believe that a promising approach is to enrich the representational structure of our network language, so that the program knows not only that "X causes Y", but also has enough detailed knowledge so that it can explain why the connection is plausible. Such a program could aid the knowledge acquisition process by automatically critiquing the evolving network. Moreover, the program would ask questions to help it fill in the gaps and lack of coherency it detects.

Using the above example, after being told an implication (ordinary heuristic rule) relating brain-mass-lesion and brain-tumor, the program would attempt to classify these terms as processes or substances, note the locations, and isolate the particular causal interaction (mass causes a lesion). The key to such a capability is a representation language that defines concepts in terms of a relatively small number of relations (such as the conceptual dependency notation of Schank), plus generic knowledge of physical processes (e.g., the idea of a mass growing in size severing an enclosing substance). A great deal of research in qualitative reasoning of physical processes [3], particularly the research of Wendy Lehnert, lays the foundation for this kind of investigation.

The learning program we will construct could be termed "the advice requester." We believe that the ability to ask good questions is the mark of a good student or researcher, and it can greatly focus the learning process. Asking good questions requires relevant background knowledge, so the learner can learn something new by relating it to some facts or some general framework he already understands. This process can be complex, because there are levels and perspectives for understanding. What may at first appear consistent, could become puzzling later as new gaps appear in an evolving network. Concepts in fact change their meaning as exceptions and complex special cases come to light.

Learning by asking is a form of knowledge-intensive learning, to be contrasted with research in automatic learning (becoming more efficient). For knowledge engineering, such an approach is a dramatic switch from giving the program surface causal rules that it in no sense understands, to giving a program knowledge of underlying causal models that enable the program to justify its causal network. Most importantly, these models provide a set of expectations of states and faults that might be included in a causal network.

To take an example from another domain in which we are working, iron casting, one fault is a shrinkage cavity. Generic knowledge would indicate that a cavity is an absence of material, and that for casting the source of material is what is poured and a reservoir (part of the mold) to allow for shrinking. A built-in generic model would indicate three reasons why a source of material does not arrive at the sink: insufficient supply (reservoir is too small), supply lost by leaking, and blocked flow from source to sink. These three generic causes set up expectations for specific causal processes that will appear in the state network. A given knowledge base might refer to a model only once, but a library of such models would form the basis of a powerful knowledge acquisition program that could learn about new domains fairly quickly. We believe that this generic library of processes is part of what we call common sense knowledge.

An advice requester that would be as proficient as our best knowledge engineers is obviously not going to be constructed in a year or two. Our approach will be first to study the causal networks we have constructed in medicine and casting, and re-represent the knowledge in structures that include the generic, underlying abnormal processes. Next, using a method we have found to be advantageous in the past for refining a knowledge representation, we will construct a simple teaching program that can explain such a causal network and help the student critique an incomplete network. Ultimately, we believe that teaching students to think like knowledge engineers, that is *to learn the process of asking good questions*, may be even more valuable than directly trying to convey our products, the constructed knowledge bases.

4. Qualitative Simulation

GOALS

In the context of the Molgen-II project, we are exploring the process of scientific theory formation and modification by computer. Qualitative simulation of biological processes is an important part of this goal because it is necessary to ask about the results of hypothetical experiments in the course of theory formation and running a detailed simulation is often too expensive.

MOTIVATION

We are carrying out this research by studying a specific biological system: the regulatory genetics of the *E. Coli* tryptophan operon (the *trp system*). In the mid 1960's Dr. Charles Yanofsky (who is a collaborator with us on this project) began to probe the existing theory of gene regulation in this operon. Yanofsky's initial experiments revealed a number of anomalies. Since that time, Yanofsky's research (which continues today) resulted in the discovery of a totally new mechanism of prokaryotic gene regulation, and continues to refine our knowledge of exactly how this mechanism functions.

Our goal is to build a machine learning system which will accept an initial theory of gene regulation equivalent to that which Yanofsky began to probe in the 60's. We will then present our system with a series of experimental results based on Yanofsky's early observations. The learning system will then propose, implement, and attempt to confirm possible modifications to its theory of gene regulation.

We view *theories* - such as that of the trp operon's function - as problem solvers. The inputs to these problem solvers are descriptions of hypothetical experiments. The problem solver's outputs are descriptions of the predicted results of these experiments. Thus our learning program will be attempting to improve the predictive performance of a problem solver in bacterial regulatory genetics.

This research in machine learning presumes the existence of a simulator of the trp system. Building such a problem solver in itself raises interesting AI research issues in qualitative simulation. And building such a system in a form which can be *reasoned about* by another program (the learning element) complicates the problem even further.

Below we discuss our past work on the construction of two versions of such a problem solver ("the simulator"). We then outline a number of interesting research issues which this work has raised, and the approaches we plan to pursue in the construction of the simulator.

BACKGROUND

Version I

An exploratory version of the system was built in the Spring of 1984. The system was constructed using the UNITS system - one of the first general-purpose expert system building tools.

This first system was more of a success as a static knowledge base than as a dynamic simulator. Building this system forced us to come up with a concrete conceptualization of the problem domain: we determined the full range of objects the system would have to simulate, and considered what types of properties and internal states these objects have, and how they should be represented within the UNITS system. This knowledge base was examined several times by our biologist collaborators (Yanofsky and Dr. Robert Landick - a post-doctoral fellow in Yanofsky's lab) to help us detect errors and omissions.

The first system never contained much simulation capability. We did provide a mechanism whereby the state of the transcription mechanism could be determined after the user specified experimental conditions such as approximate tryptophan concentration and whether or not various objects such as the trp-R repressor and the trp promotor contained deleterious mutations or not. The simulation capability was essentially provided by backward chaining on the slot values of relevant units, with the actual inferences carried out by Lisp code attached to some slots.

We learned a number of things from this prototype system. The knowledge base we created became a concrete record of the objects relevant to problem solving in this domain, and of design decisions regarding their representations. We also discovered a number of things about the UNITS system:

1. Its knowledge base editor ran fairly slowly

2. We encountered and fixed several significant bugs

3. Its rule language is fairly awkward

4. Its inheritance hierarchy lacked some important features, such as the ability of a given object to inherit slots from more than one parent class.

(Note that points 1 and 2 result from UNITS having been developed and maintained within a university research environment.)

We also confirmed an observation made long ago by other AI researchers. Previous work has shown that the simpler a language is, the more amenable it is to being both executed by one entity and interpreted by another entity (such as an explanation facility). This is one reason expert systems are now often encoded in production rules rather than Lisp. It became quite obvious that if our learning element is forced to reason about a simulator containing Lisp procedures, it would be significantly more complex than if the simulator were written in another language. Simple as the syntax of Lisp is, even a reasonable subset of full Interlisp would contain quite a large number of fairly complex constructs, and would complicate the learning element tremendously.

We also made an interesting observation about how building an expert system can help experts think about their own domain. We will consider two examples of this particular idea. Both involve subclass units which were defined in the knowledge base by Karp and then discussed with Yanofsky and Landick. One subclass was called "DNA Segments" and was intended to include contiguous segments of DNA with discrete functions, such as: promoters, terminators, genes, and operators. Among the properties associated with this class were: sequence, position within some larger functional piece of DNA, and "generalized sequence" - an attempt to capture those sequence elements common to a given subclass of DNA Segments such as promoters. The other defined class of interest was termed "Molecular Switches". This was an attempt to represent the general notion of a molecule with two functional states, where transitions between states are caused by the binding and dissociation of the molecule from some other molecule. Examples of Molecular Switches are operators, promoters, and repressors.

In both cases Yanofsky and Landick expressed interest in these concepts, and noted that biologists had coined no terms for them. This suggests that these concepts are in some sense new to biologists. We hypothesize that the process of constructing an expert system will naturally lead to the identification of such general concepts - or, equivalently - to the creation of analogies between known concepts.

The reason for this is that in attempting to represent the behaviors of N different entities, it is often much more efficient (with respect to development time and code volume) to develop one general-purpose procedure which yields the N different behaviors given different parameter bindings, than it is to develop a different procedure for all N cases. It is the knowledge engineer's job to search for such general procedures.

Version II

Recently we have begun building the next version of the simulation system. We are implementing this using the KEE knowledge engineering tool developed by IntelliCorp. This will free us from all the limitations of the UNITS system mentioned above. We have accomplished the initial obvious goal of porting the knowledge base defined using UNITS to KEE.

Related Work

Recently a significant amount of work has been done in AI in Qualitative Simulation (de Kleer and Brown, Forbus, Patil, Kuipers). While this work is somewhat relevant to the research we propose, there are several reasons why it is not sufficient.

First, most of this work attempts to simulate systems described by *Physics* using *differential equations*. Much of this work is an attempt to generalize numerical differential equations into *qualitative differential equations*. However, Biology is a much younger science than Physics, and as such does not describe its mechanisms to nearly such a quantitative degree. Differential equations are rarely if ever used by Molecular Biologists, and hence qualitative differential equations do not

RESEARCH PLAN

The next step is to define the behavior for these objects so that actual simulations can be executed. This raises the question: in what language should this behavior be defined?

We rule out Lisp for reasons discussed earlier. We also believe production rules are not a good language for defining this behavior, for reasons that will be outlined below. We now discuss the features we believe the simulator should provide, describe research questions these features raise, and consider what constraints such a simulator imposes on an underlying implementation language.

Reasoning At Varying Levels Of Detail

We believe it is important that the simulator be able to reason at varying levels of detail depending upon the demands of a particular problem. That is, it should be possible for the simulator to solve many problems without simulating every single process it knows about in the most detailed manner possible. Rather, given a problem statement the simulator should perform meta-level reasoning to determine which processes to simulate, and at which of several possible abstraction levels to simulate each process. For example, in an experiment involving an otherwise normal *E. Coli* cell with a deleterious mutation in its trp-R protein, it should not be necessary to simulate the RNA-synthesis actions of RNA-polymerase at the nucleotide level. A more abstract representation of this process can be used (e.g., at the DNA Segment level).

It should be obvious that humans solve problems in this way as illustrated by the preceding example (that is, biologists can predict the outcome of this experiment correctly without employing such a detailed simulation). As human performance in this domain is reasonably high, there is reason to believe that this approach is not a bad idea. But what reason do we have to believe it is a good idea? Why not build a simple simulator that executes at one constant level of detail and be done with it?

This simulator is really only a sub-system of the whole discovery system, and as such could be called on many times during a given "discovery deliberation". It is thus quite possible that the speed of the simulator will affect the tractability of the discovery problem.

In addition, learning itself is usually subject to large combinatorial explosions. Consider learning to be a search through a space of concept descriptions, where generalization and specialization are among the state transformation operators. The more concept description primitives there are to combine, the less feasible this computation becomes. If the simulator represents object structure and function at one very detailed level, there will be a huge number of primitives to recombine. But if objects are represented at different levels of abstraction, learning too may proceed using "primitives" at higher levels, where presumably there are few primitives at the less detailed levels.

In Biology and the other Natural Sciences, many discoveries consist of the addition of detail to some model. Objects (e.g., ribosomes, atomic nuclei) which were once considered to be primitive black boxes have their insides probed to reveal a complex inner structure, or the range of their observed behaviors may increase. If our simulator is designed to represent and execute theories at different levels of detail, adding detail to an actual theory could be as natural as adding a new cell to the front of a linked list.

Another issue is user interaction. Users will want to include high level vocabulary terms in their specifications of experiments. And similarly, they will want to see these terms used in predictions. (Note this constraint does not force the system to be able to reason at *varying* levels of detail).

The issue of reasoning at different levels of detail is very relevant to current research in expert systems regarding "Deep vs Shallow reasoning". Some researchers argue that the "shallow reasoning" or reasoning from "empirical associations" used by traditional expert systems implemented in production rules (e.g., MYCIN) is qualitatively different from "deep reasoning" or reasoning from "first principles" which human experts are able to use when their "shallow reasoning" fails, or when "deep" explanations are required. I claim that while it is certainly important to be able to reason in a more detailed manner when a standard approach to solving a problem fails, and that it is crucial to be able to provide deeper justification for a line of reasoning than simply citing rules X and Y, that there is no absolute distinction between "deep" and "shallow" reasoning. What is possible is to distinguish one line of reasoning from a *deeper* line which justifies it. The construction of this simulator should help to prove this point.

Production rules have not been designed for the task of reasoning at varying levels of detail. It is important to design a language which explicitly provides this ability.

Knowledge Representation

The initial work done on the simulator has alerted us to unresolved issues in knowledge representation related to inheritance hierarchies. The inheritance hierarchies of both UNITS and KEE provide the ability to define properties of a given class unit which are inherited by subclasses or members of that class. But in fact this notion of class partitioning blurs together - and is used by knowledge engineers to represent - at least four different concepts. These are the concepts of *class*, *abstraction*, *prototype*, and *object decomposition*. Inheritance hierarchies also force one to make some choice about what is a *primitive* object in a given domain. Yet the notion of an individual is a difficult concept to define - philosophers have devoted entire books to it. AI could benefit from a systematic study of all five of these concepts, and this simulator provides a challenging context in which to study them.

Another idea to explore is object behavior structuring. A given object may potentially exhibit several different behaviors. For example, messenger-RNA binds to different molecules, is translated into protein, and is slowly degraded within the cell. Consider two different approaches to representing this behavior. In an object oriented approach, all behavior specifications for a given object are viewed as part of that object. Thus, at a given instant in time it is easy to determine exactly what behaviors a given object will demonstrate. Consider a process-oriented structuring of behavior. Using this approach, a given behavior is structured within some larger process of which it is a part. Thus, the binding of mRNA to a ribosome would be viewed as one element of the complex process of translation, which would be considered quite distinct from the process of mRNA degradation. This makes it difficult to reason about sets of asynchronous processes operating in parallel, but provides an easier way of reasoning about a long series of events which are causally connected.

It is not clear what the precise trade-offs between these two approaches are. It may sometimes be necessary to employ both, which would probably require translation between the two. This distinction has been explored by the Computer Systems community, but these ideas should be transferred to the AI community and would probably gain some clarity in the process. seem to be useful simulation tools.

Second, the other work in qualitative simulation simply has not addressed many of the issues we propose above, such as reasoning at varying levels of detail and making more sense out of inheritance hierarchies.

Summary

We propose the following:

- To design a process specification language which will form the heart of the simulator for the trp system. This language will be fairly similar to production rules, but will overcome the shortcomings of production rules as discussed above.
- To implement an interpreter for this language which will allow both forward simulation to predict the results of a specified experiment, and backward simulation, to suggest experiments which would explain an observed result.
- To implement an actual simulator for the trp system.
- To explore possible means by which the simulator should decide at what level of detail a simulation should be run to solve a given problem.
- To explore issues in knowledge representation concerning the concepts of an abstraction, a prototype, a class, a composite object, and an individual.

5. Additional Basic Research of the Knowledge Systems Laboratory

In addition to the core research described above, there is considerably more research in the KSL that draws on the SUMEX resource and that inter-relates to the whole SUMEX community. This is briefly summarized below in three main projects of the HPP, LOGIC, and HELIX groups of the KSL. (See Appendix A on page 285 for a description of the organization of the KSL.)

Research on Multiprocessor Architectures for Symbolic Computation

As the aspirations for applied AI work rise, expert systems are becoming more complex, and the symbolic computations involved more compute-intensive. Medical and biological applications share the widely felt need for more processing per dollar in the future.

VLSI technology, of course, offers the prospect of inexpensive high speed computing, but only if methods can be found to organize large collections of processors and memories in systems for concurrent (parallel) processing. The Heuristic Programming Project began work on this problem in the mid-1970's, with SUMEX computer support, in a project called HYDROID, whose major result was a system for a network of processors known as Contract Net [67]. HYDROID was reborn in 1983 as Advanced AI Architectures (AAIA), and has received funding support from DARPA and computing support partially from SUMEX.

In the AAIA project, the proposed architectures are studied in simulation (on Symbolics workstations). The underlying architecture is a distributed processor and distributed memory network, simulated with our CARE simulator. On top of CARE various experiments in the development of Concurrent LISP are being done. Above the LISP level are levels of knowledge access and problem-solving framework. At the knowledge level, methods are being studied for rapid retrieval of objects and rules in a multiprocessor net. At the problem solving level, we are studying the "parallelization" of the Blackboard framework. The Blackboard framework was chosen because we felt that, overall, it was the most powerful of the modern AI problem solving organizations and offered significant opportunities for the exploitation of parallel processing.

The top level is the level at which applications are programmed, and the opportunities for parallelism at this level are mostly domain- dependent. However we are studying in detail applications of the particular class known as signal-understanding (or signal-to-symbol transformations), hoping to discover a few generalizations applicable to the class.

If the levels are "factored" carefully and correctly, the speed-ups from parallel processing,each level to the next, will multiply (!), yielding overall a major system-wide speed-up from modest gains at each level (which is all that one can hope for at present). The goal of the AAIA project is to refine the level-factoring and the speed-ups at each level over the next 2-4 years to produce an overall gain from multiprocessor "parallelism" of at least one hundred times that of conventional serial machines (as measured by the simulator).

A Retrospective of the AGE Experiments

The scientific work of the KSL is largely experimental in nature. Ideas are embodied in software systems and are tested in significant applications. The AGE project was one of those lengthy experiments. From the beginning it was supported by SUMEX as core research. It had multiple goals: a) to provide a readily useable software package for developing expert systems employing the Blackboard framework b) to study the Blackboard framework itself with a view toward simplifying and generalizing its various mechanisms and c) to study the problem of how to build a "knowledge engineering workstation" environment (i.e. put KE expertise into the box).

AGE-1 exists, has been widely used, and is widely distributed. Many technical reports and papers exist. At the KSL, the scientific tradition is to bring together, summarize, and interpret the results of our multi-year thematic studies in a single scientific monograph that represents the best scientific sense we can make of the many experiments in the line of study. We did it with DENDRAL (Lindsay, Buchanan, et.al.), later with MYCIN (Buchanan and Shortliffe). We will soon begin the effort to do the necessary and appropriate AGE retrospective study. It will be done as a "background" effort to other activities and will take about three years (elapsed time).

Research on Logic-Based Systems and Systems with Self-Awareness

One of the key limitations on the technology of logic programming is that the usual logical rules of inference are too weak. While traditional logical implication is an essential part of expert reasoning, by itself it is inadequate to explain the cognitive performance of human experts or to serve as the sole basis for a practical logic programming technology. Over the next five years we propose to study and implement four specific advanced reasoning techniques, viz. uncertain reasoning for resolution, theory formation based on measures of probability and simplicity, efficiency-enhancing theory reformulation, and counterfactual implication.

The key idea underlying logic programming is that of programming by description. In traditional software engineering, one builds a program by specifying the operations to be performed in solving a problem, i.e., by saying HOW the problem is to be solved. The assumptions on which the program is based are usually left implicit. In logic programming, one constructs a program by describing its application area, i.e., by saying WHAT is true. One makes one's assumptions explicit and leaves implicit the choice of operations.

Uncertain Reasoning

The actual techniques used to implement uncertain reasoning facilities have increased in sophistication since the introduction of "certainty factors" in MYCIN; the approach which has received the most attention recently is the use of Dempster-Shafer theory [64]. Here, ranges of probabilities are considered instead of specific values; this has the advantage that it is possible to describe situations where one is uncertain as to the accuracy of one's information by representing it using a wide interval of possible probabilities.

Existing work at Stanford has laid a theoretical foundation for the incorporation of Dempster-Shafer theory in a forward- or backward-chaining inference system. The inclusion of probabilistic information in a resolution-based system is not yet well understood, however, and coming to grips with this problem is one of the specific goals of this project.

Theory Formation

Many problems in AI involve learning by hypothesizing, including diagnosis, planning, natural language understanding, generation of tests or experiments, and the modelling of a user, agent or environment. Programs use *bias* to select among possible inductive hypotheses or theories.

Previous AI research has formulated bias in a procedural and often *ad hoc* manner. We seek to represent the bias employed in traditional AI approaches to theory formation in a *declarative* manner, axiomatically and semantically, so as to incorporate it into the logic programming methodology. One promising approach we plan to investigate is to represent inductive theories as the result of non-monotonic reasoning, in particular circumscription [46]. We aim to apply the tools of non-monotonic reasoning to the question of when and how to weaken an overly-strong bias, once a contradiction has arisen.

We plan to investigate diagnosis, in particular diagnosis of faults in digital circuits, as an application of these theoretical ideas about theory formation. We seek to enable the use of declarative, prior knowledge beyond the design specification, e.g. the likelihood of various faults, the observables and costs of tests; as well as to provide a more principled and flexible basis for preferences among fault hypotheses, e.g. via non-monotonic reasoning and reasoning about bias, than in previous AI approaches [14, 21]

Theory Reformulation

Understanding the role of representation in problem solving has long been recognized as a central problem in AI research. The question of how to reformulate a problem description to make its solution *transparent* is at the heart of this problem. The canonical examples cited are from the world of puzzles -- the mutilated array problem and the missionaries and cannibals problem. The latter was extensively analyzed by Amarel, to identify shifts in problem representation that make the solution process more *efficient*.

We have decided to concentrate on the largely unaddressed area of problem reformulations under a given problem solving method. Within it, we seek to study the class of efficiency reformulations that can be applied to a problem specification. We will carry out this investigation in the domain of digital circuits. Given a first order logic description of a circuit at a given level of detail (which should be sufficient to solve the problem at hand), we will find a suitable reformulation of structure and behavior rules of a circuit to make a certain class of problem solving (e.g diagnosis, simulation) easier (have better space/time efficiency). This domain is chosen mainly because a preliminary analysis shows that it is amenable to the sorts of reformulations we wish to consider.

Counterfactual Implication

A type of inference that we have just recently begun to consider is that appearing in "commonsense" implication. Consider the statement, "If it hadn't been raining yesterday, we would have had a picnic." Assuming that it was in fact raining, any complete inference scheme (such as the resolution-based theorem prover in MRS) will conclude that this statement is valid. We plan to continue the formal investigation of counterfactuals already begun and will implement the results of the investigation in MRS. In light of the fact that MRS has already been used to develop diagnostic aids in the domain of digital hardware, this seems an ideal opportunity to test both the applicability and effectiveness of this use of counterfactuals. We also hope that the inclusion of a counterfactual inference mechanism in a general-purpose expert system building tool will help illuminate the precise extent of the usefulness of counterfactuals to AI generally.

SOAR: An Architecture for General Intelligence and Learning

SOAR is to be an architecture for a system capable of general intelligent behavior -- of assimilating and working on novel tasks, using diverse knowledge, learning by experience, and reflecting on its own behavior. Work to date with SOAR already provides evidence for significant advances towards attaining such an architecture. We plan to continue the development and investigation of SOAR -- to test and augment the principles on which it is built, to expand its functionality, and to have it perform a wide range of demanding tasks. Our ultimate objective is to fashion an architecture that is capable of supporting the full range of flexible activities required of intelligent behavior.

SOAR embodies a collection of mechanisms and organizational principles that express a set of distinctive hypotheses about the nature of the architecture for intelligence.

1. *Uniform task representation by problem spaces.* Every task of attaining a goal is formulated as finding a desired state in a *problem space* (a space with a set of operators that apply to a current state to yield a new state) [52]. Hence, all tasks take the form of heuristic search.

2. *Any aspect of a task as an object of goal-oriented attention.* This includes the system reflecting on its own problem-solving behavior. An exact formulation of this property requires some care, because the architecture itself is a fixed structure. The essential feature is that no domain-dependent procedures lie outside the goal system -- for implementing operators, selecting operators, analyzing situations, or anything else.

3. *Uniform representation of procedural knowledge by a production system.* SOAR is realized in a specialized production system. All satisfied productions are fired in parallel, without conflict resolution. Productions can only add data elements to working memory; the architecture is responsible for all modification and removal.

4. *Knowledge to control search is ultimately expressed in a system of preferences.* Search-control knowledge is brought to bear by the additive accumulation (via production firings) of data elements. The end-result is a set of elements called *preferences* (about the various alternatives for behaving in a problem space).

5. *All goals arise to cope with difficulties in problem solving.* Ultimately difficulties arise from a lack of knowledge about what to do next. In the immediate context of behaving, difficulties arise when problem solving cannot continue. These difficulties are detectable by the architecture, because the fixed preference decision procedure concludes successfully only when the knowledge is adequate. It fails otherwise and the architecture itself creates goals for overcoming the difficulties. This principle of operation, called *universal subgoaling*, is the most novel feature of the SOAR architecture, and many other features build upon it, e.g., automatic detection of goal attainment and learning by chunking.

6. *The basic problem-solving methods arise directly from knowledge of the task.* SOAR realizes the so-called weak methods, such as hill climbing, means-ends analysis, alpha-beta search, etc., by adding search-control productions that express, in isolation, knowledge about the task (i.e., about the problem space and the desired states). The structure of SOAR is such that there is no need for the organization of this knowledge in a separate procedural representation. This is another novel feature of SOAR.

7. *Continuous learning by experience through chunking.* SOAR learns continuously by, in effect, automatically caching all of its goal results as productions. (This mechanism appears to be directly related to the phenomenon called *chunking* in human cognition, whence its name.) It learns both operators and search control, and it produces significant transfer of learning to new situations both within the same task and between similar tasks. This ability to combine learning and problem solving has produced the most striking experimental results so far in SOAR.

Our research will have a breadth-first flavor as we seek to add major intellectual abilities to SOAR, to make SOAR robust, and to develop a theoretical foundation for the SOAR design. Only the additions to SOAR are listed below for brevity.

Chunking as a general learning mechanism

We are currently investigating two areas where chunking may be found wanting: recovering from overgeneral learning and learning from examples. The first area involves being able to learn new chunks that override previously learned chunks that were overgeneral (that is, chunks that applied inappropriately). Since SOAR only learns from experience, we are investigating ways for SOAR to retry an errorful problem-solving episode more carefully. During the retry, it may be able to override an incorrect chunk and learn new chunks that will correct that chunk in the future.

The second area involves extending chunking. While chunking is based on learning during problem solving, the inductions necessary to learn from a set of examples appear at first glance to require a quite different learning mechanism. This research effort attempts to unify learning from examples with learning while problem solving. This extension is only one of several that could be probed to test whether chunking really is a general mechanism. (Actually, the right way to pose this issue appears to be what other aspects of problem solving must be coupled with chunking to accomplish each type of learning -- where chunking operates as the final memory-modification mechanism.)

Planning

Abstraction planning appears to be a natural uniform activity in problem solving [53] and it appears to translate into a natural uniform activity in SOAR. We will concentrate our initial efforts on this type of planning, because it seems more likely to prove useful with all tasks. Initially, for tactical reasons, we will work with tasks that are already operational in SOAR, such as the *R1* configuration task. Abstraction planning, especially with the constraint of universal applicability, should provide a major challenge to SOAR, since it poses quite novel design considerations, not present initially or in the extension to chunking. If SOAR adapts gracefully to planning, we will have another major item of evidence for SOAR. Contrariwise, if major difficulties arise, we should be able to discover some important limitations to the principles on which SOAR is built.

Problem-space creation

The creation of appropriate problem spaces is a critical aspect of SOAR's performance. For SOAR, creating new and better problem spaces takes the place of creating new and better representations. So far, SOAR does not do this. The problem spaces that are used are all instances of a few general problem spaces (for resolving ties among a set of objects or for evaluating an object or operator by looking ahead in the original space) or of user-created spaces (as in the gross means-ends structure of *R1-Soar*, *Dypar-Soar*, etc.). Indeed, it came as a surprise that we were able to avoid problem-space creation as a major roadblock early in the development of *Soar1* and *Soar2*. But any substantial degree of generality for SOAR requires a powerful capability for creating problem spaces.

2.2.1.3. Resource Hardware and Core System Development

Introduction and Background

We have already explained the systematic evolution of SUMEX-AIM from its original conception as the central node for a national community of biomedical AI scientists to a more and more distributed community and computing environment. We now want to sketch our plans for the hardware and system development of the resource for the proposed new grant period.

In summary, our development efforts will build on our past experience with Lisp workstations, attempting to make a more effective and intelligent computing environment for AI research and the dissemination of AI systems out to biomedical user environments. Just as our core research and AI applications efforts are aiming for systems that will have their impact 3-5 years from now, our computing systems work aims at the hardware foundations and system facilities of the same period. Certainly the current trend toward cheaper and more powerful workstations will continue. So as these machines become more ubiquitous, we must develop the system software that will give users the tools to take advantage of these machines in all their power and flexibility. This includes the full range of tools such as text processing, electronic mail, file manipulation, budget preparation and control, drawing and so on that keep workstation users tethered to expensive and overloaded mainframe systems. But it also includes extensions so that users can interact more effectively with their computing environment through more intelligent customized interface agents and can take advantage of the networked concurrent architecture these workstations represent. We plan no changes to our mainframe hardware facilities, but will continue to operate them for the on-going work of our community as possible with decreased DRR support.

As we will be discussing more fully, the growing collection of hosts and workstations has forced AI, distributed system, and networking researchers to reexamine the question of how to use many processors on a high bandwidth local area network (LAN) most effectively. Viewed as one large interconnected system, the amount of AI research that can be done is many times more that what was possible just five years ago, but we are encountering limitations because the traditional organization of such distributed processing power in fact wastes much of this power. At present the bottleneck in the development of network-based systems has become the software, with much of the potential of the powerful workstation hardware being unrealized. The first key is to find the appropriate role for the workstations within the context of the whole network-based system [58].

Workstations and Networking

From the outset, as our research computing began moving off of mainframe computers and onto a variety of personal Lisp machines, it was clear that these systems were an integral part of a larger network environment for the development, maintenance, and distribution of software and for access to services that are only cost effective as community resources. Systems software is continually being developed by both our own staff and the Lisp machine vendors. A network system facilitates the sharing and distribution of these software efforts and servers such as large disk files, file backup systems, high quality printing, remote network gateways, and shared mainframe hosts are best shared through network interconnections.

It is not possible or desirable to run all applications on the workstation [58]. For example, large database applications require huge amounts of disk storage and some graphics or signal processing applications are processor intensive and need special hardware. Printer services require knowledge of a diverse set of fonts and special text processing languages like Impress or Postscript, and processing mail needs address resolution and domain name servers. Still further restrictions are that particular workstations are tuned to run a particular flavor of Lisp and its extensive system support environment. Consequently, workstations have been tailored for a particular processing need, and to then look for the auxiliary software and hardware requirements elsewhere. Since our research staff and users do not all reside in the same building and since Lisp machine hardware and network servers are organized around computer rooms with cable length restrictions, we cannot currently give people the needed flexibility in geographic access to use a Lisp machine from anywhere on campus or from home either.

So, when a distributed system is viewed as a collection of heterogeneous hosts comprising one interconnected system, the system as a whole has a maximum work load potential which is a function of the resources of each of the hosts in the system, and the ability of these hosts to communicate with one another via the LAN. Currently, access to such systems and effective use of their resources fall far short of the potential for at least the following reasons:

- *Lisp Machine Cost*: While costs continue to fall, the highest performance Lisp machines are still rather expensive, ranging from around $30,000 to $120,000 and this is out of the reach of many researchers. Entry into the system is through a personal workstation and we are not able to afford giving each researcher dedicated access to the best systems. In effect without flexible access facilities, the limited number of personal computers provides for rigid control on the number of users. Unlike time-sharing systems where response degrades with each added user but where there is no rigid limit to the number of users, in a distributed environment without access to a personal Lisp machine, you cannot use *any* computing resources [58]. There is currently no adequate means of sharing these workstations and consequently keeping the cost per user at a minimum, and the usage per machine at a maximum.

- *Operating System Differences*: In order to use a remote host to run a program a TELNET connection must be established with that host. The user then logs in and runs the desired programs. This implies that a user must understand the details of the executive commands and file systems of several operating systems if he wishes to take advantage of all hosts on a network to aid his research.

- *Network Protocols*: Communication between hosts on the network is by the network protocols that each vendor supplies. In our unavoidably heterogeneous computing environment, most mainframes do supply servers for some protocols but not all mainframes supply servers for all protocols. Also, some protocols may run very efficiently on a server and others may not. This is certainly the case with respect to IP/TCP versus PUP/BSP under UNIX. IP/TCP is part of the UNIX kernel and PUP/BSP runs in user space making the latter much less efficient. This inefficiency is particularly noticeable as the number of connections increases on our file servers.

- *Resource Constraints*: A user cannot easily get a picture of what the load distribution is on the combined system resources. One server or mainframe may be idle and others busy. In fact, users simply view this system as they did a time-shared mainframe. In each circumstance the researcher has important work to do, and correctly sees the underlying system as a resource to get that work done in a timely way, and often under the pressure of a deadline. Thus, they push a particular environment for all that it is worth and the limitations of these environments are exposed and often pushed to unworkable extremes. Underlying a mainframe system is an operating system and scheduler that can manage and allocate its resources as a function of the number of users. In our current system access and allocation is at best ad hoc, and for the most part managed by each user. If our timesharing experience yields any axioms, then one of would be: In any computing environment users will attempt to reach or exceed the maximum work load potential of that system. Consequently, the resources of the system must be well managed by an agent that can visualize and appropriately and effectively allocate them.

- *Remote Connection Costs*: The primary means of accessing a remote host is to establish a TELNET connection and then run jobs as if you had a direct terminal line connection to that host. Maintaining a smooth typing response over a network is very expensive and the actual processing return for the work done on both the workstation and the remote host itself per keystroke is quite small. The cost of processing one character per packet is not that much more than the cost of 512 characters per packet. The overhead is with respect to the frequency with which the packets themselves must be processed in order to give the appearance of smooth typing. Efficient management of resources should be done in such a way that typing, mouse or voice interaction, view management and screen refresh are processed on the local workstation, and that communication with the remote host is task-oriented at a high conceptual level, and, consequently, minimal.

- *Network Transparency*: The network itself is not a transparent medium of communication in the system. If a user wishes to run a job that cannot be run on his workstation, he must log onto a particular mainframe that is also connected to the network, and run his job. If he wishes to retrieve a file he must know the file server on which that file resides. The user must always be aware of the various components of the system itself. When one uses a mainframe, he need not know how many disk drives, lineprinters, CPUs, buses, or i/o channels are involved in his getting a task accomplished. It would be considered absurd for the user to have to know on which disk drive his files are stored. The mainframe hardware is transparent to the user. This should analogously apply to a networked system but in most instances does not.

- *Concurrent Process Execution*: Some tasks may take several hours or longer to complete even on the most powerful Lisp machine. There is currently no generally accessible and satisfactory way of running such a task and sharing its processing among several idle Lisp machines, even if the task is one that can be separated into distinct and independent steps. As we undertake more and more complex AI applications and as we divide tasks logically between machines (such as is proposed for the *Interviewer* and *Reasoner* parts in the dissemination of the ONCOCIN system), parallel processing and use of workstations resources becomes an essential part of the future computing environment. Projects such as the HPP Concurrent Symbolic Computing Architectures project are working on parallel system designs with orders of magnitude improvement in performance. The results of this work are a long way off, however, and in order to reach those goals, researchers require a method of more effectively utilizing concurrency in available distributed machines..

So our plan is to work on reducing these limitations, concentrating on enhancing the computing environment of Lisp workstations and more effectively exploiting their combined resources.

Central Resource Operation

Our central mainframe computers have been powerful and superb resources for the SUMEX-AIM community over the past 12 years. However, the trend toward distributed workstations is clear and it would be inconsistent for us to seek full DRR support for these central machines for another 5 years. Still, we recognize that there is a community of users, particularly young projects which need seed support prior to obtaining major funding, who will depend on the central shared mainframe for several years. Therefore, we plan a conservative and responsible phase-out of these machines. We will discontinue DRR support for the DEC 2020 demonstration machine and the shared VAX 11/780 time-sharing system starting in year 14. We will phase-out the central 2060 more slowly, budgeting 80% support for its operations in year 14 and decreasing this in 20% steps until there is no remaining DRR subsidy by year 18. This should allow ample time for remote users to find and fund alternative computing resources, most likely workstations local to their research environments.

Hardware Purchases

Our hardware purchase plans for the next grant term are modest and are aimed at maintaining access to state-of-the-art workstations for our core work. For example, Xerox has just announced a model of the 1100 series machine that is expected to sell for $18,000-19,000, run InterLisp at comparable speeds to the 1108, and have a second integrated machine able to run IBM PC software. Other machines are being designed by Texas Instruments, Hewlett-Packard, Symbolics, Japanese manufacturers, and others that will strongly influence the system goals we have for the next 5 years. Thus, we budget $75,000 per year for new workstation hardware. In the first year we will buy 4 of the new Xerox systems for use in our development efforts and as part of the ONCOCIN dissemination research. We will select future year purchases from the then available systems.

The Lisp Workstation Distributed System/Kernel

Much work has already been done on distributed computing systems that we want to take advantage of, including work in our own Stanford Distributive Systems Group [39, 37, 9]. By supporting a *distributed operating system* the workstation may perform any function best suited to the user, the hardware, and the applications at hand [58, 38, 40, 60]. An implementation of this model consists of cooperating *kernels* providing an interprocess communication system, and services implemented as processes. Related work for distributed concurrent systems has also been done using the Actor/Apiary model [32], and the Contract-Network model [67]. In the Actor/Apiary model computation is performed by independent computing elements called actors which communicate with each other by message passing. The Apiary is a networked architecture for cooperating processors. The Contract-Network model provides negotiations for not only what is to be done but also who is best suited to do it.

In our initial approach, a Lisp Workstation distributed System (LW System) will be based on the *V System* [37] but will differ in the following respects. The V system incorporates both the V kernel interprocess communication as well as a V operating system which provide a total distributed operating system for those hosts on which it runs. But each Lisp machine for which we are targeting this design already has a highly-developed operating system. Functions such as process control and memory and device management already exist on these workstations, as do the tools necessary for managing the mouse, windows, and menus. The V Kernel interprocess communication primitives, using a fixed-length synchronous message protocol, do not. In this context, processes can reside on *any* host on the LAN, and communication between any of these processes is possible. The marriage of interprocess communication with existent operating systems in this fashion provides the basis for a distributed operating system. The resulting kernel is what we will call the LW Kernel, and the resulting system the LW system.

This wedding of the V Kernel message protocol and semantics with existing and powerful Lisp machine operating systems should yield a LW system with the strengths of both systems. The LW system will be able to take advantage of the extensive work in remote process execution and virtual graphics already incorporated in hosts running the V system. For example: The V system runs on non-Lisp diskless MC-68000 based workstations that can now be purchased for $8000. We have already written applications that run in InterLisp-10 on the DEC 2060 that allow us to remotely drive the virtual graphics terminal service (VGTS) software in these diskless workstations. On a moderately loaded DEC 2060 the remote creation of views, windows, the placing of graphical objects such as text, splines, lines, and rectangles in these windows, and the interaction of menus sent from the DEC 2060 with user "mouse-buttoning" on the workstation is very responsive. By porting the remote graphics software written for the DEC 2060 to any Lisp machine and then TELNETing into that Lisp machine from a workstation either at home or on the LAN immediately allows remote access to that Lisp machine from those locations. It should be noted that *all* remote graphics is done with the interprocess message protocol, and that the amount of information necessary for all but the graphics commands involving bitmaps is minimal and therefore achievable over relatively low speed lines.

In this model, the network consists of a collection of resources accessible by *clients* and managed by *servers*. A client can be either a program or human user [37]. In this context client and server are just "roles" played by processes. For example: A user or application might make a request of a file server. Here the user/application is the client and the file server is the server. The file server then may make a request of a disk server in which case the file server becomes a client and the disk server the server.

An LW exec will run as a process on a Lisp machine, and have its own executive window for command processing. This exec will have access to the entire LW System, and thus the LW Kernel which also runs as a process. Given the above model we might have the following example: Suppose the user wishes to run SCRIBE on some server in the distributed system. The user types "SCRIBE myfile" in the LW Exec window. The LW Exec creates a client process on the local host, and this client then queries the system for the best server for running SCRIBE and blocks waiting for a reply. When a server replies the local client then opens SCRIBE as a file to execute on the remote host. If this open is successful, the server has then created the SCRIBE process which then becomes the client while the Lisp machine client becomes a server. The SCRIBE client then requests input from this server, and receives the stream "myfile" which the client opens. The client runs SCRIBE and sends the results to the server which displays them in the local window. When SCRIBE has completed it closes the transaction and goes away. The local client/server ceases to exist, and the window is left for the user to peruse, and take further action on if desired (like printing the document).

Beneath the above scenario several other transactions took place. To initiate the first client/server relationship knowledge of the server willing to run SCRIBE was necessary. To accomplish this initial rendezvous the Lisp machine client needed to first determine where to run SCRIBE, and then log onto the remote system via that server. Determining where to run a process can be done within either a *static* or *dynamic* partitioning of the underlying distributed system.

In the static partition each host has a defined set of processes it is best suited to run at initialization time, and then this is invariant over the lifetime of that configuration. Dynamic partitioning is done when load sharing over the distributed system is desired and this can often require process migration to maintain system load equilibrium. Load sharing in this sense can only be used when the systems are relatively homogenous [58]. That is to say, one cannot migrate an executable Dandelion process to a 3600 because of the inherent hardware differences, although these two systems can have a client/server relationship because the process to process communication is machine independent.

So, in our example a static partitioning means that not all systems can run SCRIBE, and only those willing servers will answer. In this simple partitioning two servers are in the same equivalence class if they provide the same services. Here we say the distributed system is partitioned with respect to *willingness.* In the dynamic partition there is one equivalence class since all hosts are essentially identical. There are other partitionings worth examining.

Consider the relationship where two servers are equivalent if they can execute the same processes. Each of the equivalence classes in this partition is then dynamically partitioned with respect to load sharing with process migration. Here for example we might have four equivalence classes: SUN 68000 workstations, Xerox D-machines, VAX's, and 3600's. Note also that the system is always partitioned with respect to willingness.

There is also a slight variation on partitioning with load sharing. In this case we first statically partition the system with respect to willingness. Then we add the following constraint: A process will be run on the *least loaded* host willing to execute that process. This simple variation makes the system responsive to overall load without process migration. Thus, in our example we would have received three replies from servers willing to open SCRIBE for execution, as well as their load averages. One can then select the system with the least load to be the server or perhaps use more intelligent planning for complex multi-step tasks, anticipating future demands. The V system currently achieves load sharing without migration by running processes on the least loaded host. In our implementation we will begin by partitioning the distributed system with respect to willingness, and then experiment with the least loaded host constraint on this partition. Ultimately we are aiming for load sharing with process migration within classes of equivalent hardware configurations. Note that concurrency can be achieved in the simplest of these schemes.

Access to the file "myfile" was also necessary. This involves locating the file, it can reside anywhere in the system, and then acquiring read access privileges. Instead of sending "myfile" the *filepath* of "myfile" would have been determined on the Lisp machine, and the SCRIBE client would have then retrieved that file from its known source. This latter server could be a file server anywhere in the LAN.

The LW Kernel has then acted as an intelligent interface between clients and servers. Beneath the kernel the roles of processes may change and this is totally transparent to the kernel itself. A kernel or server of such a distributed system acts analogously to a hardware bus, being essentially a communications switch. In addition to the physical wires used to connect modules in a hardware bus, a standard bus arbitration protocol is agreed upon to define the semantics of the communication. Analogously, in our software model, in addition to the ability to send or receive a message, a protocol is defined for the semantics of the messages [58].

Machine Independent Interprocess Message Protocol

The machine independent interprocess message protocol is used to send, receive or forward messages between processes on either the same workstation or any workstation on the LAN which implements this protocol. These messages are synchronous and in implementations like V are fixed-length to minimize overhead in both the message sender/receiver interface as well as the parser. One can for example then allocate fixed length message buffers in the kernel for message queuing. The communication between processes is intended to look like procedure calls to the sender in the sense that at the highest level a sender calls a procedure with its specified parameters, and then as a process blocks awaiting a return value in the reply message. Note that this is unlike the actor model where messaging is asynchronous. In our model a degree of synchrony can be tolerated because the frequency of messaging is very low when compared to process execution time, and if one desires concurrency a server process can be spawned and then block awaiting a reply.

In order to send a message to a process, a "token" which includes both a host identifier and process number at that host is required. At each workstation the LW Kernel supports a process registration scheme that associates a *logical process identifier* with the registrant's process identifier [37]. Processes can then query the kernel for the process identifier corresponding to a known logical process identifier. This query is supported throughout the distributed system by the means of a process-query broadcast packet. Thus, having possession of such a token is sufficient to allow the passing of a message to the associated process. On a local host the kernel's token is globally defined to enable dispatching messages to the kernel itself.

In order to implement what are essentially *call by reference* parameters, a process can pass access permission to a memory segment to the recipient of a message. This access includes read, write and execute modes as well as the address of the segment. This is primarily used for file activity and buffers associated with those files but can also be used for creating processing "locks" on critical regions and marking data areas as read or write secure in conjunction with password or special process identifier privileges.

When a message is sent by a process, ultimately that message is formulated as a token, called procedure number, and called procedure parameters in a predefined network byte order which is transparent to both the sender and recipient of the message, and then dispatched by the resident kernel. The receiving kernel will then validate the token, and queue the message in a kernel message data buffer for the receiving process. The receiving process is scheduled by the kernel and when it is called uses a kernel procedure to formulate the data in the buffer as a procedure call and simply calls that procedure if it exists. Messaging between processes can be accomplished without addressing extensive programming language issues by using fixed length interprocess messages where each field in a message also a fixed length for which 32 bits is the chosen standard. This is sufficient for both integer and pointer constants since one can implement double precision if necessary. Under some circumstances a segment of data can be appended to a message. This segment is variable up to a maximum. There is a separate data transfer facility for moving larger amounts of data [70].

Consequently, the above formulation does a syntax check within the context of the called procedures parameter specifications, ie, placing the correct number of 32 bit values on its "calling stack," and calling the procedure in that context. Such a remotely called procedure should then validate the parameters within the semantics of its properties, then execute and return a message to the caller.

For some applications it is necessary to implement the more extensive support of a chosen base language's syntax and semantics. Here programming issues such as type checking and parameter parsing must be done. The V system, for example, uses this for its remote virtual graphics terminal service (VGTS) calls. Recognizing that for interprocess communication and kernel calls a simple synchronous message exchange will do, and that for more complex applications programming language considerations must be handled is important for both efficiency and ease of implementation. Certainly, distributed kernel interaction must be simple and fast if it is to be transparent to the system as a whole, and the "process world" if you like can be defined quite easily within the file constructs that such a messaging scheme easily supports. After all, a process can be viewed as a file open for read and execution, and complicated parameters such as strings and records can be passed as a data stream when necessary. Here one simply creates a data stream pipe between two processes and allows them to send data in buffers as their applications require. Pipes can be viewed as LW System supported *standard I/O* files, and read/write requests on those files. In these latter instances type checking, if necessary, can be done in the caller/callee context thus minimizing the overhead to those contexts where it is required. Thus, the VGTS application could be structurally imposed on top of process to process pipes with the parameter passing, and type checking synchronized by the processes involved.

The LW Kernel uses this interprocess message protocol to implement those operations necessary to send, receive and forward messages between processes as well as for creating, querying, and destroying processes throughout the distributed system. This protocol is transaction oriented, each message a send/reply pair and has less load impact on client/server communications then TELNET with its continuous "sub-connection" exchanges used to maintain an open connection state. This points towards a more robust and responsive distributive system when multiple clients are running processes on the same servers.

Protocols - Uniformity Across Vendors

Underlying all network I/O must be a network protocol for packet transfer between cooperating hosts. At SUMEX we have had long term experience with several such protocols; PUP/BSP, PUP/EFTP, IP/TCP, IP/TFTP, IP/UDP, and NS/SPP are those most commonly used on our LAN. PUP/BSP and IP/TCP have been used to implement both FTP and TELNET, PUP/EFTP is an Easy File Transfer Protocol on top of PUP used for boot like services, IP/TFTP is a Trivial File Transfer Protocol which uses IP/UDP datagrams, and NS/SPP is a Sequenced Packet Protocol similar to PUP/BSP and is used for FTP and TELNET. In the past we have elected to write servers for each new protocol in order to accommodate both vendor hardware and systems software. This was necessary because no one protocol has been supported on all such systems.

We are pleased that the Department of Defense IP protocol family is now supported on all hardware/operating system configurations at SUMEX and on those we anticipate purchasing in the future: IP software is available on the XEROX 1100 series workstations as of the Intermezzo system release, on Symbolics systems we have been a beta-test site for their IP software since their 5.1 operating system release, and we will be a beta-test site for the TI Explorer IP software this August. Similarly, IP is supported on all of our UNIX based file servers, and the LAN gateways route all IP datagrams.

There has been a great deal of deliberate effort at Stanford and SUMEX to enforce IP as a standard protocol for new software development. This was motivated by its broad acceptance and the growing number implementations throughout the networking and vendor communities. This does not imply that we will abandon the other protocols but rather since we are seeking to have *uniformity across all vendors* with this proposed distributed operating system we are choosing to implement it on top of the IP protocol family.

In particular we are going to continue in this direction and use the IP/UDP (User Datagram Protocol). We have benchmarked all of the protocols in the above set with respect to their implementations on each of the workstations and file servers we now use. FTP using IP/TCP and PUP/BSP perform similarly on unloaded systems. They both peak at about 200K bits/sec, and this maximum is really workstation/CPU limited rather than communication bandwidth limited. On a moderately loaded UNIX based file server PUP/BSP performance begins to degrade much more rapidly than IP/TCP since the latter is implemented in the UNIX kernel and the former is not. This results in redundant copying of both the data and datagram header information from kernel to user space for the PUP/BSP code, and thus, its performance varies inversely with the system load.

The XEROX 1100 series workstations use PUP/Leaf for random file access. With Intermezzo PUP/Leaf achieves a maximum transfer rate of about 40K bits/sec on 1108's and 80K bits/sec on the 1132's. We wish to achieve transfer rates in the neighborhood of 200K bits/sec for such file access. We feel that the 1100 series are currently limited by their single priority level round-robin scheduler. Weighting all processes equally is disadvantageous in this case since the emptying of the packet input queue is handled by one of these processes, and this process is the critical path with respect to maximizing transfer rate. Using the TFTP based on IP/UDP we managed to achieve 67K bits/sec on an 1108 and 90K bits/sec on the 1132. This is quite encouraging since TFTP uses a simple packet/acknowledgment exchange for data transfer. By augmenting this algorithm to allow multiple outstanding packets we ought to achieve 100K bits/sec on the 1108's and perhaps 150K bits/sec on the 1132's within the InterLisp environment. This expectation is not overly optimistic since PUP/BSP was recently rewritten for exactly the same reason. We increased the outstanding packet window from one to four and the maximum transfer rate went from 67K to 200K in the mesa environments on these same systems. Anticipating the preemptive scheduler that XEROX is now working on,' there is no reason why the InterLisp environment cannot approach the mesa environment in these respects.

Finally, PUP's and NS packets are limited to 532 and 546 bytes of data respectively, and with IP/UDP we can essentially double this size and send packets with 1024 data bytes. This along with multiple packet windows should put the transfer rate in the neighborhood of 300K bits/sec on these systems. It is worth noting that such an IP/UDP scheme has been used between M68000 workstations on a 10-megabit net achieving a file transfer rate of 800K bits/sec. Also, the V systems downloading scheme which is encapsulated in IP/UDP datagrams achieves 400K bits/sec between a M68000 and a VAX11-780. These tests were done on lightly loaded systems.

IP/UDP is a very simple protocol with very little processing overhead. Unlike IP/TCP which allows for packet fragmentation and reassembly, IP/UDP packets are integral throughout their lifetime and ideally suited for LAN applications. Another worthwhile feature is that the simplicity of the protocol requires very little kernel management, and consequently makes multiple client/server interactions quite feasible on even a single host server without impacting either the server or distributed system loads.

The Distributed Operating System Resource Manager

The distributed operating system resource manager is an intelligent-agent that will run on a Lisp workstation with the LW Kernel. It is intended to behave in much the same way as a "pie-slice" scheduler does on a mainframe operating system except that it will have a knowledge base to govern its decisions. In its knowledge base will be a representation of the current partitioning of the distributed system and dynamic load statistics of each host in each class in the partition. Additionally, it will attempt to learn about not only each client/server type but also each process type. Different processes will impact each client/server in different ways. Understanding and dynamically adjusting to the impact processes will have on the distributed load is a difficult problem and its solution is essential in the development of the resource manager. Graphics tools for examining knowledge representations of system load with respect to clients, servers, process types and partitioning of the distributed system will be provided.

When a client wishes to run a process on the system it will query the resource manager for the best server on which to run that process rather than query-broadcast on the distributed system itself. In a simple scenario, the resource manager will select all of those servers with respect to the willingness-partition, and then select the least loaded server from this list. If a client wishes to either migrate a process from itself to a server in the dynamically partitioned system, or have a server in its class in this partition download and run a process for it, the resource manager can then mediate this transaction. It will know which servers in the class are willing to run such a process, and from this list select the server that is least loaded or better yet, maintain idle-time schedules of all such hosts and select the host that will be idle for the duration of the process execution if possible.

Certainly, centralizing the functionality of a resource manager will allow us to more clearly understand the distributed system and its interactions. Graphical representations of system, and server loads, and response to this load by the creation or destruction of processes will give us innovative insight into just what rules are necessary to manage this distributed resource. Each particular process will impact a particular server in a way that is a function of that server's hardware and operation system, and the complexity of the process and its resource requirements. Consequently, the knowledge base and rules relating its members will grow with respect to each process type as well as each server type, and as the resource manager begins to understand their interactions. Also, simply having a resource manager with a server that knows which parts of the distributed system are working at any given time will prevent a great deal of user frustration. Given the large "granularity" of processing time and the relative infrequency of communications between these processes will initially allow us to develop such a manager on an independent LISP machine. If we reach the point where the trade-off between processing time and communication load becomes critical it may be desirable to install the resource manager in several or all of the nodes in the distributed system.

Just how an intelligent-agent resource manager will behave under all instances of distributed system interaction is an excellent area for AI/distributed operating systems research.

Implementation

Initially, we plan to implement the LW Kernel on Xerox 1100 series workstations. These systems have a remarkable programming environment, and a large set of networking debugging tools to facilitate the development of the distributed kernel. We also have an excellent working relationship with the systems software group at Xerox. This will be helpful for timely acquisition of the sources for the system as well as information about any problem areas we may encounter.

The early development of the LW kernel will run in two parallel phases. The underlying IP/UDP random access file transfer protocol and the LW Kernel's interprocess message protocols (IPMP) will be done first. The former will ultimately replace the PUP/Leaf service which is a major resource drain on our UNIX file servers. This will begin to then move the 1100s towards the uniform IP network standard. Also, random file access will be an integral part of the LW System's standard I/O file access, and data transfer mechanisms. Uniformity and optimization of file transfer is important if the distributed operating system is to be responsive when servers are loaded. The LW Kernel interprocess message protocols are central and necessary for all distributed system operations. The latter and random access file I/O are initially independent mechanisms and can be developed separately.

Since the LW Kernel's IPMP are transparent with respect the the distributed system, the entire mechanism can be written and debugged on a single workstation without network interaction. The kernel runs as a process on each host in the system, and dispatches messages intended for itself and any other host in the system. All that is required to send a message to the kernel is access to the kernel's "token," and this is globally available on the workstation itself. So, initially one writes the kernel process and the primitive message dispatching stubs, *Send*, *Receive*, and *Forward*. This will be followed by process operations like *CreateProcess*, and *DestroyProcess* along with *SetProcessID*, *GetProcessID*, and *GetProcessToken*. At this time all created processes including the kernel process will be able to send/receive messages to/from each other on the workstation in exactly the same way that it would be done if these processes were distributed. Then we implement the LW System I/O protocols by beginning with the pseudo-device pipe server. A pipe is a unidirectional flow-controlled communication channel between two processes using the standard I/O protocol [37]. It is implemented via sending messages to a *pipe-server* process. This server may reside on the local host or any other host in the system so the implementation generalizes rather nicely. Each pipe is a file instance and has one reader and one writer. This may be of course the same process.

The above is written on top of the resident process scheduling and window managing functions as well as the file system. Thus calls for creating and destroying processes, opening, managing, and closing windows as well as for file system directory management already exist. The kernel process allows us to simply distribute this functionality. Once this is working on a single workstation, the software will then be loaded onto a dual system and the kernel will then use the network so that we can then run processes in a two host distributed model and debug the IPMP in this environment. Once the underlying mechanisms discussed above are working this step should be fairly easily accomplished. It reduces to insuring that the kernel's message queue can be filled via the network. The mechanisms involved are identical except that a message must be further encapsulated and then sent on the network, and the underlying network software already works.

Based on this work, it will then be appropriate to develop applications using the distributed operating system and the IP/UDP random file access protocol. The following sections discuss some of the initial applications we will explore. In later years we will work on other applications like remote file management, network performance monitoring, and more intelligent interfaces for users to systems.

Mail System

Providing an effective and responsive mail system is one of the primary goals of any modern computing environment. Most users spend at least one hour each day reading and responding to their network mail, and this now generally takes place on either the DEC 2060 mainframe or one of the UNIX systems at SUMEX. What is frustrating is that during prime computing time the routine perusal of ones mail often becomes a very time consuming task because of the load on these mainframe systems. In fact at any given moment during this time 50% of the users can be found running MM, the system mail program, on the DEC 2060. Yet, mail is a very natural function to run on an individual's workstation. To this end, it is one of the first applications directed at the LW distributed operating system.

Indeed, it makes a great deal of sense to have as much of the mail processing as possible be done on a user's workstation. This processing can be partitioned into four categories: Mail storage, Mail retrieval, Mail reading and composition, and Mail delivery. Mail storage can be done both on the local workstation and file servers. Mail retrieval involves transactions between the workstation and the storage medium. Mail reading and composition can be entirely done on the workstation, and mail delivery involves transactions between the workstation and a domain name server for address resolution, and a mail spooling service for the caching and final delivery of non-LAN mail such as that going to a site on the ARPA net and not on the LAN. Let's address each of these four areas.

Mail Storage: By mail storage we mean the storing of all *unread mail* as well as *read mail*. Initially unread mail will arrive at a file server or servers in the user's mail delivery path. This is usually accomplished by alias files on hosts that may receive mail for a person or mailing-list but on which this mail is not kept. Alias files provide a forwarding mechanism to the ultimate destination repository. In any case the mail ultimately arrives at a destination file server known to the person's resident mail process. As each letter is read it is up to the reader's discretion as to whether or not it is to remain on the workstation or be returned to the appropriate file server. Records of all mail still in the system will be kept on the file server under the user's mail account. Rereading a letter that is on the workstation can be short-circuited to remove the file server from the loop. The primary activity in this area is then the moving of mail between the user's workstation and a file server(s). This can be expedited with minimum overhead using the high transfer rate IP/UDP file service to stream the data between a client and server. Indeed, at 300K bits/sec most letters will be moved in a fraction of a second with very little impact on either the client or the server.

How this mail is arranged on the server is an important consideration if access is to be efficient and the services per letter multidimensional. On each server in the user's mail path the user will have a mail directory associated with his address at the server. The directory will be organized into a *mail spindle file*, *mail header file*, *mail keyword file* and *mail folder files*. The latter may in fact be a sub-directory on hosts supporting such a scheme.

The spindle file will have an entry for each letter. Among other things this entry will have a pointer to its header in the header file, the folder where the letter is stored, status bits indicating the state of the letter. For example: Such bits could be *seen*, *unseen*, *new*, *deleted*, *answered* and *alarm*. The alarm bit is then associated with a time-date when the user wishes to see this message's header again. Each entry has the date it was read, and the date it was answered. Finally, there will be a bit field describing key-words the owner can associate with each letter, and the associated keyword file of actual keywords. The spindle file itself will be prefixed with a header. This header will at least include time-date stamps of the last read and write access to the owner's mail, a pointer to the entry for the oldest new mail, ie, mail that has arrived since the last time the mail was read, and a pointer to the next alarm entry.

Thus, when a user first runs the mail process on his workstation the process interrogates the mail server(s) in the user's delivery path. Each such server quickly gathers the headers of the newly arrived mail, checks for any alarms that may have gone off, incorporates these headers into a message and sends them to the users workstation. The actual header file can be built in background mode as mail arrives and system resources allow to minimize this processing. Note that none of the text of the mail which is the bulk of the data has yet to be touched in this transaction.

Mail Retrieval: Mail retrieval is accomplished with a workstation client and mail/file-server server. The client is mouse driven by at least a selection process that displays active letter headers in a window. The headers which appear in this window are selected by the user with a mouse/menu interaction. When the mail client is started it probes those servers in the user's mail-path for "new" mail, ie, letters that have arrived since the last read-access to the mail spindle file. These headers will be listed in a window which has mouse interaction defined for each such header. One will be able to change the displayed headers by commands like *headers* since <date>, from <string>, to <string>, subject <string>, and all. Reading the letter associated with a header then transfers the actual text of the letter from the server to the client with a read-mail transaction, unless the letter has already been transferred to the client and is cached there. This transaction causes the read-date stamp and "seen" bit to be updated in the spindle file entry.

Mail Reading and Composition: Mail commands such as read, answer, set alarm, delete, and copy, key off of header selection. When one reads a letter it is then read from the server to the client by a read-letter transaction. The text is displayed in a window and can be scrolled as well as edited. All text editing and composition is done on the local workstation. When one answers a letter immediate destination host address recognition is mandatory. This can be accomplished by requesting host address validation after the addresses have been typed. One can use the domain name server and LAN name servers for this purpose. It also makes sense to cache known host names locally and if for some reason the name servers do not reply this list can be used for a second guess. If all else fails, then one should simply attempt to deliver the letter. If in fact the address is not valid, then this will be noted when the letter is returned to the sender as undeliverable.

Mail Delivery: Once a letter is composed and the sender requests it to be delivered, it will be spooled on one of the file/mail servers. These servers already have all of the knowledge necessary to deliver any letter to a known host. Mail delivery is done in background on these servers by a low priority process. An attempt should be made to spool the mail on the server with the smallest mail queue and such a mail-queue-size query message will be sent to those servers that respond to a request-to-send-mail broadcast. Each host can override the latter broadcast by simply remembering which servers responded to earlier broadcasts, and thus maintaining a mail-delivery-path for directing mail-queue-size queries. The system resource manager will maintain current mail delivery information. Often a host in a mail-delivery-path is down for some reason, and mailers will continuously attempt to shrink their growing mail queues by uselessly badgering this host. It makes sense to be able to request server-downtime and alternative mail routes from a resource manager. If there is no alternative route, the mail client/server can periodically check until the host comes up rather than try and send mail to a down host which amounts to useless network traffic.

Ultimately, a mail-server process ought to be able to run in the background on personal workstations, and mail could then be delivered directly to that host for those users who desire such a service. This will then take the file/mail-servers out of the mail storage and retrieval loop for such hosts. Mail is simply sent directly to the workstation that has a registered address in the domain name server tables. The mail is then retrieved and read "as if" it had already been copied from a remote file/mail-server. This latter mechanism is part of the initial design. As mail accumulates on such a host, the user will be able to take advantage of those already existent file/mail-server processes to maintain mail archive directories remotely so that old mail can still be examined in the client/server role.

Virtual Graphics Terminal Service

Virtual graphics terminal service (VGTS) allows the display of structured graphical objects on a workstation running the V system [37]. We have already indicated the power of this set of tools. While running V on a small and inexpensive workstation located either at home or on the LAN, or anywhere that has TELNET access to the LAN on which a personal Lisp machine has a TELNET server running, one can then access that Lisp machine and drive the graphics display of the smaller workstation from the Lisp machine. Geographic proximity of such a Lisp machine is then moot.

As the ratio of researchers per Lisp machine increases it is no longer possible to guarantee Lisp machine cycles to everyone during prime computing time, and a means for remotely accessing these machines in graphics mode becomes mandatory. VGTS satisfies this need perfectly. In order to install the software tools necessary for remote

VGTS access there are two requirements: First the ability to TELNET into a Lisp machine is necessary. Second, the interfacing of VGTS primitives with the current graphics/window calls on the Lisp machine. We address each of these below.

Not all of the current Lisp machines have servers which allow the establishment of an incoming TELNET connection. Currently, only the Symbolics machines have this property. What is necessary here is to modify the outgoing TELNET code where applicable so that it can also run as a server process. This is really a straightforward task. What is interesting here is just how to globally establish that the incoming data stream is to be interpreted by the Lisp machine command executive, and then all output characters are to be sent via the TELNET stream and not to the local graphics display stream. This redirection of I/O streams is well within the scope of all of our Lisp machine operating systems.

The central concept of VGTS is that application/client programs should only have to deal with creating and maintaining abstract graphical objects [37]. The actual viewing of these objects is done on the workstation running V. For example: To create a view or window on a workstation/server running V from a Lisp machine/client two things are required. The client calls a routine to remotely create a file, the *structured display file* (SDF), which will then contain descriptions of graphical objects. Each such object has an client assigned item number associated with it in the SDF. This SDF is then associated with what is commonly referred to as a window by first calling a routine to create a virtual graphics terminal(VGT) associated with this SDF, and then calling a routine to create a view on this VGT. A view is seen as a white area on the screen with a border. Thus a VGT/SDF pair can have multiple views associated with it. And one can have multiple VGT/SDF pairs at any one time as well as more than one VGT associated with the same SDF. The mapping of VGTs to SDFs need can be but not be one to one. Each of these calls involves little more than the passing of a few data bytes between the client and server.

Once the SDF/VGT relationship is established and a view is created on the server, then graphical objects can be created by adding them as items to the SDF by opening a *symbol* for editing and adding an item to that symbol in the SDF. An SDF then contains symbols which are in turns lists of items. An item itself can also be a symbol. These objects can then be displayed in the view(s) associated with the VGT. Thus, objects can appear on several VGTs at the same time. A client can also create menus on the server and then interrogate the actions implied by those menus via mouse buttoning. In fact one can actually query a mouse event within a view and receive back not only the buttons that were touched but also the VGT number and view coordinates of the cursor position itself, or a list of objects that are near the cursor position. This allows the client to interrogate, as well as edit viewed objects remotely. One need not maintain a great deal of information about objects on the client. In fact, one needs only the VGT number, SDF number, which are returned by the server at when they are created, and the item number which is sent when items are added to SDFs. A client can then inquire about this item and receive its definition as a reply. Thus, VGTS is designed to maximize what is done on the server by maintaining the SDF database and allowing detailed queries about its contents which can for the most part be driven by user/mouse interaction with their graphical representation.

The VGTS has a resident view manager for moving, zooming, opening, closing, and creating new instances of views associated with VGTs. Consequently, the view overlaying, manipulating and trimming algorithms do not impact the client. A list of the current VGTS object primitives is as follows:

Filled Rectangle	These can be filled either with gray scale shades or stipple patterns or black and white monitors, and with colors on color monitors.