES. Some of the HUMANS shown are in fact PARENTS or CHILDREN, but the user evidently didn't care which.



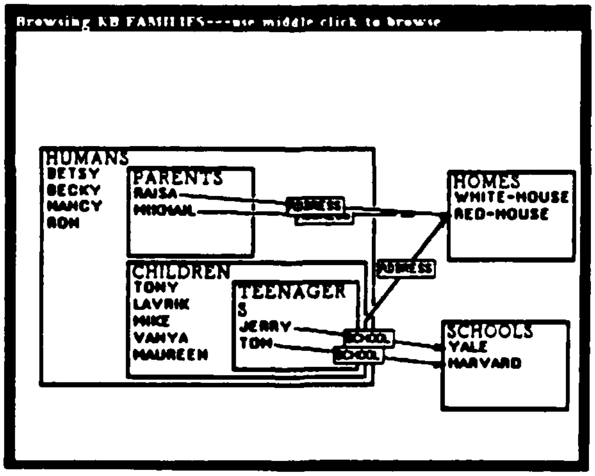


In the second picture, the ADDRESS relation has been specialized to show its application to the subclasses of HUMANS, and some more members have been added to CHILDREN.

In the third picture, the user has replaced the ADDRESS relation from PARENTS to HOMES with the actual addresses, and added the SCHOOL relation from the class of CHILDREN, which automatically displayed its range as well.

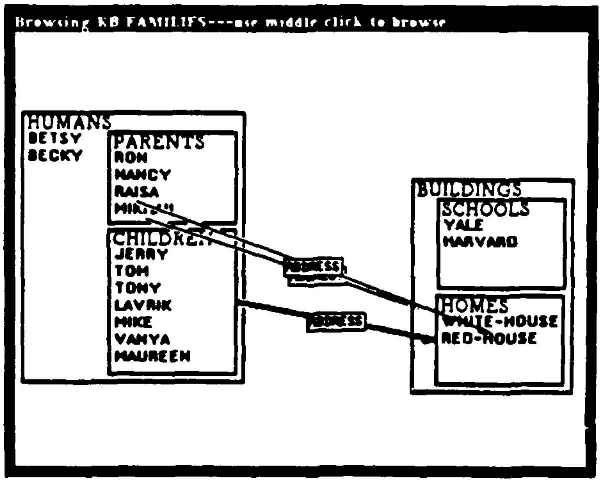


In the fourth picture, a subclass TEENAGERS of children appears, and the SCHOOL relation is specialized to show what schools they attend. Note that TOM, who was at first shown simply as a human, moved into the TEENAGERS box when it was created.



Finally, the common superclass BUILDINGS of HOMES and SCHOOLS has been added.

TEENAGERS has vanished (its members revert to CHILDREN). the SCHOOL information has also vanished, and two more HUMANS have been moved into the PARENTS box where they belong.



No pictures of filter manipulation are offered. because the filter is, by design, not displayed.

Chapter 5

The Opus Blackboard Facility

5.1 Motivation

The blackboard framework has proved powerful and useful for solving a variety of artificial intelligence problems [54]. Nii [53] describes the essential features of a blackboard system as:

1. Distinct sets of objects, each associated with a level of the blackboard

2. Properties possessed by the objects and named relations connecting them

3. Knowledge sources that detect patterns of objects on one level and, when activated, create or modify objects on another level.

These are exactly congruent with the object-based design of Opus:

1. Opus's classes are distinct sets of objects

2. Slots on objects provide properties, and Opus supports use of slots as relations

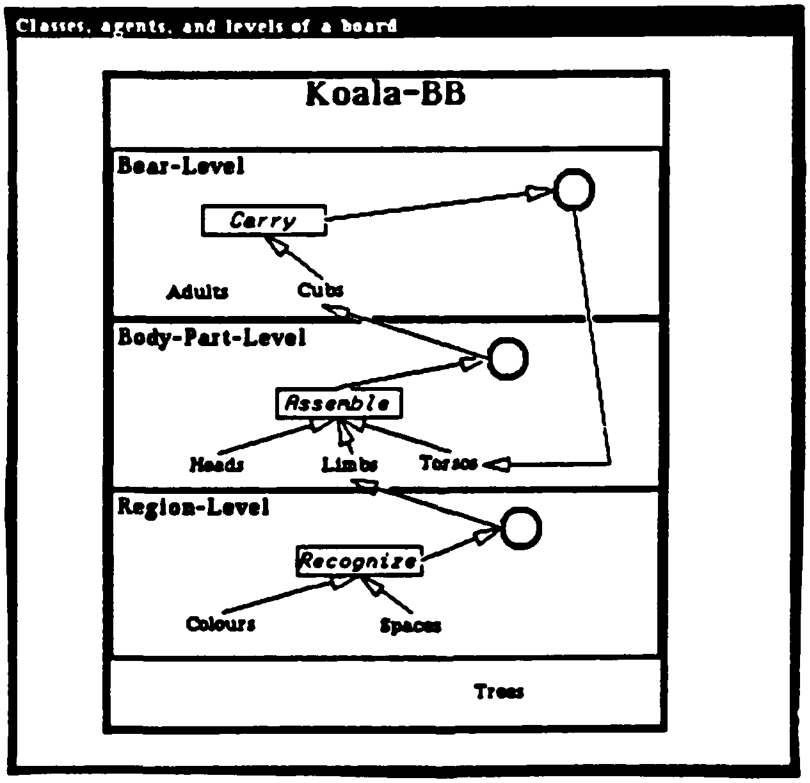
3. Opus's Active Rules detect patterns of objects and, when activated, can create or modify objects.

Hence a blackboard environment can be built on top of Opus at moderate additional cost. In addition to object-oriented programming, Opus provides other powerful and time-tested mechanisms, including a rule language (an extension of TellAndAsk^{TM}) and a possible-worlds system (an extension of KEEWorlds^{TM}), both of which are designed to interface smoothly to the object system. These mechanisms are therefore available to the blackboard system builder. The graphical interactive development interface supplied by Opus is also available to support building and debugging of blackboard systems, and has been augmented with some blackboard-specific tools built on the lower-level graphical functions of Opus.

5.2 Design

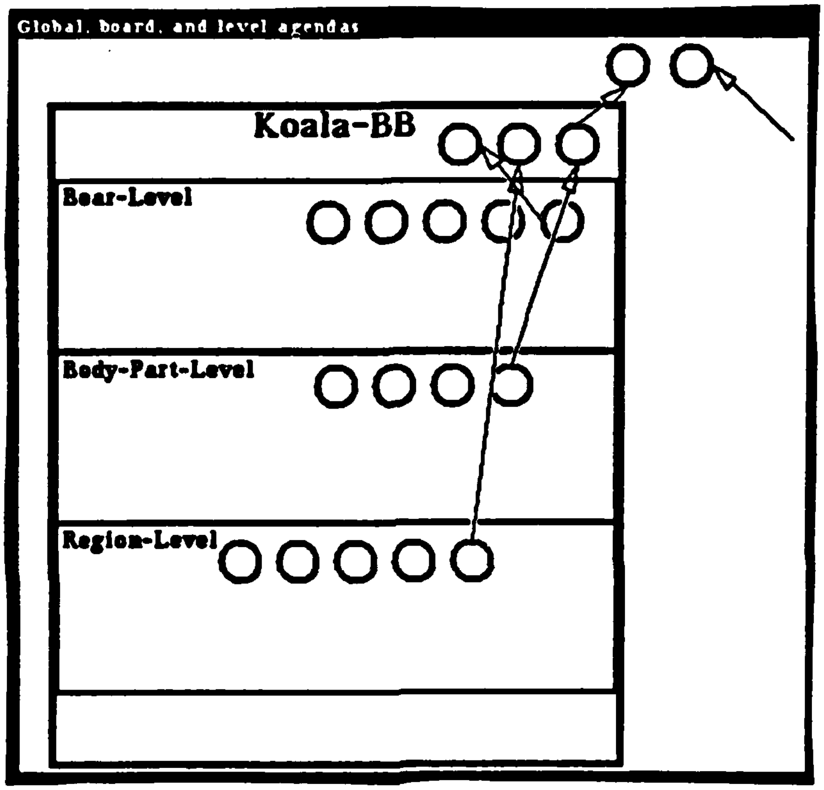
5.2.1 Structure

The blackboard facility in Opus implements a fairly conventional blackboard system as described in [53]. The highest level of structure defined by the blackboard facility is a blackboard panel, which we simply call a *board.* Each board can have one or more *levels* (a level belongs to only one board), which are used to partition the vocabulary of the subject domain (see [53] for further discussion of this). A level will generally have one or more *agents* (an agent belongs to only one level), which watch for patterns of facts on that level becoming true, and cause pieces of code to executed in response thereto. Such a pattern is known as the *precondition* of the agent, and the piece of code as its *action*. When facts matching the pattern become true, a *task* is created and is put on an agenda. When the task reaches the top of the agenda, the action of its agent is run, with the bindings from the match as parameters. The action can include adding or removing facts to or from any level of this board, or possibly even of another board. (We have adopted the terms "agent" and "task" as being more descriptive and concise than "knowledge source" and "knowledge source activation record.")



This picture shows a blackboard that attempts to pick out koala bears seen (or partly obscured) in a eucalyptus forest, as described in [53]. The unboxed text stands for classes of objects, the boxed text for agents, and the circles for tasks.

In the interests of control flexibility, there is a separate agenda for each level of each board, consisting of a list of tasks generated from the agents of that level. The agenda for a whole board is then a list of the top tasks from the agendas of its levels, and the top-level agenda is a list of the top tasks from the active boards. This architecture allows the agenda of each level to be sorted according to numerical priorities attached to the tasks, while a task's priority can be incremented by offsets attached to its level and the level's board (such an offset applies to all tasks generated by agents belonging to the level or board). Thus the relative priorities of different boards or levels can be changed independently of the relative priorities of tasks within a level. This allows for "strategic" control, or changes of "focus."



5.2.2 Representation

To maximize generality and transparency, most of these entities, including boards, levels, agents, and tasks, are represented as Opus units. This allows all of these entities and the information about them to be manipulated using the standard Opus functions on units and slots. However, this representation is inappropriate for the agenda mechanism, which uses ordinary list structures. The agenda could be modified to use data structures more suited for priority queues, such as heaps. Since boards, levels, agents, and tasks are units in Opus, their priorities can be changed by the action of an agent, and indeed the blackboard system allows for *controller boards* whose agents have actions that alter the priorities of the controlled board(s) or the levels, agents, and tasks thereof.

5.2.3 Invocation

A board can be active or inactive. When a board is made active, it finds all new sets of facts matching the preconditions of its agents and generates the appropriate tasks, adding them to the agenda. Here "new" refers to sets of facts that were not found during any previous activation. Any unexecuted tasks from previous activations are also added to the agenda. Additionally, the board remains watchful for further new sets of facts, and generates tasks from them as they arise.

Whenever the agenda has any tasks on it, the highest-priority task is scheduled for execution. Once a task has been generated, it stays on the agenda until it is executed or its board is deactivated. It is not possible to inhibit a task's eventual scheduling and execution except by deactivating its board (though the action may begin by checking for inhibitory conditions that could have become true in the meanwhile).

Three forms of debugging information are provided, all of which are selectable by the blackboard user.

- A text trace which records the creation and execution of tasks, indicating the bindings found for the variables in the agent's precondition
- A graphical trace showing tasks and how they were created by other tasks
- A continually updated display of the agenda showing how tasks are prioritized.

Filters can be attached to the traces, giving criteria for whether to include a task in the trace. It is also possible to give one or more criteria for whether a task should cause a break when the blackboard system starts to execute it. This allows developers to intervene in the blackboard problem-solving process at predetermined points.

5.3 Implementation

5.3.1 Overview

The Opus blackboard facility is built on top of two other components of the Opus system, namely the Active Rule interpreter [24] and the Assertion Nets compiler, described in Section 2. The Active Rule facility is used for triggering agents when their preconditions become true; the code has been modified slightly to distinguish between ordinary active rules, which are fired when their premises become true, and agents, for which a task is generated. The action of an agent consists of an Assertion Net, which is compiled into Lisp code at the time the agent is created. When a task is scheduled for execution, this code is run.

5.3.2 Creating boards and levels

There are functions `make-board` and `make-level` for creating boards and levels. Function `make-board` takes as arguments the name of the board and two optional arguments:

- A list of classes of objects, the intention being that all levels of the board shall have access to objects in these classes
- The name of the board controlling this board (each board can have at most one controller).

Make-level takes the name of the new level, the name of the board to which it is to belong (the board is created if it does not exist), and optionally a list of classes of objects to which the agents on this level shall have access.

Thus part of our example could have been created by

```
(make-board 'koala-bb '(trees))
(make-level 'region-level 'koala-bb '(colours spaces))
```

5.3.3 Creating agents

The preferred way to create an agent is to call the macro **defagent**, supplying the name of the new agent, the level to which it belongs, a default priority for its tasks, a keyword **:tm** or **:no-tm**, according to whether it is or is not desired that facts asserted by the action should be justified (in the sense of the Opus ATMS) by the facts matched to the precondition, and finally the specification of its precondition and action. This specification begins with the precondition, in the form of a TellAndAsk wff, followed by any number of assertion net expressions, to be executed in order.

Thus the Assemble agent in the example could be created by something like

```
(defagent assemble body-part-level 4 :no-tm
  (and (an adjacent of ?torso is ?leg)
       (?torso is in torsos) (?leg is in limbs)
       (an adjacent of ?torso is ?head) (?head is in heads))
  (if (cant.find (the torso of ?existing-bear is ?torso)))
  (if (= ?bear (lisp (new-possible-bear))))
      ;; this creates a new unit to represent the bear
  (add (the torso of ?bear is ?torso))
  (add (the head of ?bear is ?head))
  (prog-true (if (and (an adjacent of ?torso is ?leg)
                      (?leg is in limbs)))
             (add (a limb of ?bear is ?leg))))
```

The operation of defining an agent starts by creating a unit for it and putting it in the correct class (there is a separate agent class for each board). A class of agents has the form of a rule class, and the preconditions and actions of the agents can be edited using the "Edit Rules" menu item on the "Rule Class" submenu of the unit menu. The preconditions of the agent are parsed by the TellAndAsk parser and put into its **Premise** slot; the action, however, is handed to the Assertion Net compiler and the resulting function put into the Action slot (which ordinary rules do not have), leaving nothing useful in the Conclusion slot.

It may, under some circumstances, be useful to add ordinary rules to the class of agents associated with a board. These rules should be deduction rules, so as to make proper use of the Active Rules mechanism. They will fire when their preconditions are matched, independently of the blackboard agendas and *before* execution of any tasks that existed when the preconditions becamse true. Whether such rules fire before or after tasks whose preconditions became true simultaneously with those of the rule is not defined. The main advantage of using rules like this is that it eliminates computational overhead associated with agents and tasks; the disadvantage is that they are not controlled by the blackboard system's agenda.

Moving agent units into other rule classes, or using an agent class as a regular rule class, e.g. as an argument to `Assert` or `Query`, is not expected to be useful.

5.3.4 Triggering agents

An agent can be *triggered* to generate tasks at two times: when the class it is in (or rather, the board it is on) is activated, and when a new instance of its precondition becomes true while it is active.

Normally, a user would activate a board via the function `activate-board`, which causes the board's agents to be activated, rather than explicitly activating a class of agents. A class of agents is activated in the same way as a class of Active Rules. This includes checking for any existing facts that match the premises (or precondition), and firing the rule or triggering the agent.

The code for Active Rules has been modified to check, before firing a rule, whether it is an agent. If it is, a task is generated, with the bindings obtained while matching the premises of the rule (the precondition of the agent). The new task is given the default priority of the agent, then added to the agenda of the blackboard level to which the agent belongs (see below for an explanation of the agenda mechanism).

Since many tasks may be generated while activating a class of agents, scheduling of tasks from the agenda is suspended until all activation has been done. This ensures that the order in which tasks are executed will not be determined by the accidental order of matching and firing of rules, but by the blackboard agenda control mechanism discussed below. However, once the board is active, if a new fact is made known (e.g. by user input) when the agenda is empty, then any tasks resulting therefrom are started at once. This behavior can be changed by wrapping the assertion of the new fact(s) in the macro `without-agents`, which suspends scheduling of tasks while its body is executed, and then resumes scheduling if there are tasks on the agenda. Using `without-agents` is much cheaper than deactivating agent classes and then reactivating them.

A board can be deactivated by using the function `deactivate-board`, or the entire blackboard system can be deactivated by `(deactivate-bbs)`.

5.3.5 Access control

A mechanism is needed for checking at run-time that the variables in an agent's precondition are bound to objects accessible from the level of the agent, as specified when the level was created (see Section 5.3.2). Therefore, when an agent is triggered, the objects to which these variables are bound are checked against the classes of objects accessible to the agent's home level and board. Note that no distinction is made here between read and write access to an object, since in general it may be hard to determine, by looking at its action, which kind of access an agent has. Meta-levels (levels that control other levels) are implicitly allowed to access the objects accessed by the level they control, as well as tasks and agents of those levels.

No check is made for attaching a class to more than one board or level, or for overlapping classes being attached at different places. An object can, after all, be read by agents on one level and written by those of another.

The flag `*absent-level-check*` controls what happens when an object that is associated with no level or board is found to be bound to a variable in an agent's precondition. If it is `Nil`, access to the object will be allowed. But it can be set to `'warn` or `'break`, in which case the blackboard system will issue a warning or enter a breakpoint before permitting such access.

No checking is done on objects read or written during the execution of a task. Such checking could only be implemented by modifying every Opus function that can read from or write to an object: the function would have to check the object it was about to access against the level from which the currently executing task came. There is such a broad variety of these functions in Opus that any simpler enforcement mechanism would inevitably contain loopholes.

5.3.6 Agenda control

Changes to priority values can happen in two ways: by user intervention, and by the action of some task. Deveiopers can create "meta-levels", which are levels of a board that refer to tasks and agents as their objects, and whose purpose is to exercise control of the blackboard system by changing priorities. Meta-level agents would normally be triggered by the creation of a task satisfying some set of conditions. For example, it is believed that koala bears tend to stay close to each other, so it may be reasonable to increase the importance of a task that is examining a place close to where a bear has already been located; this could be done by a meta-level agent whose precondition was

```
(and (the place of ?bear is ?here)
     (?bear is in class bears)
     (an adjacent of ?here is ?near)
     (a here of the bindings of ?tk is ?near)
     (?tk is in class tasks))
```

It has been remarked by various researchers [57,60] that doing meta-level computation is sometimes less useful than doing base-level computation with a low degree of meta-level control. This observation compels the Opus blackboard environment to use a single agenda structure for base- and meta-level tasks, so that they compete on an equal footing. Thus a meta-level task can, so to speak, promote a base-level task to be more important (have higher priority) than most or all meta-level tasks, so that it will be scheduled without further interference from them. It is also possible to create meta-meta-levels to control the meta-levels, and these can be built as high as desired (at a corresponding cost in execution speed of base-level tasks).

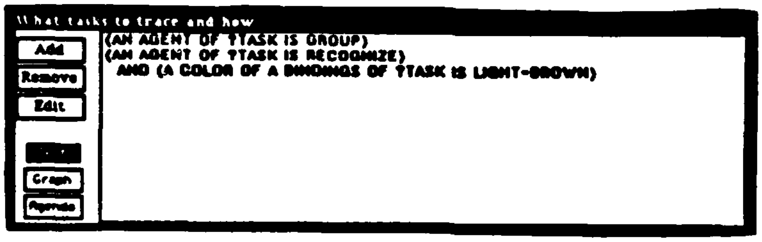
The Active Values mechanism of Opus is used to ensure that any change to the priority of a task or to the priority offset attached to a level or board is immediately reflected throughout the agenda structure. A macro `without-re-sorting` can be used to wrap pieces of code that do wholesale adjustment of priorities.

5.3.7 Task execution

A task is executed by calling the function in the `Action` slot of its agent. The arguments to the function (which is a compiled assertion net) are a set of bindings, namely those with which the task was created, and possibly a set of propositions on which any facts asserted by the task will depend (see above, Section 5.3.3). These propositions would be built from the facts that made up a true instance of the agent's precondition and caused the task to be generated.

5.3.8 Tracing and debugging

There are three tracing facilities available in the Opus blackboard environment: a text trace, a task creation graph, and an agenda display. All three can be turned on or off, and there is provision for a filter that determines at run-time which tasks are traced. Criteria for tracing a task can include the world in which the task is executing, the identity of its agent, the value to which a particular variable from the agent's preconditions is bound, and the occurrence of a value somewhere in the bindings, i.e. bound to any variable in the preconditions. This mechanism allows more selective tracing than straightforwardly calling `trace` on the function that is the compiled action of an agent. The flags and filter are more completely described in the accompanying User's Guide. The user controls them via a mouse-sensitive picture.



```
(AN AGENT OF ?TASK IS GROUP)
(AN AGENT OF ?TASK IS RECOGNIZE)
  AND (A COLOR OF A BINDINGS OF ?TASK IS LIGHT-BROWN)
```

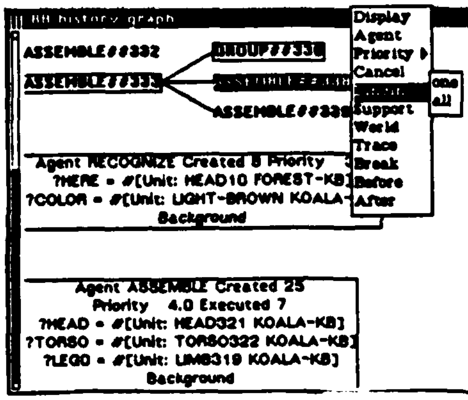
Text trace

The text trace prints messages when a task is created, before execution of a task starts, and after it finishes.

Task graph

The task graph grows an "ancestry" graph of tasks as they are generated. Each task is represented by the name of its unit, which is boxed if the task has already been executed.

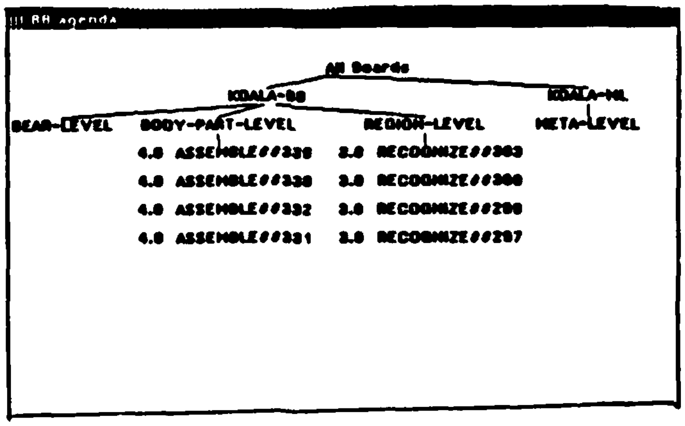
A task is the "parent" of all tasks that were generated while that task was running; the graph thus shows which tasks gave rise to which others.


BB history graph
ASSEMBLE##332
ASSEMBLE##333
GROUP##336
ASSEMBLE##338
Display
Agent
Priority
Cancel
Support
World
Trace
Break
Before
After
one
all
Agent RECOGNIZE Created 8 Priority 3
?HERE = #[Unit: HEAD10 FOREST-KB]
?COLOR = #[Unit: LIGHT-BROWN KOALA-
Background
Agent ASSEMBLE Created 25
Priority 4.0 Executed 7
?HEAD = #[Unit: HEAD321 KOALA-KB]
?TORSO = #[Unit: TORSO322 KOALA-KB]
?LEGO = #[Unit: LIMB319 KOALA-KB]
Background

The nodes representing tasks are mouse-sensitive. The functions available via mousing include displaying the task's priority, its bindings (or a selected binding), its agent, and the task that was executed immediately before or after it in sequence. The task's priority can also be changed. As the picture shows, tasks can be displayed by name only or in a fuller format; this is controlled interactively by the user.

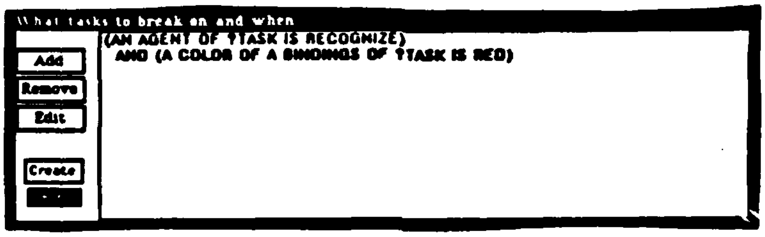
Agenda display

The agenda display shows the names of all active boards, the active levels, and below each level, a list of the tasks on its agenda and their priorities, with the highest-priority task at the top. The level and board of the task having globally highest priority are highlighted. The task nodes are mouse-sensitive in the same way as on the task graph, except that since a task on the agenda has not been executed yet, finding the task before or after it in execution sequence is not possible.



5.3.9 Breakpoints

Users can specify that a Lisp **break** should occur when a task satisfying certain criteria has been created, or is about to start executing. This facility is controlled by a filter similar to that described above and two flags, accessed via a picture. Such breaks provide a convenient opportunity for examining the state of the blackboard system and modifying it. The tracing and breaking filters can be changed and the graphical traces re-drawn to show information about the tasks specified in the new trace filter; the priorities of tasks can be changed, and they can be cancelled altogether (if not already executed or about to execute).



```
(AN AGENT OF ?TASK IS RECOGNIZE)
  AND (A COLOR OF A BINDINGS OF ?TASK IS RED)
```

Chapter 6

Multi-agent problem solving

This chapter reports on our work on multi-agent problem solving. Our research goal has been to explore the viability of multi-agent interaction as a general problem-solving approach. We conducted our exploration in the context of a job-shop scheduling environment, experimenting with different algorithms governing agent interaction to determine their consequences on the resulting job-shop schedule. We considered various utility metrics, paying special attention to an economic paradigm of agent interaction. In this paper, we first give some motivation for our interest in investigating multi-agent problem- solving solutions, then formally describe the problem domain of machine-shop scheduling which we have used to perform our experiments and the metrics we have used to assess the goodness of a particular machine-shop schedule. This is followed by a discussion of general issues in concurrent systems, our particular implementation of a multi-agent testbed, the different agent algorithms that we experimented with, and their performance with respect to our utility metrics. Finally we discuss the implications of our results

6.1 Multi-Agent Problem Solving

As AI technology matures, it is becoming increasingly necessary to build systems that both (1) integrate with existing environments and (2) provide multipurpose models of their domains [25,42]. That is, knowledge-based systems are becoming components of larger systems, accessing distributed data, exercising remote operations, and providing service to remote requesters. Systems like the earliest expert systems, dedicated to providing answers to a single class of questions, are becoming obsolete. The cost of knowledge acquisition and representation is too great for many such uneconomical uses. Instead, we are seeing model-based systems, where many intelligent processes can access a model of the underlying domain.

The requirement that knowledge-based systems act within larger contexts, the value of building systems that model domains rather than merely manipulate them, and the engineering principle of structural uniformity all suggest the design of knowledge-based systems in terms of interacting, communicating agents. Thus, from the point of view of the knowledge-based system, a remote database is just another agent that obeys a particular protocol. Similarly, from the perspective of other applications, both the parts of the knowledge-based system and the entire system itself can be viewed as agents. In some sense, the multiple-agent problem-solving paradigm is a recognition of the decomposition principle of object-oriented programming in AI [64]—that the same interface protocol should hold at every system level and that it must be transparent at the user level how a particular activity is accomplished.

The desirability of multi-agent systems naturally leads to the question of the appropriate organization of such systems. There have been several proposals in the literature for multi-agent architectures, such as blackboards [14,21,53] contract nets [61], and, extending the contract net notion of contract, various economic models [41,45,46]. (See Chapter 18 of [26] for a more thorough discussion of such proposals). In some sense, blackboards, contract nets and economic systems form an intellectual progression. Blackboards introduced the notion of explicit agents (knowledge sources) and provided a (simple-minded) architecture for their communication—a universal blackboard ether. Contract nets recognized the value of inter-agent negotiation and contracts, and the frequent need for more directed communication than simple broadcast. On the other hand, the contractors of contract nets bid and contract without motivation—the protocol does not suggest why particular agents ever want to contract for anything (or, correspondingly, why they don't always want to contract for everything). Economic theories provide both a motivation for service-providing contractors (i.e., to accumulate the representational equivalent of money) and a potentially global resource-allocation and scheduling mechanism. (Correspondingly, variable pricing mechanisms are an established method in operations research for optimization, see, for example, Chapter 8 of Lasdon [44].)

6.2 Machine-Shop Scheduling: A Multi-Agent Testbed

In this section, we motivate our choice of a machine-shop environment as a testbed for multi-agent problem-solving, formally describe the machine-shop environment and the scheduling problem, describe the metrics we use to measure the goodness of any solution to this problem, and discuss an example problem.

The usual methodology for performance evaluation is to develop a set of test-suites that present and solve representative problems and to measure performance on those examples.[1] Seeking something tractable but nevertheless sufficiently complex, we are focusing on the activities of a hypothetical abstract machine- shop which makes parts (for example, airplane parts). Any particular part begins as raw material, which is then put through a succession of machining operations until it becomes a finished part. The machine-shop has many machines, each of which can perform one or more abstract machine operation, such as "cutting," "drilling," or "twisting." For example, a machine-shop might have four cutting machines, two twisting machines, and six drilling machines. Orders for parts come into the machine-shop, and the machine-shop must produce the parts that are ordered. For example, at a given point in time, the machine-shop might have outstanding orders for

[1] There are, of course, pitfalls to this approach, even at the low-level of hardware architectures, and the pitfalls grow into craters as the to-be-tested architectures diverge and the problem complexities increase. But the only way to take the problem out of the range of the hypothetical into the domain of the scientific is to develop metrics of performance.

twenty engines and forty airplane wings.

Our machine-shop is driven by the interaction of two kinds of agents: machines and order-handlers. While machines are agents, orders per se are not agents. Rather, each order is handled by an agent called an order-handler, whose job is to make sure that the part which the order specifies is produced by the machine shop. The order handler accomplishes this by contracting with different machines for machine time. We will first give an informal example which illustrates the basic aspects of the machine-shop environment and the scheduling problem for that environment, followed by more formal definitions of each.

Suppose that O_1 is an order for a turbine blade, and let OH_1 be the name of the order-handler agent whose job is to see that O_1 is manufactured. And suppose, for the sake of argument, that in order to make a turbine blade, the raw sheet metal must be put through two successive machining operations: first cutting, then twisting.. Each of these machine operations is called a task. Thus, in order to make the blade specified by O_1, time must be scheduled for the first task on some machine that does cutting, followed by time scheduled for the second task on some machine that does twisting. OH_1 will converse with other machine agents in an attempt to get both of these tasks scheduled. For example, after conversing with other machines, OH_1 might obtain a schedule for O_1 as follows:

Task	Operation	Duration	Machine	Start-Time	End Time
1	cutting	3	Cutter3	2	4
2	twisting	1	Twister2	5	5

This says that the cutting operation is performed on machine Cutter3,beginning at time 2, and ending at time 4. The twisting operation is then performed on Twister2, starting at time 5 and ending at time 5. This constitutes a valid schedule for O_1. The most important constraint on scheduling is that subsequent tasks always begin after previous tasks finish. Thus, the twisting task can begin no earlier than time 5, since the previous task, cutting, ended at time 4. Note that the twisting task starts and ends at time 5. This is because we use a discrete time model. Tasks can only be scheduled at integer time-values, and each time value is really a block of time. One could think of the units of time as being hours. We also note that the different tasks require different amount of processing time. In this example, the cutting task required three time blocks, whereas the twisting task required only one. Of course, not all cutting tasks require three time blocks, and not all twisting tasks require one time block. In particular, some other type of part, other than a turbine blade, might require twisting for two time blocks, or four time blocks, etc.

It remains to be explained exactly how order-handlers converse with machines for machine time. In our environment, an order-handler who wishes to reserve time on a particular machine must send that machine a RESERVATION-REQUEST message. We will go into the particulars of the syntax later, but basically this message contains enough information to specify the time blocks desired, who wants them, and for what purpose, e.g.,"Hello, this is order-handler OH_1, and I would like to reserve time on your machine from time 3 to time 6 for task TASK1." The machine which receives a RESERVATION-REQUEST message must reply with a RESERVATION- REQUEST-REPLY message. This reply will reference the original reservation request, so that the order-handler will know which of possibly several requests to the same machine is being answered,.and the reply will state whether or not the request was granted.

Thus, the main occupation of order-handlers is to make reservation- requests. The main occupation of machines is to answer reservation-requests positively or negatively. In most of the protocols with which we experimented, a machine will deny a reservation only if the reservation would conflict with an existing reservation; however, in the economic model, a machine may also deny a reservation if insufficient money is offered for it by the order-handler. We will discuss the economic algorithm, and all the other algorithms as well, in detail later.

Both order-handlers and machines have the right to cancel a reservation. For an order-handler, this allows it to attempt to improve its situation. For example, if there are three machines, M_1, M_2, and M_3, that can perform task 1 (cutting) for the turbine blade, then OH_1 might very well send RESERVATION-REQUEST messages simultaneously to all of them, with the strategy of keeping the best reservation and canceling the rest. Suppose that OH_1 does this, requesting a reservation that begins at time block 0, or as early thereafter as possible, from all three machines. And suppose that M_2 responds first with a reservation that begins at time block 4. OH_1 might then send messages to all of the machines that can perform task 2 (twisting), requesting time at time block 7 (remember that task 1 requires three time blocks) or as early thereafter as possible. Meanwhile, machine M_1 might reply with a reservation for task 1 that begins at time block 2. Since this is better than the current reservation at time 4 which OH_1 has for that task, OH_1 will decide to keep the new reservation and to cancel the old reservation. It would do this by sending M_2 a CANCEL-RESERVATION message. Because of the better reservation for task 1 which OH_1 now has, the previous reservation-request for reservations for task 2 on or after time 7 might no longer be optimal—OH_1 can profitably use a reservation starting at time 5. OH_1 might then send out another round of reservation requests, this time requesting reservations on or after time 5. Obviously this can result in a few reservation cancellations. Finally, the response from M_3 might arrive, granting a reservation for task 1 at time 3. Since this is worse than the reservation at time 2 which OH_1 currently has, OH_1 would send back a CANCEL-RESERVATION message to M_3, but would have no reason to send out another round of reservation requests for task 2 because of it.

Machines may cancel reservations as well.by sending a RESERVATION- CANCELED message to the order-handler to whom the reservation was initially granted. In all of the protocols except the economic protocol, this happens rarely, because the only reason for a machine to cancel a reservation would be because it breaks down, or is taken down for maintenance.[2] In the economic protocol, canceling reservations by machines is occurs frequently (early reservations are almost always canceled.) In any event, for all protocols, an order-handler can never rely 100handler's algorithm must be able to handle the possibility that at any time a previously obtained reservation may be canceled.

In summary, then, there are four basic messages which order-handlers and machines

[2] For the purposes of modeling this possibility, we have implemented agents called imps, whose job is to randomly cause machines to fail according to some small probability. However, we have not yet had time to test system performance in the presence of imps.

Schedule of machine Puncher16

Order	Starts	Ends
SOH-20	1	3
SOH-11	6	8
SOH-29	9	11
SOH-37	13	13
SOH-23	16	18
SOH-36	20	22
SOH-5	24	24
SOH-34	25	27
SOH-27	39	39

Schedule of order-handler SOH-5

Task	Machine	Starts	Ends
Twist	Twister6	6	7
Coat	Coater14	12	13
Deburr	Deburrer19	14	15
Bend	Bender5	16	17
Twist	Twister6	18	20
Twist	Twister2	21	22
Lathe	Lathe4	23	23
Punch	Puncher16	24	24
Polish	Polisher9	25	25
Bend	Bender11	26	27

Table 6.1: Typical reservation schedules

send to each other in order to schedule the machine shop. They are:

Order Handlers	Machines
RESERVATION-REQUEST	RESERVATION-REQUEST-REPLY
CANCEL-RESERVATION	RESERVATION-CANCELED

We now give a more formal description of the scheduling problem, as follows: there are a set of machine-types, $T_1 \ldots T_i$; machines $M_1 \ldots M_j$; and order-handlers, $H_1 \ldots H_k$. Machine-types are conceptually the kinds of machining-operations that a machine can do—for example, drill, press, bend, twist, and so forth. Any given machine M_x has an *accessible* set of machine-types—the things that it can do. For example, machine M_6 may be able to drill and polish. Each order-handler, H_y has a corresponding order—a sequence of tasks, $S_{y1} \ldots S_{yl}$. Each task specifies a machine-type, duration, and additional parameters for a machine that runs that task, such as the drill bit to use or the force with which to press. (These last parameterizations being irrelevant for the simplest scheduling problems). That is, task five of an order might be to drill for three hours with a 5mm bit at 2500 rpm. We assume that tasks are indivisible. That is, the three hours of drilling must be done as a single chunk on a single machine. Additionally, the order-handler is given an "earliest time to start processing" and perhaps other parameterizations, such as a goal completion time and priority allocation. The specification of a scheduling problem is the specification of a set of machines, the types of the machines, a set of orders, and the tasks of these orders, including their type and duration. A typical problem might postulate eight machine-types, twenty machines, each of which has one accessible machine-type (selected randomly, with at least one machine for each type) and 100 orders, each starting at time zero and having from one to ten tasks (randomly selected from a uniform distribution), each task being from one to three hours long (once again, uniformly distributed) and requiring a random machine-type. Table 6.2 shows typical reservation schedules of a machine and an order-handler.

A valid schedule for the machine shop is a mapping of order-steps into machines such that every order is satisfied. The scheduling problem for that machine shop is to schedule all of the orders in "the least time possible," or, more specifically, to schedule all the orders such that some utility metric is minimized. Exactly what it means for the machine shop to be scheduled in the least time possible is open to debate. However, classical metrics for such scheduling problems include minimizing the total time to process all orders and minimizing the sum of the waiting times of each order. We call these respectively *SUM* and *MAX*. The distributed scheduling problem includes the restriction that scheduling be accomplished only through communication between order-handlers and machines, with no order-handler ever acquiring information about any other order-handler's orders, and no machine ever acquiring information about the status or schedule of other machines.

The utility metrics which we used to gauge the concept of "least time" are called SUM and MAX. The MAX of a machine-shop schedule is simply the end time of the last task which finishes its machining. This measure views the machine shop as though the entire shop were a single agent with a specific task, namely, to produce all of the parts specified by the orders. Its point of view is: we gave the machine-shop the assignment of machining all of these parts, and the machine-shop finished that assignment at time X, so X is the proper measure of the time it took to schedule the shop.

SUM measure is the sum of the processing-duration of each of the orders in the machine shop. The processing-duration for a particular order is the difference between the time the first task of the order began processing and the time the last task finished processing. SUM measure thus takes a reductionist view. The measure of the shop is the sum of the measures of the individual orders.

6.3 Issues In Concurrency and Multi-agent Systems

There are many problems which must be addressed in multi-agent systems that are absent in traditional programming environments. In this section, we will introduce these problems, and, in later sections, elaborate upon our solutions to them. The issues we discuss in this section are: message passing, dialogues and conversations, intra-agent concurrence, slot integrity, establishing context, sponsorship, and debugging.

Agents send each other messages. Therefore, there must exist some established, uniform protocol for message passing. This requirement is no different from that of single-process object-oriented systems, except that the protocol must be more intricate. In a standard object-oriented system, each object has its own "contract," which describes to what messages it will respond, what syntax to use, and what the effect of sending a particular message is. Such an object, when it receives a message, has no obligations to communicate back to the sender via another message. In fact, the object would rarely care who sent the message in the first place. Any communication could be done via a returned value, since, after all, the message was sent and received in the same process. Not so for the asynchronous communication between agents in a multi-agent system. Almost every message sent in a multi-agent system requires that another message be sent in reply. This is the only way that information can be passed. Thus, it is vital that every message explicitly state whom the message is from. It might also contain information about when the message was sent, a deadline for the reply, and information about what kind of reply is expected. It is no longer good enough for each agent to publish its own contract of which messages it replies to. In a multi-agent system, two agents must agree about the sequence of messages that can transpire between them, since it is only via this sequence of message passing that information is exchanged between them. Thus, whereas objects in a single-process system need only state what messages they respond to, agents in a multi-agent system must state what dialogues they engage in. Formally, a *dialogue* is a schema which describes a particular pattern of message exchange between two agents, and a *conversation* is any instantiation of a dialogue. For example, all order-handlers and machines must engage in a dialogue of length two that goes something like ⟨reservation-request ..., reservation-reply ...⟩. The ellipses are meant to indicate the parameters which must be supplied in any particular instantiation of the dialogue. Thus, the (stylized) sequence ⟨"this is order-handler 1 requesting a reservation from cutter3 with an early start time of 0 for task1 with a duration of 3," "this is cutter3 replying to order-handler 1 regarding a request for time starting as early as 0 with a duration of 3 for task1: you've got the reservation"⟩ represents an instantiation of the dialogue.

6.3.1 Intra-agent concurrency

We want our agents to be able to perform multiple operations simultaneously. For example, suppose $\texttt{OH}_1$ is in the process of computing the parameters for a message that it is about to send to $\texttt{CUTTER}_3$, when it suddenly receives a message from $\texttt{DRILLER}_2$, replying to $\texttt{OH}_1$'s previous request for a reservation. We would like for $\texttt{OH}_1$ to be able to handle this new message without stopping what it was doing before the message was received. Thus our agents will need their own multi-tasking mechanisms. Unlike most multi-tasking environments, however, the simultaneous computations of an agent may well be correlated. In particular, one might at some point invalidate the other. Suppose, for example, that $\texttt{OH}_1$ were making plans on the assumption that its previous reservation request of $\texttt{DRILLER}_2$ would be granted. If the reply from $\texttt{DRILLER}_2$ is negative, then $\texttt{OH}_1$ need no longer bother with those plans. This state of the world makes it difficult to write extremely smart order-handlers. For a smart order-handler would always know all the things that it was doing, and the ways in which new information could invalidate computations which have already been dispatched. We discuss this further in section 6.3.4.

6.3.2 Slot integrity

Before we can talk about slot integrity, we must make certain that the reader understands what a slot is. We assume the reader is familiar with object- oriented systems, and understands that each agent in our multi-agent system is an object in the standard sense, also sometimes called a frame. A slot, then, is one of the internal variables of an object, sometimes also called an attribute.

Because of the existence of intra-agent concurrence, agents require some mechanism, such as a locking or transaction protocol, to safeguard their slots against cross-computational collision. Such algorithms are necessary whenever multiple processes attempt to utilize a common resource. In the case of our agents, this amounts to protecting the integrity of their slots. For example, all machines have a RESERVATIONS slot which holds information about all the reservations which they have granted. A machine cannot simultaneously reply to two requests for reservation time, because the chance exists that it would award overlapping reservations. (both computations might, for example, read the slot at the same time, notice that time blocks 0 through 3 were free, and then attempt to grant these time blocks to two different order-handlers.)

6.3.3 Establishing context

A major difficulty in developing concurrent algorithms arises because one can make few assumptions about the state of the agent at the time the algorithm is dispatched in its own process. For example, without a synchronization protocol, one cannot assume that this agent is not simultaneously running the same activity in another process, that another process is not running which renders the first useless, or that the data elements that the process will examine are in a good state. For example, a scheduling sub- function, whose purpose is to schedule a particular task, cannot generally assume that the task is not already scheduled, or that the task's predecessor is already scheduled, even though the task or its predecessor may have been scheduled at the time the process was dispatched. Determining context even for the simplest of algorithms can be tricky. As a result, one tends to formulate algorithms that make as few assumptions as possible (preferably none) about the context under which they were invoked, the goal being to create algorithms which always do the correct thing, regardless of the context under which they were invoked, and which , in the worst case, merely do nothing if the context should not happen to be correct for their proper invocation. In writing a concurrent algorithm, the developer must be extremely clear about exactly which context-dependent assumptions can be made by the algorithm. Otherwise all hell will assuredly break lose.

In the following discussion, it is important to understand that every message which an agent receives is handled in a separate process.

In addition to establishing its external context, a process may also need to recover some internal context that was established prior to its dispatch. For example, agent 1 executing process A may establish values for the variables x, y, and z, and then send a message to some other agent, agent 2. When agent 2 replies to the message, the message-reception causes the dispatch by agent 1 of process B, which needs to examine the values of the no-longer existent variables x, y, and z. The only way process B can have access to these values is for process A to make them in some fashion global values. This can be done by creating special slots on the agent to hold the values, providing a means of passing them from computation A to computation B. However, this approach has two major problems. First, the agent object now becomes cluttered with a morass of state-saving slots which have no meaning except in the context of a very specific cross-computational interaction. Further, such a system requires that computation A never overlap with a copy of itself, for then it could not guarantee the integrity of its state-saving slots. (Alternatively, A could devise a more complex encoding of computational values and slots.)

6.3.4 Sponsorship

The issue of sponsorship is closely related to the issue of establishing context. Whereas establishing context is crucial to an algorithm's running correctly, sponsorship is the problem of determining whether or not a computation should be running at all. A particular computation might be dispatched in response to the occurrence of a particular event in the world, but be almost immediately rendered useless by the occurrence of another event. For example, consider an agent whose overall scheduling approach is to schedule its tasks sequentially from the first to the last, never scheduling a task until the task's predecessor has been scheduled. Suppose further that this agent lives in a world in which tasks can become spontaneously unscheduled (due to machine failure, for example). Then the spontaneous unscheduling of a previously scheduled task can throw a monkey-wrench into the agent's computation-state. He may be working on task 5, because task 4 has just been successfully scheduled, when all of a sudden task 3 becomes unscheduled, requiring, in this case, that the agent return to working on task 3. Ideally, we'd like the process which has been dispatched to work on task 5 to stop, because it is no longer sponsored. It is performing useless or harmful work, because the reasons which justify its having been dispatched no longer hold. Furthermore, it must be stopped cleanly, for various slots of the agent may be left inconsistent, if it is stopped at some arbitrary computation-point. Worse, the computation may have already sent messages to machines requesting a reservation for task five. This means the order-handler must be prepared to deal with responses to those messages, even though it might,naively, be expecting all responses to be about task three, not task five. Thus, not all the computational threads *sponsored by a particular activity* are *"in hand"* at any time. Nor is the simplest idea of retracting all consequences of a withdrawn sponsorship necessarily a good one. For example, one possible action policy on reservation-reply messages might be to cancel all reservations for tasks that are not sponsored. This is less than completely optimal, however, as the agent may be able to obtain a "good enough" substitute reservation for the spontaneously canceled one.

6.3.5 Debugging

Debugging is challenging in a multiprocess environment, as the logical threads of a conversation leap from process to process, whereas conventional (Lisp) debuggers are designed only to show the internals of a single process. Also,since the entire system is side-effect driven, errors will often occur because a slot has the wrong value, but it will be far from obvious exactly how the slot got the erroneous value in the first place. This side-effect nature of the system also makes it impossible to re-run many lisp-frames in which an error has occurred, because the re-running of the frame would cause, for example, a slot's value to be decremented twice, or a message to be sent twice, etc. The only safe way to re-run a side-effect-causing frame would be to have a mechanism for unwinding all the changes which the frame produced up to the point where the error occurred. If these changes were changes to the slots of the agent running the process, then one could envision an elaborate transaction mechanism which could handle the task. However, messages which the frame sent to other agents, once sent, are no longer the property-so to speak- of the agent or its processes. Our approach to debugging these systems relies on recording a log of inter-agent messages and developing functions to rummage around in this log.

6.4 System Implementation

In this section, we describe our implementation of a multi-agent environment. We describe our implementation of agents, messages, concurrence, locking, and debugging tools.

Every agent in our system is represented as a KEE object, called a *unit*. Units have internal variables, or attributes, called *slots*. Each slot can hold one or more *values*. Units can be sent *messages*, which cause their corresponding methods (the function-objects in the message-named slot) to be invoked. And the slots of units can have access-oriented daemons associated with them, which may perform side-effects when the value of a slot is accessed or set.

As in standard object-oriented programming, an agent may send itself a message as well as send any other agent a message. However, unlike in standard single-process object-oriented programming, messages which an agent sends to an agent other than itself must be asynchronous, while messages which it sends to itself may be either synchronous or asynchronous. The characteristic of a synchronous message is that the is message processed, and a value returned, all in the same process. An asynchronous message is not processed in the process that sent the message. Rather, that process continues immediately after the message is sent, and must expect a reply to the message to be received at some indeterminate later date.

Our system simulates concurrency using the TI explorer multi-process system. Each agent performs its computations in one or more processes. However, no agent is allowed to "cheat" by synchronously sending a message to another agent. Whenever a message to another agent is to be sent, the sending agent must create a new process which sends the message.

We have developed a set of macros to handle synchronous and asynchronous message passing. The set of macros themselves constitute a mini- language with their own syntax. Basically, there are a few options that a sender of a message has: to send the message synchronously or asynchronously, to send the message to itself or to another agent, to log the message or not (the message log as a debugging tool will be describe later), to send the same message to a list of recipients or just one recipient, and finally, in multi-recipient message- sending, to send message arguments to each recipient whose values are dependent upon who the recipient is. There exists a macro for each valid combination of these options. The name of the macro whose functionality corresponds to a specific set of options is determined by syntax rules governing the combination of the characters: - * > and <. - is the character for messages sent to an agent other than oneself (though should the recipient be oneself, there is no harm done). * is the character for messages sent to oneself. The appearance of a second - or * signifies asynchronous message-sending, e.g., -- or **, while a single - or * implies synchronous message sending. At least one > character is always required, .e.g., ->, -->, *>, **>, and hence has no meaning. However, the appearance of a double > means that the message should be broadcast to a set of recipients, e.g., -->>, ->>. Additionally, the appearance of a leftmost ¿ indicates the message should be logged (logging will be described later), e.g., >->, >-->, >*>, >**>, >->>, >-->>, etc. Finally, the appearance of a rightmost < character indicates that the values of the arguments to the recipient in a multi-recipient message macro (one with a >>) are to be computed based on the recipient itself, e.g., ->><, -->><, >-->><. As we've already mentioned, macros such as ->, ->>, and ->>< are not legal for agents to send to other agents, since these are synchronous message-sending macros. Instead, they must use, respectively, -->, -->>, and -->><.

We have implemented a fairly straightforward locking mechanism in our environment. The locking mechanism is designed to prevent collision among multiple processes, dispatched by the same agent, which may potentially read or write to the same slot of that agent. We have only a single flavor of lock, so, in particular, we do not distinguish between read locks and write locks. Our locks have *keys*. Access to a locked slot without the proper key fails. By default, each process is created with a unique key, but keys are data objects that can be passed between processes, say, for example in messages or in the slots of an agent.

When a process wants to lock a slot, it calls the function LOCK-SLOT and presents a key as an argument to the function. If the slot is currently unlocked, then it becomes locked, and the key which was presented is the only key that will open the lock until such time as the slot becomes unlocked again, guaranteeing that no other process will access the slot while it is locked. If the slot is currently locked, then the key will open the lock if it matches the key to the current lock (as will often happen if the slot is locked multiple times within the same process due to embedded calls to functions that want to lock the same slot). The caller of lock-slot can specify whether a lock failure is to return immediately or be queued; the queuing can either be to arrest the callers process or to invoke a caller-provided continuation. When a process is through with a slot, it unlocks it by calling UNLOCK-SLOT. This decrements multiplicity of the lock. When the multiplicity goes to zero, the slot is unlocked and the first process in the queue of processes waiting for the lock removed from the queue and invoke.

Naturally, the world is simplest when each locking has a matching unlock. Toward this end we have the macro WITH-LOCKED-SLOT, which locks a slot for the duration of its body, and unlocking the slot when the body exits, unwind- protecting against non-local exits.

Although debugging remains difficult in our environment, we have developed at least one debugging tool without which debugging our system would have been next to impossible: the message log. The message log is quite simply a log of messages in chronological order. The log reflects exactly what message was sent, the arguments to the message, the process which sent it, and the time at which it was sent. Any message sent by the system using a logging macro (one with a leftmost ¿ character) will be logged in the message log at the time the message is sent. Using functions we have developed over the message log, one can follow the logical pathways of a dialogue from agent to agent

6.5 Agent Algorithms

In this section, we describe the various algorithms for machines and order-handlers. We have not experimented with "clever" machines—our machines merely respond to order-handlers (though one could envision systems in which the machines would be more proactive). Order-handlers, on the other hand, might employ a multitude of different scheduling strategies. We have implemented seven different types of order-handling algorithms: simple, greedy, banded, banded-iterative, banded-random, banded-iterative-sliding, and economic.

6.5.1 Simple order-handlers

The simple order-handlers implement the simplest possible algorithm—they avoid searching if a feasible solution is at hand. More specifically, a simple order-handler will start with the first task which must be scheduled, find a machine on which it can schedule the task, and then attempt to schedule its next task, until it reaches the last task. When the last task has been scheduled, a simple order-handler stops scheduling. Thus, a simple order-handler does not search for the best reservation in any particular instance, but only for a valid reservation, nor does it attempt to better any existing reservations by rescheduling.

Surprisingly, even this simplest of algorithms cannot be written in a simple, straightforward manner. What does a simple order-handler do if one of its previously scheduled tasks is suddenly canceled by the machine on which it was reserved? It must start rescheduling back from the point of the now unscheduled task. This entails canceling reservations for all subsequent tasks. Further, since the algorithm must be flexible enough to back up, i.e., to work again on task numbers which are smaller than the task number that is currently being worked on, the possibility exists that the simple order-handler will at some point have two processes working on the same task. Even the simple order-handler algorithm, then, must have some logic in it to make sure that any particular execution of the algorithm is sponsored. A very simple way to implement sponsorship for this algorithm is to let the "sponsor" be the least-numbered task which still needs to be scheduled. Thus, for example, if a process were dispatched attempting to schedule task 6, it would first check to see if task 6 were the last-numbered unscheduled task. If it finds that task 4 is also unscheduled, then it would conclude that it (the process) is not sponsored, and immediately die. This trick reflects the idea of writing context-independent algorithms, because such an algorithm can be dispatched in a process at any time without harm, for, if the process should discover that it is not sponsored, then it simply dies.

6.5.2 Banded order-handlers

The banded order-handlers implement a slightly more intelligent algorithm than the simple order-handler algorithm. While still progressing sequentially through its list of tasks, as a simple order-handler would, a banded order-handler will attempt to "shop around" a bit for the best reservation, rather than being content with the first one it can find. A banded order-handler attempts to schedule a task by requesting reservations for it on B different machines, where B is the bandwidth of the order-handler. The order-handler will then accept the best of these B reservations, canceling the remainder. For example, the order handler H might need to schedule the task S, and there might be seven machines which can perform the operation for S. If the bandwidth of H is 3, then it will randomly select three of the seven machines, and send reservation request messages to each of them. Suppose the machines which H selects are M_2, M_5, and M_6. Machine M_5 might be the first to respond, with a reservation for S at time 6. Then machine M_2 might respond, with a reservation at time 7. Since H's existing reservation for S (at time 6) is better than the new one coming in, H will keep its current reservation, and send back a cancel reservation message to machine M_2. Machine M_6 might then respond with a reservation at time 3. Since this is better than the existing reservation, H will keep this new reservation, and send a cancel reservation message to M_5. While H is waiting for further replies to its messages, it will go ahead and continue attempting to schedule the next task based upon the currently best reservation that it has. If it should then subsequently receive a better reservation, H will attempt to reschedule the next task, etc.The banded order-handler stops when it achieves a valid schedule, in contrast to possibly attempting to go back and improve its existing reservations.

6.5.3 Greedy order-handlers

The greedy order-handler is a banded order-handler with infinite bandwidth. That is, it attempts to schedule itself on every machine that can handle a particular task. (At the opposite extreme, the simple order-handler is very similar to a banded order-handler with bandwidth 1.)

6.5.4 Iterative banded order handlers

The *iterative banded order handlers* attempt to improve upon the basic banded algorithm by virtue of sheer repetition. After an iterative banded order-handler achieves a valid schedule, it attempts to better each of it reservations, starting back again from the beginning, by sending reservation request messages anew. This technique is guaranteed to do at least as well as the regular banded algorithm, since no order-handler will ever accept a new reservation that is worse than the existing one. There is in fact some reason to believe that it will do substantially better than the single-pass banded algorithm, because presumably at any time any given order-handler, in attempting to schedule a task, will find that many desirable time blocks have already been reserved by other handlers, many of which reservations will eventually be canceled because they are inferior to some other reservation. When an order-handler schedules the last task in its list, it therefore makes sense to look back and see if some time blocks which had been reserved before are now free. This algorithm works best in the absence of bottlenecks. Obviously if there is only one machine which can perform a particular operation, then there will be no duplicate reservations to cancel. Since a task scheduled on a bottleneck machine has no hope of improving under this algorithm, no task which occurs before the bottleneck task can be improved to any benefit (since improvement only manifests itself when it can be passed down to the last task, and this is impossible when a bottleneck task stands in between).

There are two kinds of iterative banded order-handlers: random and sequential. The random order handler will perform its "first round" as a normal banded order-handler, after which it will randomly select tasks to improve. The sequential iterative banded order-handler always begins at the beginning and moves sequentially toward the end in its attempts to improve its tasks. The random order-handlers, while being less efficient, may have a global emergent behavior which is advantageous: namely, the avoidance of any deleterious lock- step phenomena.

6.5.5 Sliding order handlers

The concept of sliding in an order-handler algorithm is this: that after one has done the best that one can do, one might as well make life easier for one s fellow order-handlers by rescheduling all of one's tasks except the last task to be as late as possible, since it is only the reservation-time of the last task which really matters. By "sliding" its all-but-last reservations to the right on the time line, an order-handler frees up valuable time blocks to the potential advantage of other order-handlers, while in no way diminishing its own performance. This implies that all sliding order-handlers have two distinct modes of behavior: a selfish mode and a generous mode. In selfish mode, an order-handler tries to improve its own lot; whereas, in generous mode, an order-handler tries to improve the lot of others. It is obviously optimal to have about half of the order-handlers in each mode at any particular time, so that the selfish ones can take advantage of the generosity of the generous ones. It is important to emphasize that these order handlers are generous, but not altruistic: the last task of an order-handler never gets slid right. It is only allowed to slide left. This guarantees a monotonically improving situation for each order-handler. We implemented one version of slider: the iterative banded slider. This is just the iterative banded algorithm modified so that, once a complete schedule is achieved, the order-handler has an even chance of being selfish or generous on the next iteration. If the order-handler is selfish, then it schedules exactly as the iterative banded algorithm would schedule a new iteration. If it is generous, then it implements the banded algorithm in reverse, attempting to schedule first the next-to-last task as poorly as validly possible (we don't want to schedule it after the last task), then to schedule the task before that as poorly as validly possible, etc., until it schedules the first task. At that point the generous iteration is complete, and it has an even chance again of being either selfish or generous on the next iteration. While the complementary principles of selfish and generous behavior seem intuitively appealing, this algorithm has at least two shortcomings: it is inefficient, and it is not very effective against bottlenecks. To see how it is inefficient, consider an order-handler that has just finished a generous iteration and which is about to begin a selfish iteration. Clearly, by definition, it has just made its situation as difficult to improve as feasibly possible, since it has given up all of its good reservations for poor ones. One might therefore expect the sliders to do a lot of useless thrashing. Bottlenecks pose the same problem as they do in the normal iterative algorithms: a task which is scheduled on a bottleneck machine cannot be slid for the very reason that all of the time blocks both to the left and right of it in time are booked. The presence of this non-slidable task in the order-handler's task list prevents the order-handler from being too selfish in exactly the same way it did before: tasks behind it cannot be moved to the left passed it. Similarly, tasks to the left of it cannot be moved passed it to the right, preventing the order-handler from being too generous.

6.5.6 Economic models

The economic order-handler is by far the most complex in its algorithm. Economic order handlers must pay machines for their time. Every order handler begins with a fixed pot of funds, which it must wisely apply to the problem of scheduling its tasks. Machines publish price-lists which reflect how much any time block is worth, and they are constantly updating these lists according to the demand for their time. The economic system we have implemented is based upon the concept of a preemptive auction: unlike the case with all the other order-handler/machine environments, in the economic-order- handler's world, no reservation is secure. All time blocks on all machines are always available for purchase, regardless of whether or not they are already owned by an order-handler. Any other order-handler may steal away an order- handler's reservation by paying more money to the machine which granted it. When this occurs, the old reservation is said to be "bumped," and a message is sent to the unfortunate first-reserving order-handler that its reservation has been canceled due to a bump. An order-handler whose reservation is "bumped" will also receive a refund which includes the price originally paid for the reservation, plus some extra amount, as recompense for the inconvenience of having been bumped.

The overarching reality of the economic system is that all transactions which involve reservations now also involve some transfer of funds. An order-handler must pay to make a reservation, and an order handler is entitled to a refund when it cancels a reservation, or a reservation is bumped by another order- handler. These two refunds are different. Whereas the bump refund will be in excess of the amount originally paid for the reservation, the cancellation refund will be less. The cancellation refund will be a percentage of the reservation price, and the bump refund will be a percentage of the bumping (new, higher) reservation price. Thus, an order-handler that originally paid 100 monetary units for a reservation might receive a refund of 90 monetary units if it cancels that reservation, (90reservation is bumped by another order-handler for 200 monetary units (70of the bumping reservation price). Depending upon the values one assigns to the bump refund and the cancellation refund, one creates an economic system with more or less volatility. Obviously a desirable property of any scheduling system is some kind of a convergence to a solution. We would like our economy to converge as quickly as possible without hindering the order-handlers so much that they cannot achieve reasonable schedules. Convergence means that all of the order-handlers achieve a valid schedule, and reach a point where they cannot afford to do anything besides sit on their schedules peacefully. To that end, we would like the funds of each order-handler to be monotonically decreasing with time. Such a property would guarantee quiescence, if not convergence. That is, it would guarantee that the order-handlers all reach a point at which none of them can afford to make any more transactions, regardless of whether or not they are all scheduled. This is not possible however, unless one is willing to lower both the cancellation- and bump-refunds to 0. As long as an order-handler is refunded some positive amount for a canceled or bumped reservation, its account can go up.

A somewhat more reasonable condition is to require that the sum of all the order-handlers accounts be always monotonically decreasing. This embodies the idea that when order-handlers interact with each other via bumping, the net result be always a flow of funds from the order-handlers to the machines. That idea plays a part in the solution to attaining monotonicity, but it is still possible at any time that all the order-handlers would cancel a particular reservation, resulting in a net increase of funds for each order-handler, and hence for the sum. The quantity, as it turns out, which can always be made to decrease, is the sum of the worth of all the order-handlers, where the worth of an order-handler is defined to be the sum of its cash and the prices that it paid for its reservations. Even though an individual order-handler's worth may still go up, since another handler may bump one of its reservations, causing the handler to receive in cash more than it originally paid for the reservation, still the total worth of all order- handlers can be made to decrease, so long as the bumping order-handler pays enough to the machine to create a net negative cash flow. Clearly the case of a bump is the only interesting one, since whenever an order-handler cancels a reservation its worth obviously decreases, and whenever an order-handler makes a new reservation its worth obviously stays the same, and neither of these actions affects the worth of any other order-handler.

Here is the way that a machine in our economic system sets prices: The cost of reserving time from i to j on a machine is simply the sum of the costs of reserving time on all of the time blocks individually. The cost of any single time block is determined as follows: if the time block is owned, which is to say that it is currently part of a reservation that was granted to some order-handler, then the price of that time block is equal to the price which the order-handler that owns it paid for it, multiplied by some factor called the PRICE-INCREASE-FACTOR, which we have set in all our experiments to be 2. So, for example, if H_1 paid 400 for time block 3, and if time block 3 is still currently owned by H_1, then its current price, to any would-be bumper, assuming a price-increase-factor of 2, would be 800. As soon as a time block becomes free (no longer owned), then its price immediately becomes the price that was last paid for it, and this price begins to decay exponentially according to the factor PRICE-DECREASE-FACTOR. For example, if at time 100 H_1 were to cancel its reservation for time block 3, then time block 3 would become free, and its value would immediately be 400 (not 800). If the price-decrease-factor were .10, then at time 101 the price of time block three would be 360, and at time 102 its price would be 324, etc, until the time block either becomes owned, or it reaches the value ROCK-BOTTOM, which might, for example, be 10. All time blocks start out initially at a default value of INITIAL-VALUE, which might be, for example, 100.

The above pricing system has some desirable properties: when the demand for a time block is high, the price for that block goes up (exponentially). When a time block is free, the price of the time block drops (exponentially) until some order-handler purchases it. Thus, the pricing system is designed to be responsive to demand. The hope is that such a system will quickly reach an equilibrium state, where all of the time block prices are near their "true" values.

Now that the economic environment has been explained, we can go into the actual algorithm for economic order-handlers.

Every economic order-handler starts out with the same amount of funds, say 1000. An order-handler must decide how to best apply its funds to the problem of getting all of its tasks scheduled. The economic order-handler strategy that we implemented is rather conservative. It considers the total amount of unscheduled machine time that it must schedule (the sums of the durations of its unscheduled tasks), and, dividing this into its remaining funds at any time, calculates the average amount of funds it can allocate to the scheduling of any one time block. It will never pay more than this for any time block. So, for example, if $\mathtt{OH}_1$ had a total of two tasks, neither scheduled, the first with a duration of 3, and the second with a duration of 1, then the total machine time to be sc.ieduled would be 4. When $\mathtt{OH}_1$ begins, it has 1000 monetary units at its disposal, and the average amount that it can spend per time block is $1000/4 = 250$. Thus, $\mathtt{OH}_1$ would allow itself to spend up to 750 to schedule task 1, and up to 250 to schedule task 2. Once all of its tasks are scheduled, an economic order-handler will use its remaining funds in an attempt to better its existing schedule.

The economic order-handler employs a basic banded strategy. That is, it will send messages to B machines, where B its bandwidth, and accept the reservation response that it gets. There are two differences here from the basic banded strategy. First, best does not always mean earliest. And secondly, an economic order-handler will wait until it has received all of the responses to its messages before attempting to schedule the next task, unlike the non-economic banded strategies. This latter is because canceling reservations is in general too costly to allow for indiscriminate reserving and canceling of reservations. Further, time is no longer of the essence in the economic world. In the non- economic world, the first one to a time block got to keep it, and no other handler could take that time block away. There was therefore a premium on sending lots of messages and reserving lots of time blocks. In the economic system, any order-handler can bump an existing reserved time block by simply paying enough more for it (we should elucidate here that if any time block belonging to a reservation is bumped, then the entire reservation becomes bumped, even though not all of the time-blocks in the reservation were bought-out by the bumper). Thus, there is no premium at all upon buying early, because early buyers at cheap prices will simply be bumped by later buyers at higher prices. There is in fact some advantage to buying at just the point at which no other order-handler is willing to pay double. For example, suppose $\mathtt{OH}_1$ buys time block 4 for 100. There may be many other order-handlers who would also be willing to buy it for 100, but none willing to buy it for 200. The effect of this boundary phenomenon can be lessened by making the price-increase-factor lower than 2 and closer to 1 (however, this can drastically increase the time it takes for prices to converge to their true values). Most of the advantage to an order-handler in the economic system comes from spending one's money wisely, not from acting hastily.

Getting back to the first difference between the economic banded strategy and the non-economic banded strategy, best does not always mean earliest when it comes to reservations. Best means best value for one's money. For example, an order handler might be able to obtain a reservation at 3 for 100 monetary units, or a reservation at 4 for 10 monetary units. In most cases, the reservation at 4 would be a better value, since it is so much cheaper. This gives the order- handler more money to work with in attempting to schedule its other tasks.

In order to determine the value of a reservation at a particular price, the order-handler compares the price of that reservation with its internal assessment of the worth of that reservation. The ratio of these two values is the value of the "deal" it is getting for the reservation at the price that the machine set for it. For example, suppose a machine quotes 100 as the price for a particular reservation at time 3. and 10 as the price for a particular reservation at time 4. And suppose that the order-handler internally rates the worth of a reservation at 3 as 50, and a reservation at 4 as 40. Then the best-value utility of the reservation at 3 is 100/50 = 2, and the best-value utility of the reservation at 4 is 40/10 = 4. Thus, the reservation at 4 is a better value for the money.

Order-handlers calculate their internal assessments of the worth of a reservation based upon the maximum amount they are willing to pay for that reservation (which, we have already explained, is based upon the average amount per time-block which they can spend) and upon the earliest start time that the reservation is desired. For example, if task 1 of $\mathtt{OH}_1$ is scheduled and finishes at time 2, then the earliest start time for task 2 would 3 (since any earlier time would conflict with task 1's reservation). If 250 is the most that $\mathtt{OH}_1$ would be willing to pay for any reservation for task 2, then the value of time 3 for task 2 is judged to be 250. From there, the assessed value decreases linearly in such a way that the price of the 10th best start time, time 12, and all thereafter, are assessed to be worth ROCK-BOTTOM. This method of assessment is somewhat arbitrary, but the idea is that, regardless of what time early-time is worth, time early-time+10 is so bad that it is surely worth rock- bottom. Given those two datapoints, we need some continuous function through them. The simplest is linear, though one could imagine a function that fell off more steeply near early-start than near early-start+10, for it seems reasonable that the difference in worth between time 3 and time 4 should be greater than the difference in worth between time 11 and time 12.

Once an economic order-handler is in possession of a complete schedule, it is at liberty to use its remaining funds to attempt to improve its schedule. It does this by searching for large gaps in its schedule and attempting to compact them. For example, suppose the current schedule for $\mathtt{OH}_2$ were:

Task	Operation	Start time	End time
1	cut	0	2
2	bend	10	11
3	twist	14	17

There is a gap of 7 between tasks 1 and 2, and a gap of 2 between tasks 2 and 3. $\mathtt{OH}_2$ would, in this situation, choose task 2, as the largest gap task, to begin improving. It would try to reschedule task 2 as close to time 3 as possible. But this time, rather than using the average amount per unscheduled time block (which is infinite, since there are no unscheduled time blocks remaining), the order-handler would use the cancellation value of task 2 plus up to 75% of its remaining funds to reschedule task 2. Once again, the number 75 is rather arbitrary, but the idea is that the order-handler should be able to use almost all of its funds for each task that it attempts to improve, since it has little else to spend its money on once all its tasks are scheduled. Some money should always be left in reserve, however, to deal with the troublesome occurrence of a bump.

We have not yet mentioned how bumps are dealt with by the economic algorithm. They turn out to be troublesome indeed, for a bump will often necessitate not only the rescheduling of the bumped task, but of several tasks immediately following it. And each of these successor tasks must be canceled, at a loss, which might turn out to be greater than the refund which the order- handler received for the bumped reservation! Thus, being bumped can be a disaster all the way around. This is one reason that the price for bumping, the price-increase-factor, was set to 2. A bumping order-handler must pay double the price in order to bump the reservation.

The economic algorithm is written in a truly context-independent fashion. It has a top loop which continually examines the state of its schedule, and which always decides to work on the least-numbered unscheduled task, if one exists. Thus, a bump per se is not recognized by the algorithm at all. It will simply note the existence of an unscheduled task and behave accordingly. Tasks downstream of the unscheduled task are not immediately canceled, because the algorithm will surely attempt to reschedule the bumped task in the very same time block that it was bumped from, in which case all of the downstream tasks would still be valid. If that turns out not to be the case, and the bumped task were scheduled later than it had been, causing its end time to conflict with the start time of its successor task, and perhaps other downstream tasks as well, if it were rescheduled significantly later than before, then all of the conflicting downstream tasks would be canceled.

Unfortunately, this particular economic algorithm did not turn out to be viable. It is possible that it contained programming errors which were not uncovered (and this is highly likely, since so many errors in the original algorithm were uncovered), or it is possible that the algorithm functioned "correctly," but in any case it is clear that its emergent behavior was disastrous. Here is what happened:

One nominal virtual of a multi-agent approach is the ability to focus energy and quickly converge to a solution. However, the economic algorithm ran for days, rather than hours, and still never reached quiescence. The longest algorithm other than the economic was the the sliding algorithm. The sliding algorithm took about three hours to complete, whereas the economic algorithm was allowed to run for three days without reaching quiescence. Upon closer examination, we discovered that various order-handlers were attaining complete schedules, only to have their schedules destroyed time and again by another handler's bumping one of their reservations. Some order-handlers, with few tasks, were able to secure stable schedules, since they were able to spend more money per task than order-handlers with many tasks. This allowed them to bid up the price of their reservations to a point where no other order-handler was willing to bump them. Order-handlers with more tasks were not so fortunate. They tended to get into bumping wars in which their schedules became progressively worse. Consider, for example, two handlers, $\mathtt{OH}_1$ and $\mathtt{OH}_2$, each with a complete schedule and no funds left. They will both be in an improvement phase, in which they attempt to obtain better reservations for already-scheduled tasks. The algorithm allows them to attempt this, even though neither has any money, since they are allowed to use the cancellation refund for any task in order to buy a better reservation. Suppose that $\mathtt{OH}_1$ is able to improve one of its tasks by bumping a reservation belonging to $\mathtt{OH}_2$. $\mathtt{OH}_2$ will be unable to buy the reservation back, since it will now cost twice what $\mathtt{OH}_2$ originally paid for it, and the bumping refund will not be that much. So $\mathtt{OH}_2$ will have to reschedule it somewhere else. Suppose it does this, and obtains a significantly worse reservation. This may necessitate canceling several reservations downstream of the bumped task, which results in a net loss of worth for $\mathtt{OH}_2$. Suppose that $\mathtt{OH}_2$ must cancel three tasks as a consequence of the bump. It must now reschedule those tasks with the refunds it received for their cancellation. And suppose that $\mathtt{OH}_2$ reschedules them in such a way that two reservations of $\mathtt{OH}_1$ become bumped. *$\mathtt{OH}_1$ will now find that its previous* bump has actually boomeranged. Although the attempt was to make its schedule a little bit better, $\mathtt{OH}_1$ will find that it cannot repair the damage done to its schedule by these two bumps without making its schedule worse. For it will suffer a net loss of worth,due both to cancellation refunds and to the fact that the old reservations are now twice as expensive as they used to be. $\mathtt{OH}_1$ will have to buy new reservations at cheaper–and hence later–times. But in buying new reservations, $\mathtt{OH}_1$ may very well bump some reservations belonging to $\mathtt{OH}_2$, and so the cycle begins all over again.

The bump is thus a sinister device in this case, since it is an attempt to improve one's own situation to the detriment of another's. In the above example, every time the bump is employed, it immediately benefits the bumper a little bit, and immediately hurts the bumpee a whole lot. Thus, it has a globally deleterious effect. This is reminiscent of the results of price-wars. The problem, of course, is that the entire preemptive auction protocol is based upon the bump, and one can no more alleviate the problem with this system by forbidding bumps than one can alleviate price-wars in a capitalist system by forbidding competition. It is only by means of bumping that prices achieve their true values. Further, the bump is actually a means of alleviating the problem of first-come-first-serve unfairness that is inherent in all of the previous algorithms, for, while there is no basis for believing that the first order-handler to attain a reservation at a particular time has any particularly valid need for that reservation time, there many reasons to believe that the one who pays the most for a reservation time has the most valid need for that reservation time.

6.6 Measurements

In this section, we discuss the relative performance of the order-handling algorithms (with the exception of the economic algorithm, which never achieved a completed schedule) with respect to the two measures SUM and MAX. We tested these algorithms on three different machine shops: Shop1, Shop2, and Shop3. The machine-types for each of these three shops were: bend. coat, cut, deburr, drill, lathe, punch, polish, twist, for a total of 9 different machine types, and each machine type was represented by at least one machine in each of the three shops, with a random distribution among the machine-types. Each order- handler in each of the three shops had an average of five and a half tasks, with each task having an average duration of two time blocks. All of the banded scheduling strategies used a bandwidth of 2 (which we empirically determined to be optimal for the machine shops under consideration), and all of the iterating strategies used an iteration-level of 10.

Shop1 consisted of twenty machines, and forty order-handlers. There were two bottlenecks in this shop: one drilling machine, and one bending machine.

Shop2 consisted of twenty machines and twenty order-handlers. Since this shop had approximately half the scheduling load as shop1, the order- handlers had about twice as much "elbow room" to get their tasks scheduled as the order-handlers in Shop1.

Shop3 consisted of forty machines and eighty order-handlers. Thus, it was double the size of Shop1. The intent here was to see if any of the algorithms, which might have been effective on a smaller shop, became bogged down when the size of the problem doubled.

Each scheduling-strategy was run ten times on each shop. So. for example, in Shop1, first every order-handler was given the SIMPLE scheduling strategy, and the agents were set in motion until the shop became quiescent (no more messages being sent by any agent). Then the SUM and MAX measures were applied to the resulting schedule and recorded, along with the total number of messages that were sent within the shop. This process was repeated ten times, so that ten different scheduling datapoints were obtained for the SIMPLE strategy. Then the scheduling strategy of all of the order-handlers was changed to BANDED, and the problem run again ten times, etc., until ten datapoints were collected for each type of scheduling strategy. This testing regimen was performed on each of the three machine shops. Figures 6.6, 6.6, and 6.6 show scatter plots of the various algorithms in each shop.

The statistics have some surprises: namely, that some of the so-called "improved" algorithms actually performed worse than their simpler antecedents. Most noticeable is the poor performance of the greedy scheduling strategy. Of the non-iterative algorithms, the greedy would seem to be a likely candidate to be the one having the best performance, since a greedy handler is always guaranteed of obtaining the best of the available reservations for any task at the time that it attempts to schedule that task. The simple strategy, by contrast, might well miss better reservations for a task, since it only attempts to schedule any task on one (out of many possible) machines. Nevertheless, the greedy strategy is clearly the worst performer of all the strategies, while the simple strategy on average performs quite a bit better. A reasonable explanation for this result is that every greedy order-handler effectively puts "out of commission" as many time blocks as it can, relative to each task which it must schedule. Although it will later cancel unneeded time blocks, other handlers may never reclaim them, since this is a non-iterative strategy.

The iterative strategies all perform noticeably better, with the sequential banded iterative strategy being the clear winner across all of the tests. There is also a surprising result here, which is that the sliding strategy does not generally perform as well as the sequential iterative strategy, even though it was supposedly an improvement upon this strategy. Though we pointed out some of the potential flaws of this strategy earlier, namely, its ineffectiveness against bottlenecks and the somewhat self-defeating consequences of alternating between selfish and generous behavior, we might nevertheless expect it to perform at least a little better than the sequential strategy, and certainly not worse. The reason for this expectation is that no generous iteration will ever cause a sliding order-handler to worsen its contribution to either the SUM or the MAX metrics, yet by being generous, a sliding order-handler will allow other order-handlers to better their contributions to the SUM and MAX metrics. One likely explanation for the ineffectiveness of the sliding strategy would be that, if the generous iteration of the order-handlers were largely ineffective in allowing the selfish-iteration handlers to better their schedules, then. the net result would be that essentially all the order-handlers would behave as (non-sliding) sequential order-handlers, with only half of the iterations. Thus, in comparison to a 10-iteration sequential order-handler, the 10-iteration sliding order- handler would behave as though it were a 5-iteration sequential order-handler. It is quite conceivable that the sliding scheduling-strategy would beat the sequential strategy if its iteration-level were doubled or tripled.

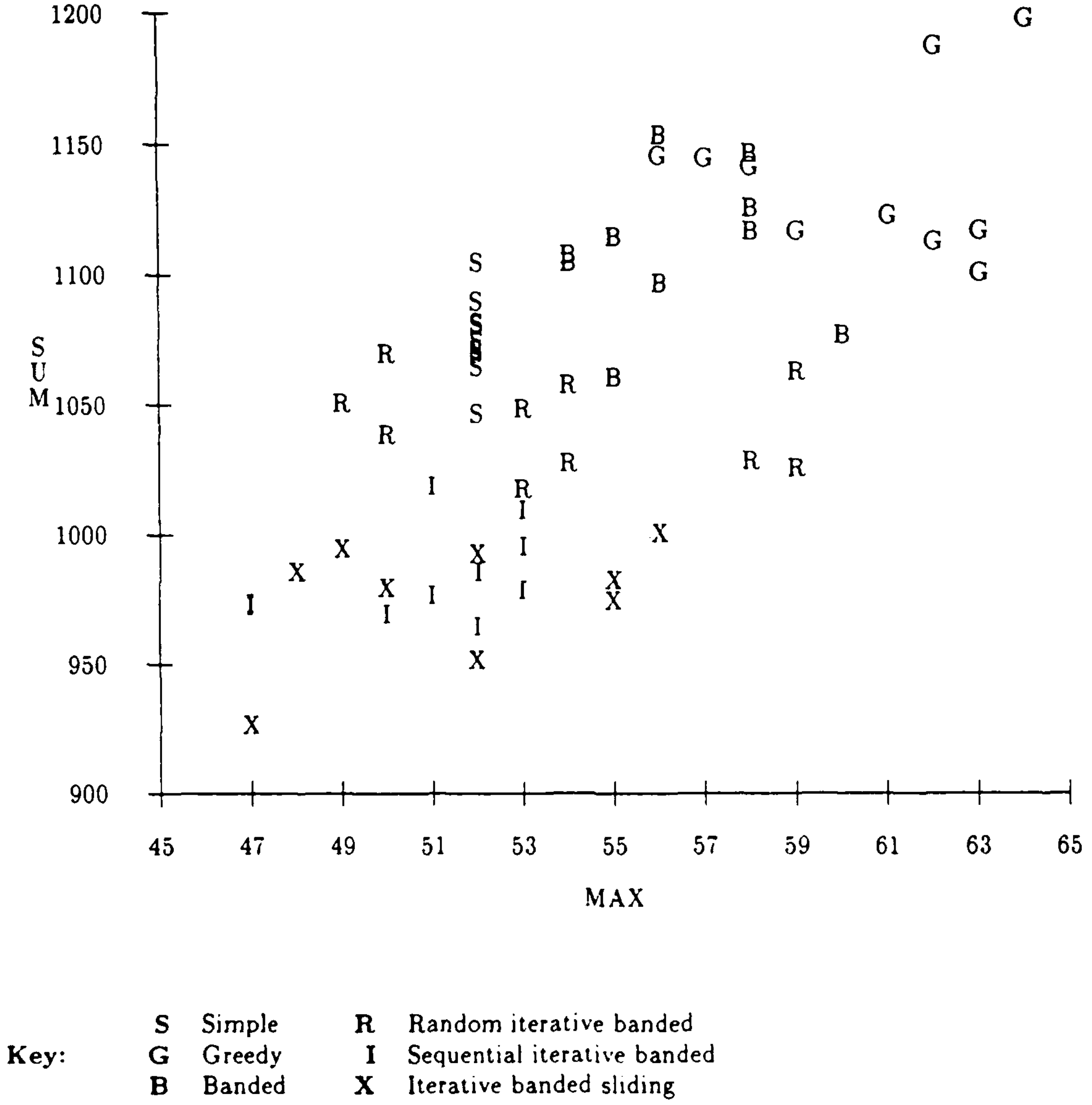


Figure 6.1: Sum-max scatter of Shop1—40 Order handlers, 20 machines

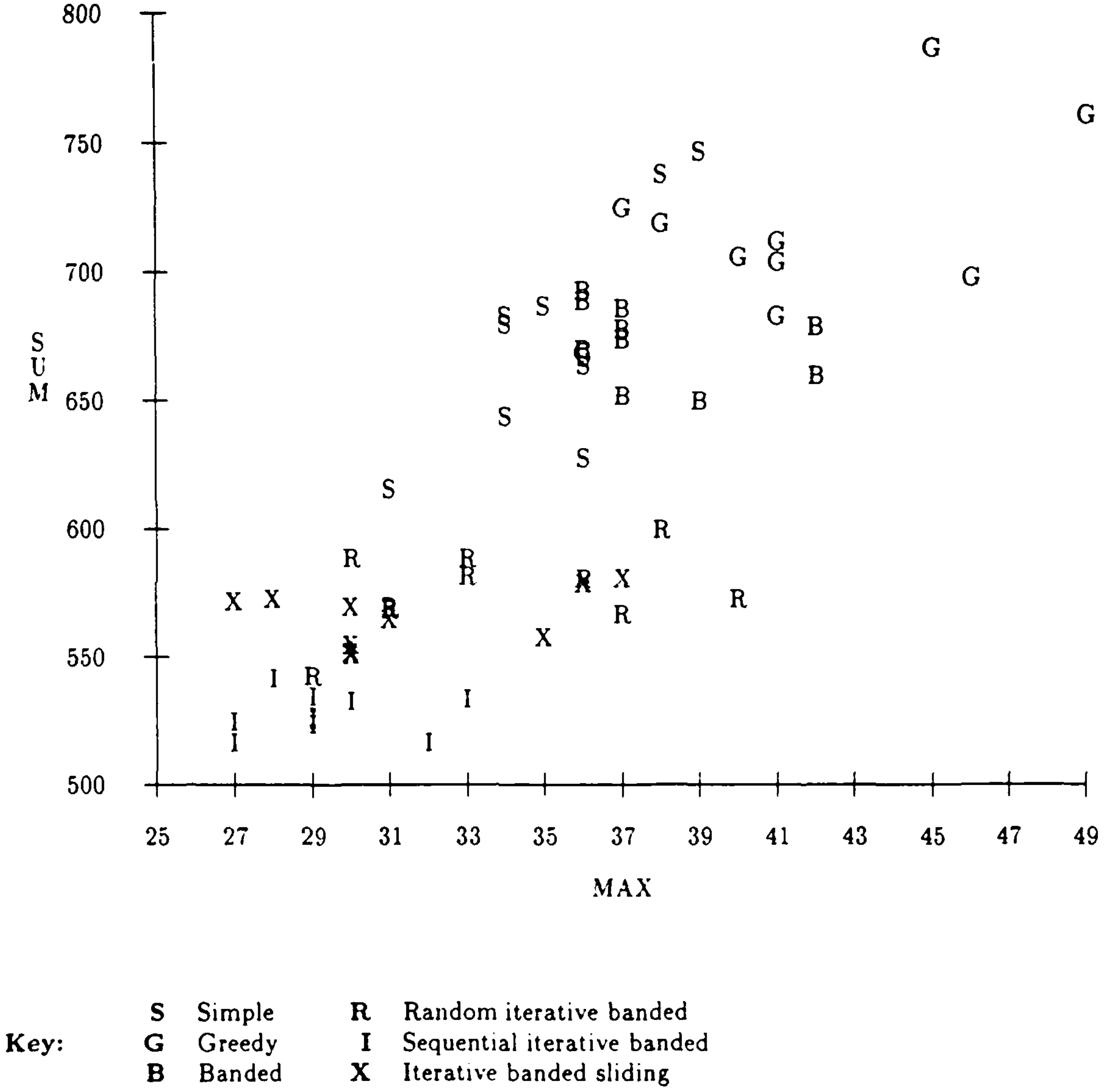


Figure 6.2: Sum-max scatter of Shop2—20 Order handlers, 20 machines

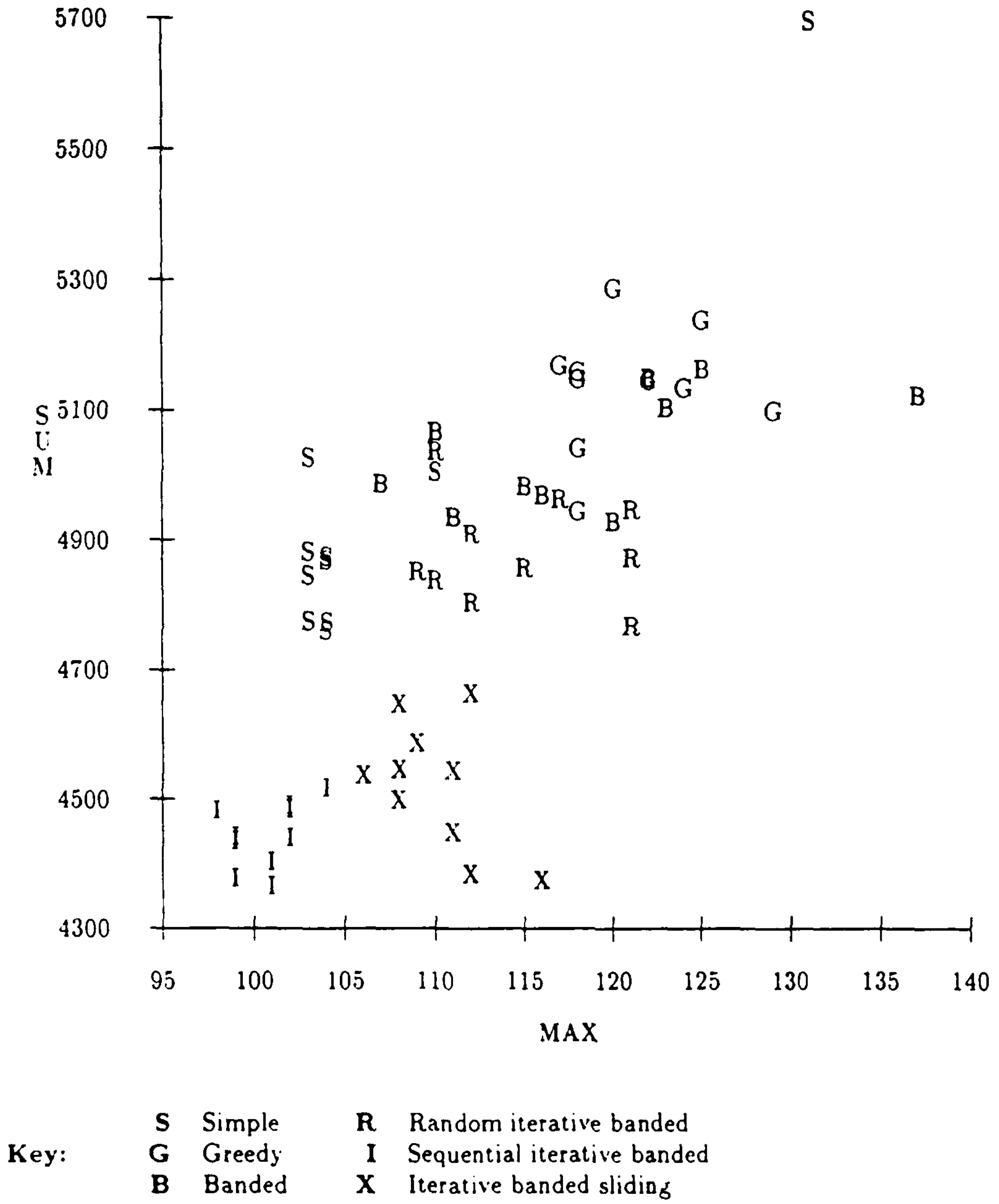


Figure 6.3: Sum-max scatter of Shop1—80 Order handlers, 40 machines

6 7 Conclusions

The one thing that can unequivocally be concluded about multi-agent systems is that their algorithms are difficult to design correctly, even for the simplest of algorithms, and that the emergent behavior of interacting agents can be non-intuitive. Extensions to Lisp which included constructs that made it easier to recover context, or to examine the state of other processes, would definitely enhance the ease with which concurrent algorithms could be designed and verified.

Another unequivocal result is that the sum of a bunch of local goodness does not always equal global goodness, as is most notably illustrated by the greedy scheduling strategy, and by the bumping wars of the economic strategy. Further, the (desirable) constraint that multi-agent algorithms be "local," in the sense that they only have immediate access to their own situation, without a global view of the problem, tends to promote selfish strategies, i.e., strategies whose only concern is to make one's own schedule better, without regard to one's fellow order-handler, though it is at least possible for a local strategy to be somewhat altruistic, as in the case of the sliding strategy. It is an open question at this point whether a purely selfish strategy can yield a near-optimal global result, or whether the most effective strategies must always incorporate some concept of altruism.

Chapter 7

Inference in Parallel

Recent developments in massively parallel computing hardware offer the potential of achieving very high levels of performance at moderate cost for a wide range of software, including knowledge-based systems. An early generation of parallel extensions to existing languages makes it possible to explore this potential. We have done some of this exploration on the Connection Machine^{TM}, concentrating on algorithms for doing rule-based inference in parallel.

We regard rules as a promising candidate for parallel implementation because of their inherently non-deterministic underlying model of computation. When the premises of a rule are matched, all possible combinations of the matching facts are (at least implicitly) generated. It seems natural to consider all these combinations in parallel, since the programmer may know of no *a priori* reason to prefer any of them over another. When large numbers of rules (as opposed to large numbers of facts or combinations thereof) may be applicable to a problem, it is usually desirable to consider them in parallel, since they are useful (or inapplicable) independently of each other; much work has been done on techniques for choosing the most promising rules and using them first, but in most cases this represents no more than an attempt to squeeze the maximum efficiency out of serial hardware that intrinsically is not suited to the non-deterministic model of deduction from which rule systems spring.

We are aware that rule-based programmers often do rely on serial execution of rules, and even on one of a pair of very similar rules always being executed before the other, for correctness as well as efficiency. We are strongly in sympathy with the desire to combine non-deterministic search (as embodied in rule-based deduction) with fully deterministic execution (as embodied in most traditional programming languages; see Chapter 2 on Assertion Nets for more details). It will be shown below how our approach to executing rules in parallel does accommodate this desire, and moreover allows the separation of inherent seriality from efficiency-driven sequentialization.

We regard it as unfortunate that the most highly regarded algorithm for rule matching on serial machines, Rete, has been used as a candidate for parallel implementation. Rete is an inherently serial algorithm, based on the assumption that facts are added or deleted one at a time, and that the consequences of each addition or deletion must be explored before the next one can be processed. It is hardly surprising that attempts to obtain a high degree of parallel speedup for this algorithm have failed, since its success comes from squeezing through the von Neumann bottleneck as effectively as possible: it stores intermediate results to minimize the time spent re-computing them. We have chosen a less serial algorithm to build on, and believe we can achieve a speedup limited more by the size of the problem and the available hardware resources than by anything else.

We begin this chapter with a description of the Connection Machine and the *Lisp^{TM} language, since the reader may not be familiar with these. We then show how facts can be stored on the machine in a way that facilitates both ordinary access to them and matching of rule premises, exploiting the design strengths of the hardware architecture. Finally, we comment on how rules running on the Connection Machine might be used as part of an overall KBS.

7.1 The Connection Machine

The Connection Machine[1] is a massively parallel SIMD machine with high communication bandwidth between individual processors, receiving its instruction stream (and data) from a serial front end machine. Each instruction cycle causes all processors to perform the same operation, except that processors may "deselect" themselves and will then carry out no further instructions until they are reselected. Processors may communicate with "neighbors" at a defined offset from their own address, or with arbitrary other processors whose addresses they have computed (but these two modes cannot, of course, happen simultaneously). The manufacturer suggests that the machine be thought of as a large "active memory" wherein data can move around under the control of the front end, and can perform simple operations on themselves as they move.

The newest model of the Connection Machine, the CM-2, can be ordered with either 16,384; 32,768; or 65,536 processors, and there are optional extras such as floating-point accelerator chips. We make no use of these extras; the discussion below is largely independent of the size of a CM, except that when concrete examples are given, they may assume 65,536 processors. The algorithms described below were, for the most part, developed using $^1/_4$ of a 32k-processor machine, or 8,192 processors.

Languages supported by the CM-2 include *Lisp, an extension of Common Lisp with constructs that enable the programmer to use the CM-2 facilities.

7.1.1 Processors

Each CM-2 processor consists of a one-bit-wide logic unit, 65,536 bits of memory, some flag bits (including the flag which tells whether the processor is selected), and communication interfaces. The logic unit can perform any Boolean function from three input bits (two memory bits and a flag) to two output bits (one memory bit and one flag). Thus arithmetic, Boolean, and shift operations on integers (signed or unsigned) of arbitrary length are possible, but only within the processor's local memory. A field of the local memory, of arbitrary placement and size, can also be put into the communication network, to be delivered into another field at the destination processor.

[1] Connection Machine and *Lisp are trademarks of Thinking Machines Corporation.

7.1.2 Virtual processor sets

The physical processors of a CM have addresses ranging from 0 to one less than the number of processors. For computations where the number of data items exceeds the number of processors, the CM supports the concept of *virtual processors*, whereby the memory of each physical processor is divided up into smaller pieces, each of which is thought of as the local memory of one virtual processor. The execution of instructions is modified so that every virtual processor executes the current instruction if it is selected; this is done by having each physical processor execute it up to n times, where n is the number of virtual processors per physical processor, known as the *VP-ratio*. This ratio is always a power of 2.

Another valuable feature associated with virtual processors is the ability to treat the CM-2 as a multi-dimensional array of processors, rather than simply using one-dimensional addresses from 0 to 65,535. When a virtual processor set (or VP-set) is created, it can be given a list of dimensions, each being a power of 2 (the product of the dimensions must be at least the number of physical processors), and processors can then be addressed with corresponding lists of numbers. For example, one could define a VP-set with dimensions (16, 32, 256) and then refer to processor (13, 25, 101) rather than "manually" convert this to a hardware address. The communication system can use these addresses, thus greatly facilitating the programming of problems that have a "natural" multi-dimensional model.

Several VP-sets can exist side by side, with different sizes and dimensions. The virtual processors in different VP-sets do not share physical memory, but rather, a region of each processor's physical memory is allocated for each VP-set. One VP-set is always current; the current VP-set can be changed by the program. Processing occurs only within the memory allocated for the current VP-set. Communication with other virtual processors in the current VP-set respects the grid defined by this VP-set; it is also possible to communicate with virtual processors in other VP-sets (which may even reside on the same physical processors as the ones communicating with them, though this is transparent to the programmer).

7.1.3 Communication

Grid communication

There is special hardware support for communication between virtual processors in the same VP-set whose addresses differ only in one co-ordinate by a constant amount. This is known as *NEWS* (North, East, West, South) communication. For example, all processors can be instructed to send a certain value to the processor whose address is obtained by adding 19 to the 0'th co-ordinate of their own address. When this address would be off the edge of the grid, the message can optionally wrap around to the other edge, or be lost. This kind of communication can be used to move a data structure bodily, or to "reduce" or "expand" it along a chosen dimension (see below).

Random-access communication

A store-and-forward routing network, with buffering and load-balancing algorithms for handling collisions of messages, is available for general communication from any virtual processor to any other. The destination address may be one- or multi-dimensional, and can refer to a VP-set other than the current one. Since several messages may be sent to the same processor, options are available for the treatment of confluent messages: the values contained in them may be combined by arithmetic (add, max, min) or Boolean (and, or) operators, or one may be selected randomly and the rest discarded. There is also an option for signalling an error if the destinations of two messages coincide.

7.1.4 The *Lisp language

The *Lisp language consists of Common Lisp extended by a set of forms, functions, and macros for producing and handling parallel data, i.e. data that are stored in the local memory of each CM processor. Such data items are represented on the front end by a structure that specifies where the data are to be found in the CM and possibly what type they are.

Notation

The *Lisp extensions are mostly parallel analogues to Common Lisp constructs, and have names derived from theirs in one of two ways. *Lisp forms that produce a side-effect on some or all CM processors are named by prepending a * to the Common Lisp name, thus `*defvar`, `*setf`. Those that return a value on CM processors are named by appending `!!` (the two exclamations are intended to resemble a pair of **parallel** lines) to the Common Lisp name, thus `zerop!!`, `sqrt!!`. The special constants `nil` and `t` have analogues `nil!!` and `t!!`. Some *Lisp forms have no Common Lisp analogue; of these the simplest is the function `!!`, which when applied to a Lisp value of suitable type returns that value on all selected CM processors, so that `(!! t)` is the same as `t!!`.

References below to a *Lisp form acting on "each processor" should be understood to mean "each currently selected CM processor."

Parallel data

Dynamic allocation of the local memory of CM processors in the style of Lisp is not supported. But numbers, Boolean values, characters, arrays of fixed length (including strings), and defstructs having fields of these types (or rather, `*defstructs`) can be stored and manipulated on the CM processors, as can pointers to objects stored on the front end machine. The only way in *Lisp for a datum on one virtual processor to point to one stored on another processor is to use an explicit address or an "address object."

The parallel data structure used by *Lisp is the parallel variable or *p-var*. A p-var is declared with respect to a VP-set and space is allocated for it (the same amount of space on each virtual processor) in the region of physical memory belonging to that VP-set. If the p-var's type is not completely specified, then the amount of space required for it may grow as values are assigned to its elements. For example

```
(*proclaim '(type (p-var (unsigned-byte 2))) p-var-1)
(*defvar p-var-1 (!! 3) ''first p-var'' vp-set-0)
```

creates a p-var two bits in size, initialized to 3 in all virtual processors. An attempt to (*set p-var-1 (!! 5)) would result in an error. But if the *proclaim had not been done, p-var-1 would *initially* have taken up two bits on each VP, and would be expanded to 3 bits to contain the value 5.

Parallel computation

Computations carried out on individual CM processors using parallel variables and values can be expressed very similarly to serial Common Lisp. Functions that return parallel values, or require p-vars as arguments, are defined with *defun; other parallel constructs, such as *let, *when, if!!, *setf, and so on, have semantics analogous to let, when, if, setf.

Communication between CM processors is implemented by a set of functions with no Common Lisp analogues. If x, y, and other-address!! are p-vars of appropriate types, (pref!! x other-address!!) returns in each processor the value of x on another processor; (*setf (pref!! x other-address!!) y) transmits the value of y to the other processor and stores it in the location of x there. (*pset :no-collisions y x other-address!!) does exactly the same thing; the first argument to pset!! can promise that no two processors will write to the same one, or specify a mode such as :max for combining the values if this does happen. news!! and *news do much the same things as pref!! and *pset using the NEWS grid instead of random-access communication; both these take scalars, rather than p-vars, for the address offsets used to determine which processor is communicated with. The functions self-address!! and self!! return, in each processor, an address and an address object pointing to that processor; (self-address-grid!! n) returns the nth co-ordinate of each processor's address. Although this is not quite standard *Lisp, we will use (x!!), (y!!), (z!!) as abbreviations for the 0'th, 1st, and 2nd co-ordinates (where defined) of a processor's grid address. (*Lisp does support the use of :x, :y, :z as alternatives to 0, 1, 2 in some functions that take a dimension argument.)

Two other communication functions are of interest to us: one which replicates a parallel value along a specified dimension of the grid, and one which reduces (in the sense of the Common Lisp reduce function) a parallel value along a dimension. spread!! takes as arguments a parallel value and scalars representing a dimension and a co-ordinate, and returns in each processor the value of the p-var at the processor found by substituting the co-ordinate into its address at that dimension. Thus (spread!! a 0 5) replicates the values of a on the hyperplane $x = 5$ into the entire grid. scan!! takes a parallel expression, a combining function, and optionally a dimension; it returns in each processor the result of combining the values found in processors "before" it in that dimension. Thus the value at (33, 14, 4) of (scan!! a '+!! :dimension :z) would be the sum of the values of a at (33, 14, i) for i from 0 to 4.

Serial and parallel

The value of a p-var in a particular processor can be read to the front end by (pref p-var-1 1989) or (pref-grid p-var-1 5 18 3) (the addresses 1989 and (5, 18, 3) do not necessarily designate the same virtual processor). Certain combination functions such as *sum, *max, and *and return to the front end the sum, maximum, conjunction, etc. over all processors of the parallel value to which they are applied. For bulk transfer of data between the front end and the CM, functions pvar-to-array and array-to-pvar are available, which transfer data between a VP-set and an array of similar shape on the front end.

7.1.5 Performance considerations

The decoding of an instruction received by the CM-2 from the front end, and its distribution to the processors, takes as much time as several cycles of the processors' logic units. Therefore, operations on medium to large data items are more efficient than operations on small ones, especially single bits. If the VP-ratio is greater than 1, some of this overhead time is incurred once per physical processor, and can be amortized over several virtual processors. Therefore, working at a low VP-ratio can be less efficient than a higher VP-ratio.

Near neighbor communication (including the scan!! and spread!! operations that are based on it) is significantly faster than random-access communication. The latter also slows down, due to collisions, as the number of messages increases, unless the pattern of origins and destinations has regularities that can be exploited by the routing hardware. It is therefore important to avoid pref!! and *pset wherever possible, replacing them with judicious use of the other operations. For example, to transpose values stored in processors with addresses (i,0) to processors with corresponding addresses (0,i) it is faster to

```
(scan!! (and!! (=!! (x!!) (y!!)) (spread!! a :y 0)))
```

than to

```
(pref!! a (grid!! (y!!) (!! 0)))
```

In general, pref!! and *pset operations, where present, tend to dominate processing time. The slowness of these operations on large numbers of messages also argues against the use of high VP-ratios. Where a pref!! or a *pset is unavoidable, processors with redundant values should be selected out of the communication operation, and these values re-created afterwards by means of a spread!! or whatever other means are suitable.

Large values (hundreds of bits) are transmitted fairly efficiently. We attempted to speed up communication-intensive parts of our code in two opposite ways, namely by splitting few large messages into many smaller ones so as to get more parallelism, and by merging many small messages into few large ones so as to reduce the number of colliding messages; both attempts failed.

The highest VP-ratio that seems attainable on the CM-2 using Release 5.0 of the system software is about 1,024; the space available for p-vars is significantly reduced with this many virtual processors, as each VP seems to use 12 bits of physical memory to store its flag bits and other overhead items.

7.2 Representing Facts in Parallel

In this section we discuss how facts can be stored on the CM in such a way as to facilitate inference, especially rule matching. For expository purposes, we begin with a representation that is not very efficient, and then derive two beter ones from it. The relative efficiency of these two depends on the "density" of facts (which is defined below). It may be necessary to translate between representations, so we discuss the ways of doing this efficiently.

7.2.1 Representing objects

We assume that, for the purposes of storing facts in the CM-2's processors, a mapping has been established from the set of objects mentioned in our set of facts to a finite set of non-negative integers (preferably contiguous and starting at 0). The reason for this assumption, and the circumstances under which it can be relaxed, will become apparent shortly. It is extremely helpful if the predicates are strongly typed, with no intersection between the types: then objects of different types can map to the same integer with no confusion.

The assumption is not unrealistic for several KBS application domains with which we are familiar. In diagnosis, the sets of symptoms and of problems that may cause these symptoms are usually known when the system is started up, and rarely change during processing. In configuration, the set of parts from which a complete solution can be built, and (if these are represented as objects) the constraints between the parts, are given ahead of time. Scheduling problems often specify the sets of resources available, tasks to be scheduled, and time intervals over which to schedule them, with no possibility of unexpected additions.

As an example, we shall use a hypothetical university job fair, attended by 1,000 graduating seniors who have majored in 30 different subjects (double majors are allowed, but no triple or higher-order ones), and by 200 companies from all over the 50 states, seeking employees in 40 different job categories. The initial information about these entities is contained in predicates *Major(student, subject)*, *Relevant(subject, job)*, *Hiring(co, job, state)* and *WillGoTo(student, state)*. The companies are all small, and none of them does business in more than four states. Most of the students are quite particular about where they will relocate, but some are desperate for work and will go almost anywhere. Appointments for subsequent interviews are made according to the rule

$$\frac{\begin{array}{c} Major(student, subject)\ \&\\ Relevant(subject, job)\ \&\\ Hiring(co, job, state)\ \&\\ WillGoTo(student, state) \Rightarrow \end{array}}{Interview(co, student, job)}$$

7.2.2 Single bits

The germ of our scheme for representing facts on the CM-2 is quite simple. To represent facts containing a particular predicate, we create a VP-set whose dimensions are at least as big as the sizes of the domains of the predicate's arguments, and establish a natural 1-to-1 mapping between virtual processors and possible tuples of arguments to the predicate. Then we assign a bit in each virtual processor to say whether the predicate holds true of the corresponding arguments. The true instances of *Major* can thus be represented by a Boolean p-var defined in a 2-dimensional VP-set of size 1024 × 32 (or any larger size), simply by having the p-var take the value `t` at processor `(x,y)` iff student number x majored in subject number y. More generally, for an n-ary predicate we would use an n-dimensional VP-set.

The motivation for this representation (and others based on it) is that it turns many of the queries and updates that one usually wishes to perform into very simple operations for the CM. To query whether a fully ground fact is true, all that is needed is to compute an address and read one bit. A query with one variable unbound is done via a **scan!!** along the dimension corresponding to the unbound variable. Existentially or universally quantified queries translate to calls of ***or** or ***and**. Matching two facts on some argument becomes an address comparison. And in Section 7.3 we shall see that matching two entire predicates' worth of facts is also simple and quick.

7.2.3 Bit strings

As we observed above, single-bit Boolean values are inefficient to process; and as we shall see below, storing one fact per virtual processor can quickly lead to insupportably large VP-ratios. We can, however, store far more facts than there are virtual processors while retaining the basic idea of dedicating a bit in memory to each potential fact. This is done by abolishing one (or more) of the dimensions of the VP-set, and forming the bits that would have been spread out along this dimension into a bit-string (or unsigned integer, if you prefer) to be stored in a single virtual processor. So the predicate *Hiring*, instead of being represented by a Boolean p-var in a VP-set of size (256, 64, 64), could be represented by a p-var of 64-bit integers (representing sets of states) in a VP-set of size (256, 64); or actually 50-bit integers would do. Then the interest of company 71 in hiring someone to do job 15 in Alaska would be represented by the 49th bit of the value of this p-var in processor (71, 15). A VP-set of size (256, 64, 64) would take up 2^{20} or over 10^6 virtual processors; one of size (256, 64) or 16,384 is less extravagant.

7.2.4 Explicit representation

We define the *density* of a predicate to be the proportion of its bits (in either of the above representations) that are 1. The *Hiring* predicate is sparse, i.e. has density much less than 1, since no company operates in more than 4 states out of 50. This will be true of many predicates, including those that are really functions, or almost functions (in OPUS, slots whose maximum cardinality is small), and have a large co-domain (large valueclass). *Major* is another sparse predicate. Storing sparse predicates with one bit for every potential fact seems wasteful; it would be better to ignore the 0's and remember the locations of the 1's.

This suggests that we exploit the sparseness of a predicate by representing one or more of its arguments explicitly, storing their values in the CM, rather than implicitly by bit positioning. The explicit representation may continue to use the integers to which the argument values are mapped, or may consist of the actual values. If these values are numbers, or other entities that can be stored into a p-var, they can be stored directly on the CM; otherwise, the CM can, under Release 5.0 of *Lisp, store pointers to the values on the front end. An advantage of using actual values is that it avoids the need to determine the size of a domain and construct a mapping from it to the integers.

For example, we could allocate a VP-set of size (256, 64, 4) and a 6-bit p-var in this VP-set to store the *Hiring* predicate, and the fact that company 71 has a vacancy in job 15 in Alaska can be represented by this p-var having the value 49 at, say, (71, 15, 2). We call this a "semi-explicit" representation, since some of the argument values are represented implicitly by the location where a value is stored, and one of them by the value itself.

This representation actually saves little or no space relative to bit-strings. Assuming overhead of 12 bits per VP, a single VP with a 50-bit p-var takes 62 bits of physical memory to represent four facts; four VPs with a 6-bit p-var in each take $4 \times (6 + 12) = 72$ bits. Of course, one or other of these VP-sets may already have existed for some other purpose, so that the 12-bit overhead could be amortized across several uses. But in general, it seems unlikely that much space can be saved unless the "density" of facts in a predicate is well below 0.1.

Once the jump from single bits to explicit values has been made at one argument position, it becomes easier to justify converting more arguments from implicit to explicit representation. Continuing the above example, suppose that no one company has vacancies in more than 10 job categories. We can then use a VP-set of size (256, 16, 4) and two p-vars of size 6 bits to represent the values of the *job* and *state* arguments explicitly. Compared with the bit-string representation given above, this amounts to replacing a 50-bit p-var with 12 bits of p-var storage, or altogether 62 bits with 24 bits, in a VP-set of size 16K. Thus while a fact density of 0.08 gave no improvement in space use, one of 0.02 gives a factor of roughly 2.5 to 4 (depending on how the overhead cost of a VP-set is treated).

If the companies were larger and did business in more than four states each, we might still want to exploit the sparseness of the *WillGoTo* predicate, representing *state* explicitly. Unfortunately, this predicate is not, so to speak, uniformly sparse over all states, because of the few students who will go almost anywhere. The sparseness can still be exploited, at some price in complexity, by dividing the students into the discriminating and the desperate, and using one VP-set to represent the *WillGoTo* predicate as applied to the discriminating students, with *state* explicit, and another to represent it for the other students. This second VP-set will be larger along its *state* dimension but smaller along its *student* dimension.

Semi-explicit representations give up, to some extent, the advantages claimed above for implicit representations. A particular fact might be stored at any of several addresses, and it may be necessary to examine them one by one. We shall soon see that the more explicit the representation, the more serial computation is needed to do inference. Fully explicit representations, using a p-var for every argument position, are not out of the question, but we expect them to be advantageous only rarely. They are directly analogous to the representations generally used on serial computers.

7.2.5 Translating between representations

Since some representations are economical for storage while others are fast for inference, and since space constraints may change depending on what inferences are being done, we can expect that we will sometimes want to translate facts in one representation to another.

It is easy to convert single bits into bit-strings and vice versa, using **scan!!** and **spread!!**; the details are left to the reader. Converting the bits into numbers for a semi-explicit representation is a little harder, because the address of the processor that receives a fact must be computed.

Converting bit-strings to numbers directly is an appealing idea, because it avoids any need for a VP-set large enough to hold all the single bits. However, some serialization, and therefore slowness, seems unavoidable, because the bit-string representation stores several facts on one virtual processor, and it cannot simultaneously compute the different addresses to which they must be sent. The reverse conversion, from numbers to bit-strings, is easy: left-shift a 1 by the value of the number, and **or** the results together with a `scan!!`.

If we choose to store actual argument values and omit any mapping to the integers, then conversion to any representaation that assumes such a mapping is, of course, impossible.

7.3 Running a Rule in Parallel

These representations for facts lead to several slightly different ways of doing inference. These are based on exploiting, as far as possible, the correspondence between the content of a fact and its location; where this is not possible, we can either "throw silicon at the problem" in the shape of large numbers of virtual processors, or do some things serially.

A consequence of our data-parallel orientation (which, in turn, is a consequence of the philosophy behind the CM's design) is that the front end decides to run a particular rule, or possibly a specified set of rules. All inferences that can presently be done by this rule are done at once, and then the front end chooses another rule to run (possibly after observing the results of the first rule). This allows the user's program to retain control over what inferences happen when, in the style to which rule writers have become accustomed in conventional rule systems with their agendas, conflict sets, or other control mechanisms. It can be seen as an advantage of the CM over MIMD machines, where different threads of computation proceed on their separate processors without much knowledge of each other's progress, causing tricky issues of load-balancing and synchronization to arise.

7.3.1 The basic idea

Once again we start with the single-bit representation of facts. Take the rule

$$Major(student, subject) \,\&\, Relevant(subject, job) \Rightarrow Qualified(student, job)$$

Create a VP-set of size (1024, 32, 64) and allocate Boolean p-vars in it for *Major, Relevant*, and *Qualified*, which may as well be named `M`, `R`, `Q`. Copy the bits representing the true instances of *Major* into `M` in the $z = 0$ plane of this VP-set, then `spread!!` them along the z dimension. Likewise copy the bits for *Relevant* into `R` in the $x = 0$ plane and `spread!!` them along the x dimension. At this point, the values of `M` and `R` are both `t` at processor (x, y, z) iff student x majored in subject y which is relevant to job z. So we

```
(*set Q (and!! M R))
```

and reduce the resulting bits to the correct answers for *Qualified* by **or**ing them together in a `scan!!` along the y dimension. Finally, we copy this result into the p-var where *Qualified* is to be stored.

In general, we create a VP-set having a dimension for every variable that appears in the rule, sized according to the type of the variable. We then copy the bits for each premise of the rule into the subspace of this VP-set having zero co-ordinate values in dimensions corresponding to variables not mentioned in the premise. Each premise is spread out along as many of these dimensions as are necessary, and then combined with the result obtained from the premises that came before it (this result may also need to be spread). Note that the combining function need not be an `and!!`, but can be any Boolean operator. The final result is then reduced along the dimensions corresponding to variables that do not appear in the rule's conclusions, and copied to wherever it belongs.

The advantage of this scheme is that matching of facts, which is the most time-consuming part of serial rule systems, becomes trivial; it is obtained virtually "for free" as a consequence of the arrangement of the facts. It is not really free, because the facts do not necessarily come neatly arranged in the right way (different rules can have different "right ways"), and large numbers of messages have to be sent between processors to re-arrange them: this is one of the slower operations on the CM. But it is enormously faster on a CM than on a serial machine, due to the large amount of hardware (and firmware) devoted to sending messages. We therefore believe that this inference mechanism is well-matched to the fundamental strengths of the hardware architecture.

If bit-strings rather than single bits are used, the above scheme works with few changes, except that we replace a `scan!!` along a bit-string with something like a `plusp!!` on the unsigned integer.

7.3.2 Recycling space

A fully configured CM-2 has only 4 gigabits of memory, and there will certainly be rules that would, under the above scheme, require more bits than this if we simply multiplied the sizes of the domains of all the variables. But such rules rarely mention all their variables in the conclusions, and often have (or can be re-arranged so as to have) disjoint subsequences of their premises which mention different sets of variables. For example, in

$$\begin{array}{c}
Major(student, subject)\ \& \\
Relevant(subject, job)\ \& \\
Hiring(co, job, state)\ \& \\
WillGoTo(student, state) \Rightarrow \\
\hline
Interview(co, student, job)
\end{array}$$

subject is mentioned only in the first two premises and *state* is mentioned later. Thus the binding of *subject* becomes unimportant before *state* ever gets bound, and there is no need to use separate dimensions of a VP-set for *subject* and *state*. If there were a dimension for each variable, the resulting VP-set would have size $1024 \times 32 \times 64 \times 256 \times 64 = 2^{35}$. Instead, after the first two premises have been dealt with, the resulting bits can be reduced along the dimension used for *subject*, which can then be re-used for *state*, or could be if it were large enough. In practice, we would probably create the VP-set using a size of 64 for this dimension, large enough to accomodate both *subject* and *state*. The VP-set would then have 2^{30} virtual processors—still too many for a CM-2, but tolerable if we can gain a few more factors of 2 by other means (see below).

Another possibility is to re-write the rule as the two rules

$$\begin{array}{c} \mathit{Major}(\mathit{student}, \mathit{subject})\ \& \\ \mathit{Relevant}(\mathit{subject}, \mathit{job}) \Rightarrow \\ \hline \mathit{Qualified}(\mathit{student}, \mathit{job}) \end{array}$$

$$\begin{array}{c} \mathit{Qualified}(\mathit{student}, \mathit{job})\ \& \\ \mathit{Hiring}(\mathit{co}, \mathit{job}, \mathit{state})\ \& \\ \mathit{WillGoTo}(\mathit{student}, \mathit{state}) \Rightarrow \\ \hline \mathit{Interview}(\mathit{co}, \mathit{student}, \mathit{job}) \end{array}$$

so that the *subject* and *state* variables do not appear in the same rule. Splitting the rule up in this fashion is likely to lead to slower execution than using a single rule, but might be preferable if space were tight and *subject* had many possible values and required a large dimension, while *state* required a small one: then the 4-dimensional VP-set needed for the second rule using (*student, job, company, state*) would be smaller than the one for the original rule using (*student, job, company, subject/state*).

7.3.3 Data compression

If a rule demands more space than the CM can provide, even with careful use of bit-strings and sharing of dimensions, there are several ways to proceed. If it is truly the case that billions (not merely hundreds of millions) of facts must be considered, funding may be requested to purchase several additional CM-2s; or the VP-set theoretically needed can be "sliced up" along some dimension and the slices executed in sequence. For example, given a space 16 times too large for the machine, with an x dimension of 64, one could create a VP-set with an x dimension of 4, and run the rule for values 0–3 of x (but covering all values of the other variables), then with values 4–7, and so on for a total of 16 iterations. It will be quite simple to re-assemble these answers into a single predicate. (If the space required to represent an individual predicate exceeds that provided by the CM-2, additional funding is strongly indicated.)

But large numbers of potential facts do not necessarily mean that the number of actual facts is large: many predicates turn out to be sparse, as mentioned above in Section 7.2.4. A space apparently billions of bits in size may conceal a paltry few tens of millions of 1's, and we can represent them quite handily in a few hundred million bits, using a semi-explicit representation. Inferences on facts represented this way turn out to be only slightly harder than on the fully implicit representation.

The best case is one in which an explicitly represented variable appears only once in the premises of a rule: no matching is then necessary on the bindings of this variable, which can be carried straight through from premise to conclusion on the processors whose bits (representing the other premises of the rule) are all 1. In our rule

$$\begin{array}{c} \mathit{Major}(\mathit{student}, \mathit{subject})\ \& \\ \mathit{Relevant}(\mathit{subject}, \mathit{job})\ \& \\ \mathit{Hiring}(\mathit{co}, \mathit{job}, \mathit{state})\ \& \\ \mathit{WillGoTo}(\mathit{student}, \mathit{state}) \Rightarrow \\ \hline \mathit{Interview}(\mathit{co}, \mathit{student}, \mathit{job}) \end{array}$$

the *student* and *company* variables have this property. In general, since we are free to choose which variable(s) shall be explicitly represented, this best case may occur oftener than most "best case" scenarios. However, making this choice for purely local reasons (local to one rule, that is) may involve extra computational cost for translating between representations, when we use the facts inferred by one rule as inputs to another.

It also will not necessarily lead to the best use of space. Suppose we decide to represent *student* explicitly; consider the *Major* predicate, which is certainly sparse. Some departments have hundreds of students, while most have a few score, and some very few. This means that not only are the numeric values representing individual students be large (ten bits), but the size of the dimension along which they will be spread cannot be small, even if we cream off the highly popular departments and represent them separately. We cannot make effective use of the sparseness of the *Major* predicate in this way. So there will often be incentives to use explicit representation for variables on which matching is to be done.

Matching with explicit representation

This is harder than matching on implicitly represented variables, because the association between address and content is absent. For fast matching of two premises on a variable which is explicitly represented in one or both of them, it is best to use separate dimensions for the occurrences of that variable in each premise; one of these will be available for recycling immediately after the match. Suppose, for example, that *Major* is represented in a space of size (1024, 2) with *subject* explicit, and *Relevant* is represented using single bits in a space of size (32, 64). We then create a VP-set of size (1024, 2, 32, 64, ...) and copy and spread the facts into it in the way outlined above; a match is indicated by each processor whose z co-ordinate is equal to the numeric value it holds for a student's major subject. The matches are then reduced along the z dimension, which can be recycled. We may want to make this dimension larger than 32 so that we can re-use it to represent *state*.

The adaptations of this process to the cases when *Relevant* is stored using bit-string or explicit representation for *subject* are left to the reader.

If there is not enough room in the CM for the desired extra dimension, so that both predicates use the same dimension for *subject*, then there is no escape from serialization: we must iterate through the students' majors and find the jobs that each one is relevant to.

Matches on two or more explicitly represented variables (such as computing $a(x, y, z)$ & $b(y, z, w)$ with both y and z represented explicitly) impose further computational requirements—though as in the case of space economy, increases in explicitness are less expensive than the initial jump from fully implicit to partly explicit representation. Such a match requires either two dimensions for y and two for z, or a two-stage process which first matches on y only, and then recycles the extra dimension used for y and uses it to match on z. Thus, compared with matching on one explicit variable, we have a choice between using much more space and using twice as much time (probably closer to three or four times after various overheads are taken into account).

7.4 Running several rules in parallel

While some knowledge-based applications do require the processing of very large amounts of data in parallel, others process moderate or small amounts of data thr^ugh large numbers of rules. It may be that the rules are inherently serial, each of them producing facts which the next rule needs, but there is usually some degree of parallelism between rules; much previous work has been devoted to ways of exploiting this parallelism. The approach outlined above can sometimes be used without modification on a set of rules that differ only by the presence of different constants in their premises and conclusions, such as

$$PaintJob(car, Red) \;\&\; Seats(car, 2) \Rightarrow Type(car, SportsCar)$$

$$PaintJob(car, Brown) \;\&\; Seats(car, 6) \Rightarrow Type(car, Sedan)$$

$$\vdots$$

because such a set can be re-written as a single general rule plus a table of facts. But in general we expect to encounter many rule sets that are not of this kind, or not immediately recognizable as such or transformable to this form. In this section, we discuss ways running several rules at once on the CM, tacitly assuming that no one of them takes up more than half of the CM under a reasonably efficient representation.

7.4.1 Compatible rules

We can say that two rules are compatible with respect to their premises, if they have the same number of premises and the premises can be put into correspondence with each other such that they refer to predicates that are stored using the same representation in VP-sets of the same size. A similar definition can be made for compatibility with respect to conclusions. We then say that two rules are compatible if they are compatible with respect both to their premises and to their conclusions. So if the university has 600 staff, each of whom owns cars of various different models (out of 25 altogether), and there are 55 garages nearby, each of which can service some makes of cars, then a rule

$$Owns(employee, model) \;\&\; Services(garage, model) \Rightarrow Customer(employee, garage)$$

would be compatible with

$$Major(student, subject) \;\&\; Relevant(subject, job) \Rightarrow Qualified(student, job)$$

(the order of arguments in predicates is immaterial; we can map these to dimensions in any order we please, and the dimension sizes are still $(1024, 32, 64)$ for single-bit representation).

These two rules would require VP-sets of the same size to run in, and would be executed by the same sequence of *Lisp operations. They can therefore be run together in a VP-set of size (1024, 32, 64, 2), where the extra dimension is used simply to distinguish one rule's space from the other's. This construction is, of course, generalizable to numbers larger than

2. It remains to arrange for copying data in and out of this "layer cake" of a VP-set; recall that nothing was said about the two rules using the *same* predicates in their premises, so the information cannot simply be copied once and spread. If we have to do separate copy operations for each rule then almost nothing is gained over executing the rules serially, since the copying usually takes far longer than the inference.

This copying can be done in parallel if we change our representation for predicates slightly: after identifying sets of predicates that require the same sized VP-set to store their bits, we create one VP-set for each such set of predicates, having an extra dimension (i.e. a set of binary predicates would get a 3-dimensional VP-set) which is used to distinguish different predicates in the set. So *Major* and *Owns* can be stored together in a VP-set of size (1024, 32, 2), or perhaps (1024, 2, 2) if each employee owns at most two cars and a semi-explicit representation is used. It then becomes possible for one *Lisp operation to cause facts for different predicates to be copied into the spaces for the different rules, since they are coming from values of the same p-var at different addresses.

This idea extends easily to the other two representations for facts. Two additional data structures become necessary: a mapping of predicate names to co-ordinates in the extra dimension (this mapping can be kept on the front end); and some form of index saying which predicate is used in each premise or conclusion of each rule. This index must be kept on the CM, since the processors associated with each rule must be able to construct the addresses for fetching and writing facts.

7.4.2 Faking it

The ability to run rules in parallel will be much more useful if it applies to sets of rules that are not quite compatible in the sense described above. Minor sacrifices in space and time appear to bring this within reach.

If two rules have compatible premises but use different Boolean combinations of them, they can be run in parallel, though slightly more slowly than otherwise, by storing the appropriate Boolean operators with the rules and instructing each processor to use the operator(s) from its rule, rather than and!!.

If one rule has a premise which another rule does not have, and they are compatible with respect to the premises they both have, then we can just add to the second rule a dummy premise which is the same size as the real one in the other rule, and uses a predicate that is always true. The additional storage for such a predicate would not be a serious cost; alternatively, we need not store it at all, but could arrange for processors associated with the second rule to preform a Boolean operation that ignored one of its arguments (all Boolean operations are supported in *Lisp, including trivial ones).

If a set of predicates vary somewhat in size, and appear in otherwise similar rules, we may prefer to create a VP-set big enough for the largest of them. and store them all in it, accepting some waste of space.

7.5 Performance

We have been blithely ignoring questions such as how the best combination of representations for each predicate is chosen, how the mappings from objects to integers are constructed, how VP-sets and dimensions are allocated, recycled, and shared, and others of similar import. We have not even discussed whether rules should be fully compiled into *Lisp code or run with some kind of interpretation. This matters, because we cannot always predict whether a predicate computed by some rule will turn out to be sparse, and it may be desirable to check this at run-time and choose representations accordingly.

These questions, while important, basically come from a fairly well-understood and explored domain of combinatorial optimization, and thus are not of direct research interest relative to the possibility (or otherwise) of using the CM to speed up inference. The better solutions to them that we can find, the wider the set of problems for which the CM will be able to provide useful performance; we have provided the raw material, in terms of constraints and resource demands, on which such solutions will be based. We feel, however, that since the hardware and software of the Connection Machine is still evolving fairly rapidly, it would be premature to propose concrete algorithms for translating rules to *Lisp. The timings which we present below were therefore obtained with hand-written code.

7.6 Results

We tried two approaches to the interview arranging problem given above. Both of these involved supplying the data to the CM in single-bit form; one of them translated into bit-strings and the other (assuming some sparseness) translated into a semi-explicit representation with the job title being represented explicitly. In each case the times given include the cost of copying facts from the VP-sets where they are stored into the "working" VP-set, and of storing out the results into another VP-set; in practice, it would often be more efficient for the next rule to retrieve the results from the workspace.

The bit-string representation required 16k processors to run the rule on 1000 students and 32 companies with a VP-ratio low enough to prevent memory shortages; we squeezed it into 8k processors by cutting the number of students to 500. The processing time on the CM was about 0.8 seconds for each segment of 32 companies, or 5.6 seconds for the whole 200, of which roughly equal amounts were spent translating single bits into bit-strings, copying data from one VP-set to another, and doing inference. Since no assumptions of sparseness are implied by this representation, the potential number of interviews that could be arranged in this time is as high as $500 \times 200 \times 40 = 4,000,000$ yielding an average speed of some 714,000 LIPS or 87 LIPS per CM processor. On a fully configured CM-2 this would scale up to almost 6 million LIPS. The actual number of interviews arranged depended on the number of job categories in which each company had vacancies, and ranged up to about 250,000 (or 500 per student, a tiring schedule) yielding useful inference speeds of about 5 LIPS per processor, for a projected performance of a third of a million LIPS on a full CM-2.

The semi-explicit representation was constructed assuming that 900 of the students are willing to relocate to at most 6 states (and 40 states for the others), that each company operates in at most 4 states, and that each major subject is relevant to roughly 3 job categories. The sparseness of *Relevant* was only exploited in the size of the VP-set used to hold the answers. The sparseness of *WillGoTo* and *Hiring* on the *state* variable amounted to a factor of $^6/_{50} \times {}^4/_{50} = 0.0096$ and allowed us to run the rule on 1000 students and 32 companies in 8k processors. The entire problem was solvable on 8k processors in 15.6 seconds of CM processing. Assuming (unrealistically!) that the students' preferences and the locations the companies have chosen are randomly distributed, there will be about 16 companies active in each state, so the 900 students will each be able to interview with roughly 76 companies in up to 6 job categories, yielding a potential for 405,000 interviews, whie the other 100 will interview with almost 200 companies for 120,000 interviews. This total of 525,000 interviews represents about 4 per processor per second.

The difference between the maximum and observed performance figures reflects the difficulty of exploiting all the sparseness that the problem exhibits, and the inherent disadvantage of the semi-explicit representation for inference speed.

7.7 Extensions and Future Work

The functionality provided by the rule system implementation we have outlined is very basic, and lacks several features regarded as important in many modern rule systems. In this section, we survey some of these features and discuss the possibilities of adding them.

7.7.1 Functions

No attention has yet been paid to how the CM can execute rules with premises that perform computation rather than (or as well as) table lookup. Premises such as $x = y + z$ are useful and occur often. If the values concerned are of a type that the CM can store, and if the CM processors can do the computation, and if a semi-explicit representation is used, and if all the variables concerned are represented explicitly, and if actual values rather than integers are used (which prevents any translation to a fully implicit representation), then we can handle this kind of premise. Otherwise, we need something new.

The main obstacles to computing function values within rules are two: the computation may not be performable by the CM, and the result may need to be found (or inserted) in the mapping from a domain to the integers. Little can be done about the former, except perhaps await the arrival of more capable parallel machines and languages. But if the functions are purely extractors or recognizers, and do not create any truly new values, then it may be possible to replace them with tables or association lists, which can be stored on the CM and used in parallel.

The need to work with domain mappings is a more surmountable obstacle, especially in the case of values that can be stored on the CM. (There may exist a narrow class of functions whose arguments and results can readily be stored on the CM but whose computation is not easily done there; we shall ignore it.) Techniques are known for storing mappings on the CM, and even for extending them. Doing this in parallel (i.e. obtaining the mapped images of many values at once) requires either more ingenuity or more virtual processors than doing it once, but is possible.

7.7.2 Backward inference and constants

A large and venerable class of knowledge-based applications work not by forward inference from a mass of data, but by backwards inference from a tightly specified goal, or sometimes by forward inference from a specific fact; mixtures of these reasoning modes can also be profitably used [68]. Backward chaining is just as deserving of parallelism and speedup as forward chaining. We can offer little to the early stages of a backward chaining proof of a goal, since it will perform a relatively small number of highly constrained inferences, quite possibly using diverse rules; this is bettei done on a MIMD machine. We also have not *found a way to use the CM to exploit* AND*-parallelism between conjoined subgoals.* The most effective way to do backward chaining on the CM may well be to do forward chaining instead.

Another way in which inference may be restricted is by the presence of constant arguments to predicates in rule premises (or conclusions). It is easy for the CM to take advantage of this: the use of a constant presumably means that there are fewer variables in the rule, so a smaller VP-set will be needed, and the addresses from which facts are fetched will have that constant "hard-wired" into one of their co-ordinates (or rather, the integer corresponding to it). The decision about whether to run a particular rule in response to some event will also have to take the presence of constants into account, since they may make the rule inapplicable.

7.7.3 Truth maintenance

It is possible, though expensive in space, to add justification-based truth maintenance to our proposed implementation strategy for rules on the CM. We can retain a record of how each fact was inferred by the simple act of not throwing away the bits (or other values) that we obtained while processing the premises of a rule, and instead keeping copies of them on the processors where they arose. Then when some original fact is retracted, we can trace through the inferences that were made from it. For each inference, we must check whether the fact it concluded is supported by any other inference using the same or another rule. This latter requirement means that an index between facts and the rules that can conclude facts is needed; the information about rules that we proposed to store on the CM in Section 7.4 is probably enough.

7.8 Interfacing to serial machines

Our present design requires a serial front end machine not only to control the use of rules, but also to inject facts into the CM so that the rules can work on them, and to extract them and present them to the user or to other components of a knowledge-based system. Work has been done [19,35] on parallel implementations of truth maintenance and frames, but we have not determined the extent to which they are compatible with our design.

In any case, a KBS written in Lisp could not run entirely on the CM, since important pieces of Lisp functionality are not supported on it. The question then arises, "Which is to be master?" In other words, whether the state of the system's knowledge is to be maintained on the serial machine and transmitted to the CM as required, or vice versa.

Since transfers of large blocks of data between the front end and the CM are relatively slow, it seems unwise to perform such a transfer and then allow the state of knowledge on the CM to get out of date, so that another large transfer is required. This argues in favour of having the "authentic" version of knowledge reside on the CM. It is quite easy for the front end to interrogate the CM about the truth of any particular fact, or any defined set of facts drawn from a single predicate: just consult the mappings from object types to integers to determine which processor(s) the fact(s) are stored on, and retrieve a single fact with `pref!!` or a block of them with `pvar-to-array`. It is much harder for the CM to find and present to the front end all facts that mention a given object; this might well involve getting values stored in different VP-sets, and so the front end would be required to direct the retrieval. This suggests strongly that a rule system implemented on the CM will be far more useful if it is accompanied by a compatible implementation of a frame system.

7.9 Summary and Implications

We have defined a spectrum of representations for facts on the CM, with associated inference methods, that enables various compromises between space- and time-efficiency to be struck. The representations work best when the sets of values to be dealt with are known in advance, but can, with some loss of efficiency, handle changing sets: upper bounds on the sizes of the sets are presently essential (but it is rumored that future releases of CM software will have features permitting expansion of sets). The inference methods yield a high degree of useful parallelism, in that many facts are processed simultaneously in ways which could not easily be duplicated on a serial machine using more sophisticated representations. Transfer of facts between the front end and the CM remains a bottleneck. Effective use of the CM's resources appears to require pre-processing or optimization of the *Lisp code into which rules are translated; we are not sure that the time is ripe to develop such a pre-processor, since improvements to the *Lisp language, and the broader state of the art of parallel knowledge-based system technology, are likely. The development of techniques for implementing other components of KBS's in parallel, and the assembly of a compatible suite of such techniques, is much to be hoped for.

7.9.1 Implications for hardware

We have come to believe that virtual memory would be almost as useful for massively parallel hardware as it has proved to be for serial hardware. Since the cost of magnetic disk storage stubbornly refuses to decline as fast as that of silicon circuits, another technology may have to be found for this.

For our purposes, and (we suspect) for many other purposes as well, flexibility in the number of CM processors allocated to a particular task is highly desirable. One can envisage a hybrid architecture wherein a large-grain parallel machine, with perhaps 16 or 64 processors, is linked via some form of nexus or crossbar to a pool of small processors in a CM-like configuration, but grouped into sections of, say, 1,024 processors, versus the 8,192 that is the smallest section presently allowed. A problem would initially be split up into pieces to be distributed among the large processors, each of which could then attach a suitably-sized section of smaller processors for assistance with highly data-parallel parts of its subproblem.

7.9.2 Implications for software

It seems inevitable to us that parallel languages that free the programmer from doing explicit allocation of processors will become the norm. But we do not expect that this will make much difference to the algorithms and ideas in this chapter, except that explicit attention to sizes of VP-sets will no longer be needed.

The major change we expect to see is that rule systems will re-adapt themselves to parallelism. The concepts of "conflict set" and "agenda" will change their meanings radically once it becomes possible to distinguish computationally between what used to be accidental and essential sequencing of inference steps. A computational distinction will also arise between sequencing as resource allocation and sequencing as data flow. If rule 1 is more important than rule 2 but enough resources exist to run both of them at once, this can be done; if rule 2 requires the facts inferred by rule 1, then it must be run afterwards. We thus expect that pattern-driven non-deterministic languages in the style of Assertion Nets 2 will be highly suited to parallel implementation.

References

Bibliography

[1] Balzer, R., "Transformational Implementation: An Example," *IEEE. Transactions on Software Engineering vol. 7*, 1981, pp. 3-14.

[2] Balzer, R., "Draft Report on Requirements for a Common Prototyping System," *SigPlan Notices vol. 23*, 1988.

[3] Barstow, D., *Knowledge-Based Program Construction.* New York: North-Holland, 1979.

[4] Beckman, L., Haraldson, A., Oskarsson, O., and Sandewall, E., "A Partial Evaluator, and its Use as a Programming Tool," *Artificial Intelligence vol. 7*, 1976, pp. 319–357.

[5] Bobrow, D. G., "New programming languages for AI research," *SRI Technical Note 82* Menlo Park: SRI, 1973.

[6] Bobrow, D., and Stefik, M., The LOOPS manual. Technical Report: KB-VLSI-81-13. Xerox PARC, Palo Alto, California, 1981

[7] Bobrow, D., and Winograd, T., "An overview of KRL, a knowledge representation language," *Cognitive Science vol. 1*, 1977, pp. 3–46.

[8] Boose, J. H., "A knowledge acquisition program for expert systems based on personal construct psychology," *International Journal of Man-Machine Studies vol. 23*, 1985, pp. 495–525.

[9] Boose, J. H., "Knowledge acquisition for knowledge-based systems," *Seventh National Conference on Artificial Intelligence, Tutorial Program MP4 materials*, 1988, pp. 35.

[10] Cardelli, L., "Compiling a Functional Language," *ACM symposium on Lisp and Functional Programming*, 1984, pp. 208–217.

[11] Clancey, W. J., "Heuristic classification," *Artificial Intelligence, vol. 27*, 1985, pp. 289–350.

[12] Clocksin, W. F., and Mellish, C. S., *Programming in Prolog, 2nd edition*, Berlin: Springer-Verlag, 1984.

[13] Cohen, D., "Automatic Compilation of Logic Specification into Efficient Programs," *Proceedings of Sixth National Conference on Artificial Intelligence* Menlo Park: AAAI, 1986.

[14] Corkill, D. D., and V. R. Lesser, "The Use of Meta-level Control for Coordination in a Distributed Problem Solving Network," *Proc. 8th Int. J. Conf. Artif. Intell.*, Karlsruhe, Germany (August 1983), pp. 748–755.

[15] Dahl, O.J., and Nygaard, K., "SIMULA-an algol-based simulation language," *Communications of the ACM, vol. 9*, 1966, pp. 671–678.

[16] deKleer, J., Doyle, J., Steele, G. L., and Sussman, G. J., "Explicit control of reasoning," reprinted in *Readings in Knowledge Representation*, eds. R. J. Brachman, and H. J. Levesque, Belmont: Morgan Kaufman, 1985, pp. 346–355.

[17] DeMichiel, L. G., "Overview: The Common Lisp Object System," *Lisp and Symbolic Computation, vol 1*, 1989 pp 227–244.

[18] Dewar, R.B.K, Grand, A., Liu, S.-C., Schwartz, J.T. and Schonberg, E., Programming by refinement, as exemplified by the SETL language. *ACM Trans. Program. Lang. Syst. vol. 1*, 1979, pp. 27–49.

[19] Dixon, M., and de Kleer, J., "Massively parallel assumption-based truth maintenance," *Proceedings of the Seventh National Conference on Artificial Intelligence*, 1988, pp. 199-204.

[20] Dybving, R. K., *The Scheme Programming Language*, Englewood Cliffs: Prentice Hall, 1987.

[21] Erman, L. D., F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty," *Comput. Surv.*, vol. 12, no. 2 (June 1980), pp. 213–253.

[22] Fickas, S., "Automating the Tranformational Development of Software," *IEEE. Transactions on Software Engineering, vol. 11*, 1985, pp. 1268–1277.

[23] Fikes, R., and Kehler, T., "The role of frame-based representation in reasoning." *Communications of the ACM, vol. 28*, 1985, pp. 904–920.

[24] Fikes, R., R. Nado, R. Filman, P. McBride, P. Morris, A. Paulson, R. Treitel, and M. Yonke, "OPUS: A New Generation Knowledge Engineering Environment. Phase 1 Final Report," IntelliCorp, Mountain View, California, 1987.

[25] Filman, R. E., "Reasoning with Worlds and Truth Maintenance in a Knowledge-Based Programming Environment," *Communications of the ACM, Vol. 31*, 1988, pp. 382–401.

[26] Filman, R. E., and D. P. Friedman, *Coordinated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill, 1984.

[27] Freudenberger, S., Schwartz, J.T., and Sharir, M., "Experience with the SETL Optimizer," *ACM Trans. Program. Lang. Syst. vol. 5*, 1983, pp. 26–45.

[28] Garey, M. R., and Johnson, D. S., *Computers and Intractability: A guide to the Theory of NP-Completeness.* New York: Freeman, 1979.

[29] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment* Reading: Addison-Wesley, 1984.

[30] Goldberg, A., and Robson, D., *Smalltalk-80: The Language and Its Implementation* Reading: Addison-Wesley, 1983.

[31] Green, C., "An application of theorem proving to problem solving," *Proceedings of First International Joint Conference on Artificial Intelligence*, 1969, pp. 219–239.

[32] Green, C., "Results in Knowledge Based Program Synthesis," *Proceedings of Sixth International Joint Conference on Artificial Intelligence*, 1979, pp. 342–344.

[33] Green, C., and Westfold, S., "Knowledge-Based Programming Self-Applied," *Machine Intelligence 10*, New York: Wiley, 1982, pp. 339–359.

[34] Hayes, P. J., "Computation and Deduction," *Proceedings of 1973 Mathematical Foundations of Computer Science Symposium*, Czechoslovakia: Academy of Sciences, 1973.

[35] Evett, M., and Hendler, J., "Parallel knowledge representation on the connection machine," *Proceedings of Parallel Computing 1989*, 1989. (to appear).

[36] Hewitt, C., "Procedural embedding of knowledge in Planner," *Proceedings of the Second International Conference on Artificial Intelligence, 1971*, pp. 167–182.

[37] Hewitt, C. E., and H. Baker, "Laws for Communicating Parallel Processes," in B. Gilchrist (ed.), *Information Processing 77: Proceedings of the IFIP Congress 77*, North Holland, Amsterdam (1977), pp. 987-992.

[38] Kahn, K. M., "Partial Evaluation, Programming Methodology, and Artificial Intelligence," *AI Magazine, vol. 5*, 1984, pp. 53–57.

[39] Kahn, G., Nowlan, S., and McDermott, J., "MORE: An intelligent knowledge acquisition tool," *Proceedings of the Ninth International Conference on Artificial Intelligence*, 1985, pp. 581–584.

[40] Kant, E., *Efficiency Considerations in program synthesis: A knowledge-based approach*, PhD. Thesis, Computer Science Department. Stanford University, 1979.

[41] Kornfeld, W. A., and C. E. Hewitt, "The Scientific Community Metaphor," *IEEE Trans. Syst. Man Cybern.*, vol. SMC-11, no. 1 (January 1981), pp. 24–33.

[42] Kunz, J. C., "Model Based Reasoning in CIM," in M. Oliff (Ed.), *Expert Systems and the Leading Edge in Production Planning and Control*, pp. 111–130.

[43] Kunz, J.C., Stelzner, M.J., and Williams, M.D., "From classic expert systems to models: Introduction to a methodology for building model based systems," in *Topics in Expert System Design: Methodologies and Tools*, eds. Guida, G and Tasso, C. New York: North-Holland, 1989, pp. 87–110.

[44] Lasdon, Leon S., *Optimization Theory for Large Systems*, New York, MacMillian, 1970.

[45] Malone, T. W., R. E. Fikes, K. R. Grant, and M. T. Howard, "Market-like Task Scheduling in Distributed Computing Environments," Sloan School of Management, MIT, March 1986.

[46] Malone, T. W., R. E. Fikes, and M. T. Howard, "Enterprise: A market-like task scheduler for distributed computing environments," Working paper, Xerox Palo Alto Research Center, Palo Alto, October 1983.

[47] Marcus, S., "Taking backtracking with a grain of SALT," in *Knowledge Acquisition Tools for Expert Systems*, eds. Boose, J. and Gaines, B., London: Academic Press, 1988, pp. 211–226.

[48] Martin, J., *Application Development Without Programmers*, Englewood: Prentice Hall, 1982.

[49] McCune, B., *Building Program Models Incrementally from Informal Descriptions* PhD. Thesis, Computer Science Department, Stanford University, 1979.

[50] Milner, R., "A Proposal for Standard ML." *ACM symposium on Lisp and Functional Programming*, 1984, pp. 184–197.

[51] Morris, P.H., and R.A. Nado, "Representing actions with an assumption-based truth maintenance system," *Fifth National Conference on Artificial Intelligence*, 1986, pp. 13–17.

[52] Newell, A., "The knowledge level," *Artificial Intelligence vol. 18*, 1982, pp. 87–127.

[53] Nii, P., "The blackboard model of problem solving," *AI Magazine, vol. 7, no. 2*, Summer 1986, pp. 38–53.

[54] Nii, P., "The blackboard model of problem solving," *AI Magazine, vol. 7, no. 3*, Summer 1986, pp. 82–106.

[55] Novak, G.S., "GLISP: A Lisp-based Programming System with Data Abstraction," *A.I. Magazine vol. 4*, 1983.

[56] Phillips, J., *Self Described Programming Environments*. PhD. Thesis, Computer Science Department, Stanford University, 1984.

[57] Rosenschein, J. and Singh, V., "The Utility of Meta-level Effort," Knowledge Systems Laboratory Report KSL-83-20, Stanford University, March 1983.

[58] Rulifson, J. F., Derksen, J. A., and Waldinger, R. J., "QA4: A procedural calculus for intuitive reasoning," SRI Technical Note 73, Artificial Intelligence Center, SRI, 1972.

[59] Schwartz, J.T., "Automatic data structure choice in a language of very high level," *Communications of the ACM vol. 18*, 1975, pp. 722–728.

[60] Smith, D. E., "Controlling Inference," PhD thesis, Computer Science Department, Stanford University, 1985.

[61] Smith, R. G., and R. Davis, "Frameworks for Cooperation in Distributed Problem Solving," *IEEE Trans. Syst. Man Cybern. vol. SMC-11*, 1981, pp. 61–70.

[62] Steele, Jr, G. L. and Sussman, G.J., "The Revised Report on Scheme, a Dialect of Lisp," Artificial Intelligence Memo 452, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1978.

[63] Stefik, M., "An examination of a frame-structured representation system," *Proceedings of Sixth International Joint Conference on Artificial Intelligence*, 1979, pp. 845–852.

[64] Stefik, M., and D. G. Bobrow, "Object-Oriented Programming: Themes and Variations," *AI Magazine, vol. 6*, 1985, pp. 40–62.

[65] Sussman, G. J. and McDermott, D. V., "From Planner to Conniver - A genetic approach" *Fall Joint Computer Conference*, 1972.

[66] Teitelman, W., "INTERLISP Reference Manual," Xerox PARC, Palo Alto, California, 1978.

[67] Teitelman, W., and Masinter, L., "The Interlisp Programming Environment," *Computer, vol. 14*, 1981, pp. 25–34.

[68] Treitel R., and Genesereth M. R., "Choosing directions for rules," *Journal of Automated Reasoning, vol. 3*, 1987, pp. 395–431.

[69] Walters, J. R., and Neilson, N. R., *Crafting Knowledge-Based Systems.* New York: Wiley, 1988.

[70] Winograd, T., "Understanding Natural Language," *Cognitive Psychology vol. 3*, 1972.

DISTRIBUTION LIST

addresses	number of copies
RADC/COES ATTN: Michael L. McHale Griffiss AFB NY 13441-5700	5
Intellicorp Inc. 1975 El Camino Real West Mountain View, CA 94040	5
RADC/DOVL Technical Library Griffiss AFB NY 13441-5700	1
Administrator Defense Technical Info Center DTIC-FDAC Cameron Station Building 5 Alexandria VA 22304-6145	5
Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington VA 22209-2308	2
AFCSA/SAMI ATTN: Miss Griffin 10363 Pentagon Washington DC 20330-5425	1
HQ USAF/SCTT Washington DC 20330-5190	1
SAF/AQSC Pentagon Rm 4D 269 Wash DC 20330	1

HQ AFSC/XTHK 1
Andrews AFB DC 20334-5000

Air Force Info for Industry Office 1
Tri-Service Industry Info Ctr
5001 Eisenhower Ave
Alexandria Va 22333-0001

Air Force Info for Industry Office 1
1030 East Green Street
Pasadena Ca 91106

Air Force Info for Industry Office 1
AFWAL/GLIST (AFIFIO)
Wright-Patterson AFB OH 45433

HQ AFSC/XTH 1
Andrews AFB MD 20334-5000

HQ AFSC/XTHK 1
Andrews AFB MD 20334-5000

HQ SAC/SCPT 1
OFFUTT AFB NE 68113-5001

DTESA/RQE 1
ATTN: Mr. Larry G.McManus
Kirtland AFB NM 87117-5000

HQ TAC/DRIY 1
ATTN: Mr. Westerman
Langley AFB VA 23665-5575

HQ TAC/DOA 1
Langley AFB VA 23665-5001

HQ TAC/DRCA 1
Langley AFB VA 23665-5575

ASD/AFALC/AXAE 1
ATTN: W. H. Dungey
Wright-Patterson AFB OH 45433-6533

WRDC/AAAI-4 1
Wright-Patterson AFB OH 45433-6543

WRDC/AAAI-2 1
ATTN: Mr Franklin Hutson
WPAFB OH 45433-6543

AFIT/LDEE 1
Building 640, Area B
Wright-Patterson AFB OH 45433-6583

WRDC/MTEL 1
Wright-Patterson AFB OH 45433

AAMRL/HE 1
Wright-Patterson AFB OH 45433-6573

AFHRL/OTS 1
Williams AFB AZ 85240-6457

AUL/LSE 1
Maxwell AFB AL 36112-5564

HQ Air Force SPACECOM/XPYS 1
ATTN: Dr. William R. Matoush
Peterson AFB CO 80914-5001

HQ ATC/TTOI 1
ATTN: Lt Col Killian
Randolph AFB TX 78150-5001

Defense Communications Engr Center 1
Technical Library
1860 Wiehle Avenue
Reston VA 22090-5500

C3 Division Development Center 2
Marine Corps
Development & Education Command
Code DIOA
Quantico VA 22134-5080

US Army Strategic Def 1
ATTN: CSSD-IM-PA
PO Box 1500
Huntsville AL 35807-3801

Commanding Officer 1
Naval Avionics Center
Library D/765
Indianapolis IN 46219-2189

Commanding Officer 1
Naval Ocean Systems Center
Technical Library
Code 96423
San Diego CA 92152-5000

Commanding Officer 1
Naval Weapons Center
Technical Library
Code 3433
China Lake CA 93555-6001

Superintendent 1
Code 1424
Naval Postgraduate School
Monterey CA 93943-5000

Commanding Officer 2
Naval Research Laboratory
Code 2627
Washington DC 20375-5000

Space & Naval Warfare Systems Comm 1
Washington DC 20363-5100

CDR, U.S. Army Missile Command 2
Redstone Scientific Info Center
ATTN: AMSMI-RD-CS-R/ILL Documents
Redstone Arsenal AL 35898-5241

Advisory Group on Electron Devices 2
Technical Info Coordinator
ATTN: Mr. John Hammond
201 Varick Street - Suite 1140
New York NY 10014

Los Alamos Scientific Laboratory 1
Report Librarian
ATTN: Mr. Dan Baca
PO Box 1663, MS-P364
Los Alamos NM 87545

Rand Corporation 1
Technical Library
ATTN: Ms. Doris Helfer
PO Box 2138
Santa Monica CA 90406-2138

USAG 1
ASH-PCA-CRT
Ft. Huachuca AZ 85613-6000

1839 EIG/EIT 1
Keesler AFB MS 39534-6348

JTFPO-TD 1
Director of Advanced Technology
ATTN: Dr. Raymond F. Freeman
1500 Planning Research Drive
McLean VA 22102

Air Weather Service Tech Library 1
FL4414
Scott AFB IL 62225-5458

HQ ESC/CWPP 1
San Antonio TX 78243-5000

AFEWC/ESRI 3
San Antonio TX 78243-5000

485 EIG/EIR 1
ATTN: S Buzinski
Griffiss AFB NY 13441-6348

ESD/XTP 1
Hanscom AFB MA 01731-5000

ESD/ICP 1
Hanscom AFB MA 01731-5000

ESD/AVSE 1
ATTN: Capt Lesieur
Hanscom AFB MA 01731-5000

ESD/SZME 1
Hanscom AFB MA 01731-5000

Director NSA/CSS 1
T513/TDL
ATTN: D W Marjarum
Fort Meade MD 20755-6000

Director NSA/CSS 1
W157
9800 Savage Road
Fort Meade MD 21055-6000

Director 1
NSA/CSS
DEFSMAC
ATTN: Mr. James E. Hillman
Fort George G. Meade MD 20755-6000

Director 1
NSA/CSS
R5
Fort George G. Meade MD 20755-6000

Director 1
NSA/CSS
R8
Fort George G. Meade MD 20755-6000

Director 1
NSA/CSS
S21
Fort George G. Meade MD 20755-6000

Director 2
NSA/CSS
R523
Fort George G. Meade MD 20755-6000

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.