MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 561 December, 1979

Using Parallel Processing for Problem Solving

William A. Kornfeld

ABSTRACT. Parallel processing as a conceptual aid in the design of programs for problem solving applications is developed. A pattern-directed invocation language known as Ether is introduced. Ether embodies two notions in language design: activities and viewpoints. Activities are the basic parallel processing primitive. Different goals of the system can be pursued in parallel by placing them in separate activities. Language primitives are provided for manipulating running activities. Viewpoints are a generalization of context mechanisms and serve as a device for representing multiple world models. A number of problem solving schemes are developed making use of viewpoints and activities. It will be demonstrated that many kinds of heuristic search that are commonly implemented using backtracking can be reformulated to use parallel processing with advantage in control over the problem solving behavior. The semantics of Ether are such that such things as deadlock and race conditions that plague many languages for parallel processing cannot occur. The programs presented are quite simple to understand.

ACKNOWLEDGEMENTS: This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Office of Naval Research under Office of Naval Research contract N00014-75-C-0522.

PREFACE

This work is a shortened version of my master's thesis [1] submitted in the Spring of 1979. The original text covered several ancillary topics of which I was not satisfied with either the presentation or technical content. I had originally planned to complete these parts in a short time and produce an expanded version as a technical report. This task proved harder than I first realized. Consideration of many of these additional topics has led to a gradual evolution of the ideas in directions that differ in emphasis and specific technical detail from those expressed here. I have decided it best to take what I felt was the stable core of the ideas and issue them at this time.

The topics that have been removed include:

(1) *Manipulative viewpoint inheritance*, the placing of filters between viewpoints so that some subset of the assertions will be inherited. This mechanism can make use of justifications on the assertions in a viewpoint to control the filtering. More recently I have been thinking of generalizations of this concept in which the contents of a viewpoint can be expressed as a *function* of the contents of its parents, not simply a subset. I plan to develop this idea in the future.

(2) *Use on parallel hardware.* I had developed a scheme for implementing Ether on multiprocessors. Its only purpose was to show that the *broadcast* primitive could be implemented on a multiprocessor system without having to invoke a network broadcast. It was not well-developed and I thought it best to drop.

(3) *Conjunctive Subgoals with Shared Variables.* This is a very important topic that I will develop more completely in a future paper.

In addition, I am currently developing with Hewitt [2] a paper on the relationship between the problem solving philosophy embodied in Ether and the landmark works of modern philosophers of science such as Popper, Lakatos, and Kuhn. We feel there is much similarity between the notions of conjecture and refutation in science as expressed by Popper and the parallel execution of many activities including opponent activities. I was largely unaware of this relationship when the work reported here was done.

The term *viewpoint* that is used in the present work replaces the term *platform* that I had originally used in my thesis and in the paper describing Ether that appeared in IJCAI6.

ACKNOWLEDGEMENTS

Special thanks go to my thesis supervisor, Carl Hewitt, for his support, keen interest, and hyperopic view of the computer field.

Jerry Barber, Randy Davis, Roger Duffey, Ken Forbus, David Levitt, and Luc Steels have been kind enough to carefully read earlier drafts of this work and make many comments that have enormously aided the presentation.

In addition to these, there are many people I would like to thank for contributing ideas in conversation, among them: Beppe Attardi, Jon Doyle, Mike Genesreth, Harold Goldberger, Ken Kahn, Bob Kerns, Henry Lieberman, Marvin Minsky, Vaughn Pratt, Maria Simi, Barbara Steele, Richard Stallman, and Sten-Ake Tarnlund.

My deepest appreciation goes to all the folks at the Artificial Intelligence Laboratory and the Laboratory for Computer Science who have created an unparalleled working environment, a truly exciting place to be.

CONTENTS

Chapter I Introduction

My interest is in studying possible uses of parallel program architectures for the solution of problems in artificial intelligence. What distinguishes this class of problems from others is the volume of "nonessential" computation that gets done. Programs spend the bulk of their time *searching* through spaces of facts and methods for one that might possibly be useful.† This search often takes place at many levels simultaneously; in determining whether a selected fact or method is useful may entail a vast search through a different subspace of facts or methods. How this is orchestrated within a program is usually referred to as its *control structure.* The intent of a control structure is to avoid as much of this search as possible, although a certain amount of it seems necessary. There have even been some researchers suggesting that it is not possible to curtail this search very much, at least for certain special cases of reasoning [3, 4], and have suggested solutions involving vast amounts of parallel hardware that can decide many of these questions by use of sheer computation power. While I don't wish to take exception to their conclusions, only leave these questions in abeyance, this is not the position taken by the present work. The emphasis is not on "brute force" solutions, but on techniques for gaining more programming flexibility and greater control over the problem at hand.

This work builds on various ideas in the problem solving literature and combines them with some new ideas about parallel computation. The synthesis will be a new pattern-directed invocation language known as Ether. Ether follows in the tradition of Planner in having a collection of assertions representing facts about the world and procedural objects that interact with these facts via pattern matching.

Two important subcomponents of Ether are a language for talking about *activities* and a hierarchy of *viewpoints* for structuring data. The notion of an activity is intuitively similar to the notion of a process, that is a locus of control with some purpose. It is not as rigid as the concept of a process as might be defined by allusion to a Turing or Von Neumann machine model. The concept of process for these models can be thought of as a totally ordered sequence of state changes to a tape or other kind of memory. This is not the case with an activity. It may be the case that several assertions are *broadcast* (i.e. added to the database) or several procedural objects executed by one activity without there being any particular ordering between them. The kinds of things that might become activities are "Do antecedent reasoning on the facts in the following viewpoint," "Attempt to achieve the following goal," or simply "Run the following code." Activities are objects that can be talked about through the Ether database and acted upon by special primitives. A kind of thing one might do to an activity is to prevent its continued execution. This might be desirable if the purpose of the activity has already been accomplished. One might also change the rate at which an activity is working. In Ether the host machine is thought of as a finite resource that has a certain (constant) amount of power.

---

† The problem of search isn't the only distinguishing feature of artificial intelligence research. There are many serious issues of data or "knowledge" representation that are unique to the field. While many researchers today would consider these to be the only serious concerns, I think this attitude has led to the creation of a number of sophisticated systems for the representation of knowledge with no clear ideas on how to use them in programs. In contrast to this, we will be concerned only with questions of program organization.

The power can be distributed amongst running activities in whatever way seems appropriate at the time. If some newly discovered information suggests that one way of accomplishing a goal is more likely to succeed than another, the amount of processing power being used by the former activity can be increased to the detriment of the other.

Many AI languages reason by creating and manipulating world models inside the machine. Context mechanisms allow multiple inconsistent world models to reside in the database concurrently. Often many of the world models share much of their structure. Context mechainisms make this economical by supporting an inheritance between contexts. These mechanisms, as usually conceived, do not allow processing to happen in more than one world model at a time.. A generalization of the context idea is the *viewpoint*. Viewpoints have an inheritance structure similar to contexts but unlike them, there is no restriction on the number of viewpoints that may be available at one time -- all may be processed concurrently. Many of our examples depend on the ability to build many incompatible but concurrently accessible world models.

Throughout this paper we will present a number of control structures that can be viewed as parallel generalizations of well known techniques, such as forward and backward chaining, OR and AND subgoals, depth first and breadth first search, and goal filtering. The parallel techniques tend to be more flexible.

One facility of most other procedural deduction systems that Ether has chosen to leave behind is *automatic backtracking*; although useful in certain circumstances, the purpose of this work is to study what could be accomplished by allowing the user's program to explicitly control the search. Even though backtracking is not present, the system is set up so that the program is not committed to "believing" hypothetical assumptions just because they were once made. Why these are different ideas will become clear later on.

In this work I will not be concerned with the question of whether the (parallel) programs described are to be run on a conventional sequential machine *or* a specially designed parallel machine. However, the language to be described is designed so that an implementation on parallel hardware would require little revision of the language constructs. The arguments for parallel processing advanced are as relevant to conventional serial machines as they are to more advanced parallel architictures.

The contribution of Ether can best be understood in a historical context. In Micro-Planner, many theorems could be specified for accomplishing some purpose. They were chosen non-deterministically by the interpreter. The inability of the user to order the application of methods made programs semantically clean but led to grossly inefficient searches via the automatic backtracking mechanism. The Conniver language was a procedural deduction system made to look and act more like a conventional programming language. The driving motivation for this was to curtail the needless search engendered by backtracking. The end result was a system without a rich semantic model and thus the complexity barrier was reached very quickly. The presence of explicit parallelism in Ether allows both goals to be achieved. I believe Ether programs to maintain the clean semantic model of Micro-Planner while allowing search to be tightly controlled.

Chapter II Combinatorial Implosions

This chapter introduces a phenomenon that is possible in parallel programming environments. When many *activities* are running concurrently in an attempt to solve some problem, some activities can produce information that obviate the need for others or help them solve their problem more quickly than if they were run in isolation. We would not reap the benefits of this sharing of information in a sequential system because the activities must be run in some order decided before this information has been produced. We will use the term *combinatorial implosion* for this unpredictable and useful interaction between running activities.

Discussion of this phenomenon as a justification for parallel processing is curiously scant in the literature.† The usual argument for parallel schemes is that problems can be broken into parts that are separately solved on separate processors, thereby finishing faster than could have been accomplished on a single processor. Combinatorial implosions, however, can occur just as readily on time shared sequential processors as on truly parallel machines. The main example of this chapter is discussed *assuming* it will be run on a single time-shared processor.

2.1 An example problem

This problem is excerpted from the PROTOSYSTEM-1 automatic programming system [6]. It is presented in a somewhat stylized format that is functionaly identical to an a tual problem occurring in the design phase of the automatic programming system.‡

We are presented with a set of (boolean) predicates $\{P_1, P_2, \ldots, P_n\}$ and a (boolean) predicate $\mathcal{P}$, such that $\mathcal{P} \supset P_1 \vee P_2 \vee \ldots \vee P_n$. The goal is to determine all sets $S \subset \{P_1, P_2, \ldots, P_n\}$, such that $\mathcal{P} \supset \bigvee_k P_k$ for each $P_k \in S$ and for which there are no proper subsets R of S such that $\mathcal{P} \supset \bigvee_k P_k$ for each $P_k \in R$. In other words we want to find the smallest subsets of $\{P_1, P_2, \ldots, P_n\}$ that cover the predicate $\mathcal{P}$. Note that there can be more than one such subset.

In order to simplify the following discussion we will make two definitions.

1. A tuple of predicates $\{P_{i_1}, P_{i_2}, \ldots, P_{i_n}\}$ *works* if $\mathcal{P} \supset P_{i_1} \vee P_{i_2} \vee \ldots \vee P_{i_n}$.

2. A tuple of predicates $\{P_{i_1}, P_{i_2}, \ldots, P_{i_n}\}$ is *minimal* if there is no $S \subset \{P_{i_1}, P_{i_2}, \ldots, P_{i_n}\}$ such that S *works*.

---

† A similar notion known as the *acceleration effect* [5] was developed independently.

‡ In PROTOSYSTEM-1 terminology the problem is one of generating *driving datasets candidates*.

2.2 Sequential Solutions

In terms of the above definitions, we are looking for all subsets of the given set of predicates that *work* and are *minimal*. There are two algorithms that we, as programmers, might pick as solutions to this problem; they are known as **Top-Down** and **Bottom-Up**. Descriptions of these algorithms follow.

2.2.1 Top Down

Create a **results-list**, initially null.

Top-Down($\{P_1, P_2, \ldots, P_n\}$) =

1. if for every **j**, where $1 \leq j \leq n$, $\{P_1, \ldots, P_{j-1}, P_{j+1}, \ldots P_n\}$ does not work add $\{P_1, P_2, \ldots, P_n\}$ to the **results-list**.

2. else execute **Top-Down** ($\{P_1, \ldots, P_{j-1}, P_{j+1}, \ldots, P_n\}$) for each of the $\{P_1, \ldots, P_{j-1}, P_{j+1}, \ldots, P_n\}$ that *works*.

3. Return the **results-list** as the answer when all computation is completed.

2.2.2 Bottom Up

Create a **results-list**, initially null, and a counter **k**, initially 1.

Bottom-Up ($\{P_1, P_2, \ldots, P_n\}$) =

1. Generate all k-tuples of the P_i and remove all of the k-tuples that contain a proper subset that is on the **results-list**. Check each of these to see if they work, and if they do add them to the **results-list**.

2. Increment **k** and iterate until $k = n$, then stop and return the **results-list**.

Both algorithms are optimal in the sense that no test for workingness (a very expensive operation) is ever performed that could be logically eliminated; no algorithm could be created that will always require fewer tests than either of these.

2.3 Parallel Algorithms

2.3.1 Parallel Algorithm I

The two algorithms have very different characteristics. **Top-Down** will work faster if the minimal working subsets are large with respect to n, and **Bottom-Up** will work faster when they are relatively small. There is no way to decide which one will be fastest for a given problem short of running one of them. The variability is sufficiently great that we could produce a faster algorithm on sequential machine by running them concurrently with one another by time-sharing and waiting for the first to finish with the result. This is one (albeit weak) form of combinatorial implosion. The timing variability between methods need only be high enough so that on the average twice the time of the fastest to finish is less that the average speeds of both.

2.3.2 Parallel Algorithm II

In this section we will improve on the algorithm of the preceding section by allowing the two subactivities to pass each other information. We note two facts:

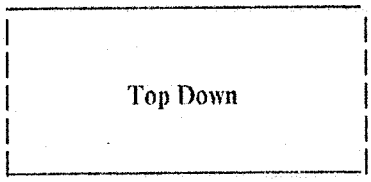
1. If $\{P_{i_1}, P_{i_2}, \ldots, P_{i_n}\}$ does not work, then any subset of it will not work.

2. If $\{P_{i_1}, P_{i_2}, \ldots, P_{i_n}\}$ works then any superset of $\{P_{i_1}, P_{i_2}, \ldots, P_{i_n}\}$ will work and not be minimal.

As **Top-Down** is running, it produces as computational by-products numerous sets that don't work and in order from largest to smallest. By property 1 above it can be immediately deduced that all subsets of these sets will not work and these can be eliminated from immediate consideration. This fact is of course implicit in the design of the **Top-Down** algorithm, but can be of great use to **Bottom-Up**. As **Top-Down** discovers sets that don't work they can be passed to **Bottom-Up** and used to prune many sets from possible consideration. A byproduct of the running of **Bottom-Up** is the enumeration of sets (in increasing order) that work. By property 2 these can be used to prune all supersets of the set from consideration by **Top-Down**.

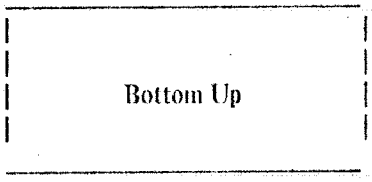
The parallel algorithm we envision has these two activities running concurrently and passing results to each other as they are discovered. A block diagram is shown in figure 1. Arrows in the figure show conceptual flow of information. As a process discovers new sets that it should report to the other process, it broadcasts this fact. Whenever a fact is learned that eliminates the need for what the activity is doing at the moment its work is halted and a relevant one is begun. This algorithm is easy to express in Ether, although difficult or awkward using other parallel programming methodologies.

Fig. 1. Block Diagram for Parallel Implementation



Sets that work and are minimal

Sets that don't work



Chapter III Basic Ideas

3.1 Pattern Directed Invocation

The principle feature of pattern-directed invocation languages are a large, continually changing collection of *assertions* that represent facts of importance to the problem solver and some means of procedure invocation based on pattern matching involving this collection of assertions. In Ether,† these define the two principle operations: `broadcast` and `when`.

We choose the term *broadcast* for the operation that adds new assertions to the ones already known. The reason we are using the term *broadcast* (and avoiding the term *database*) is to supply a certain conceptual model. A *database* is often thought of as a data structure in which items are inserted in some definite order. It might matter to the overall behavior of the system in what order two assertions were entered. The database itself often has to ensure its own consistency. If this database is used as a joint repository of information used by many activities running in parallel there are many opportunities for unforseen and undesirable interactions to occur between these running activities. The standard conceptual model of a *database* is at too low a level. The pattern-invoked procedures, called *sprites*, are thought of as watching for broadcast assertions matching their patterns. If one of them is invoked it can broadcast new assertions to other sprites or create new sprites.

3.2 What Sprites Are

Sprites consist of two parts, a pattern and a body. They watch for assertions to be broadcast that match their patterns. If a sprite's pattern successfully matches an assertion, the body of the sprite is executed in the environment of the match. Sprite bodies principally contain two kinds of constructs: more sprites that are activated and commands to broadcast new assertions to the collection of sprites.‡

An example of a sprite that serves the function of an antecedent theorem is:

```
(when (ON =x =y)                          ;When a block is on another block
      (when (OVER y =z)                   ;and the second block is over a third,
            (broadcast (OVER x z))))      ;assert the first block is over the third
```

The pattern of this sprite will match any assertion with three elements that has `ON` in the first position. When this sprite is triggered it creates a new sprite (as the sole action of executing its body). If the assertion `(ON A B)` is broadcast, this sprite will create a new one of the form:

---

† "Ether" (according to many noted 19th century physicists) is the name of the medium through which all information travels at finite speed.

‡ In the current implementation the bodies of sprites are evaluated by the Lisp interpreter in the lexical environment of pattern matching. Broadcast and when are ordinary Lisp functions. Other Lisp functions, such as cond, can also be used.

```
(when (OVER B =z)                          ;When B is known to be over a certain block,
      (broadcast (OVER A z)))              ;assert A is over that block.
```

If an assertion of the form `(OVER B C)` is also broadcast, the action of this sprite will be to broadcast `(OVER A C)`.

Nested sprites, are common enough to motivate a simpler notation. The patterns are all collected together into one list delimited by curly brackets. For example, the form:

```
(when {(ON =x =y)
       (OVER y =z)}
      (broadcast (OVER x z)))
```

is functionally equivalent to the one above.

Sprites obey an important property known as *commutativity*. When there is a sprite S that is capable of triggering on an assertion A, the behavior of the system is invariant with respect to the order of creation of S and A. It does not matter if the sprite was created before the assertion or vice versa for the sprite to trigger. The Ether collection of assertions satisfy another important property known as *monotonicity*. Once an assertion has been broadcast it can never be erased. The modularity of Ether code depends upon these properties.

3.3 Explicit Goal Assertions

Many pattern-directed invocation languages have specific syntactic constructs for doing antecedent and consequent reasoning (e.g. Planner's antecedent and consequent "theorems"). It was later realized that only one pattern-directed invocation facility was needed [7, 8, 9]. Sprites can serve as antecedent theorems as was shown above. Assertions that represent goals can be marked as such so that the same syntactic construct can be used for consequent as well as antecedent reasoning. deKleer et. al. [9] describes a language using explicit control assertions closely resembling the subset of Ether developed in this chapter. A natural deduction logic that bears a certain resemblance is described in Kalish and Montague [10].

A simple consequent theorem embedded in a sprite is shown in figure 2.

Fig. 2. Simple Consequent Sprite

```
(when (GOAL (MAMMAL =x))                   ;When there is a goal of demonstrating x is a mammal,
      (broadcast (GOAL (HUMAN x)))         ;Try to show x is human.
      (when (HUMAN x)                      ;If you show x is human,
            (broadcast (MAMMAL x))))       ;broadcast that x is a mammal
```

This sprite, when invoked by a goal assertion, broadcasts a new goal assertion that can be picked up by (possibly several) consequent reasoning sprites and worked on in parallel. A sprite is created that watches for an assertions to be broadcast that watches for the new subgoal goal to be satisfied. If and when this assertion

appears the result (HUMAN X) is broadcast. Sprites of this kind, that watch for results of independent activity, are called *continuation sprites* because of their similarity to continuations employed in other programming languages.

There are many advantages in using explicit goal assertions. They allow the system to reason about its goals (possibly concurrently with working on them.) There are at least four reasons why a procedural deduction system should be able to reason about its goals.

1. It is often useful for a system to determine consequents of its goals in order to evaluate the plausibility of the goals themselves. If the system can know what its goals are, then it can reason about the possible applicability of techniques aimed at accomplishing these goals. This idea will be developed extensively in section 5.3.

2. Moore [11] presents another use of the ability to have access to the goal structure. He demonstrates numerous examples of situations in which there is a goal with two OR subgoals. It is shown (because of the existence of these two subgoals) that there exists a third subgoal, the successful solution of which will signify a solution to the main goal. His examples of this are all of one form that might be characterized as the dual of resolution. (He calls it "restricted goal resolution".) If one subgoal is of the form:

$$P \wedge Q_1 \wedge \ldots \wedge Q_i$$

and the second is of the form:

$$P' \wedge R_1 \wedge \ldots \wedge R_j$$

where P and P' unify then it is sufficient to solve the goal:†

$$Q'_1 \wedge \ldots \wedge Q'_i \wedge R'_1 \wedge \ldots \wedge R'_j$$

where the individual propositions are instantiated by variable bindings resulting from the unification of P and P'. It is based on the observation that it does not matter to the main goal which of P' and $\neg$P' is achieved. Since one of P' and $\neg$P' will be true, and all other prerequisites of the respective OR subgoals are achieved, one of them will certainly be satisfied.

While this example seems somewhat artificial, (although Moore does develop the idea extensively) other, more semantically meaningful examples involving, for instance, planning can be imagined. Suppose you want to do two things (have two goals): (1) Get money at the bank and (2) Buy a book. Let's say the bank has two branches, one in Kendall Square and one in Harvard Square, you are nearer Kendall Square than Harvard Square, but there are book stores only in Harvard Square. To deduce that you should go to the

---

† In our notation, the Q'_i and R'_i are the terms Q_i and R_i with variables replaced by constants resulting from the unification of P and P'.

bank's branch in Harvard Square the reasoning system must reason about its goals.

3. A more pragmatic, though no less important, reason for having explicit goal assertions is as an easy technique for producing arbitrarily parallel processing on conventional machines. Conventional Lisp interpreters contain a significant control state; there is a non-negligible amount of time required to switch processes. If goals are created in a manner similar to the calling of functions in Lisp, a great deal of effort must be expended to allow several goals to work in parallel. By not mimicing this aspect of the host language, large-scale parallel development becomes possible.

4. A problem that can arise in (particularly, but not exclusively) parallel problem solving systems is one of ensuring that effort is not duplicated unnecessarily. If two distinct goal activities broadcast identical subgoals we would like them to initiate only one new activity, not two. This follows automatically from the invisibility of multiple broadcasts of a single assertion.† This is similar to a well-known difficulty with recursive programming as can be seen in a recursive definition of the Fibonacci function. A Lisp definition of Fibonacci is:

```
(defun FIBO (n)
       (cond
          ((= n 0) 0)
          ((= n 1) 1)
          (t (+ (FIBO (- n 1))
                (FIBO (- n 2))))))
```

Calling `(FIBO 6)`, for example, will require `(FIBO 2)` to be evaluated five separate times, and this count increases exponentially with its argument. An Ether implementation of this same function is shown in figure 3.

Fig. 3. Ether Implementation of Fibonacci

```
(when (COMPUTE (FIBO =n))                                  ;If asked to compute (fibo n)
      (cond
         ((= n 0)                                                        ;If n is 0,
          (broadcast (IS (FIBO 0) 0)))                    ;broadcast the answer is 0.
         ((= n 1)                                                        ;If n is 1,
          (broadcast (IS (FIBO 1) 1)))                    ;broadcast the answer is 1.
         (t (broadcast (COMPUTE (FIBO <- n 1>)))          ;Otherwise compute (fibo n-1),
            (broadcast (COMPUTE (FIBO <- n 2>)))          ;and compute (fibo n-2).
            (when {(IS (FIBO <- n 1>) =a)                 ;When you have a value for (fibo n-1),
                   (IS (FIBO <- n 2>) =b)}                ;and a value for (fibo n-2),
                  (broadcast (IS (FIBO n) <+ a b>)))))))  ;broadcast their sum for (fibo n).
```

Because this sprite will only respond to the assertion `(COMPUTE (FIBO 2))` once, it will only compute the answer once. Sequential problem solving systems can get around this problem because there is a guaranteed sequencing between attempts to solve a goal. Each goal activity merely has to record that it worked on the

† Ether sprites will respond to an assertion only once regardless of the number of times it has been broadcast.

goal and what the result was. A later attempt to achieve this goal first checks to see if this information was in the database. The THGOAL primitive of Micro-Planner half solved this by first checking the database to see if the goal was present; apparently Micro-Planner repeatedly attempted failing branches.

A similar problem that we easily avoid is that of the infinite goal stack. If a goal attempts to set itself up as a subgoal, work automatically stops at that point. This problem is much less serious one in parallel problem solving systems compared with sequential systems executing a depth-first search where it can cause the system to come to a grinding halt. In parallel systems an infinite goal stack only degrades the efficiency of the system.

Given the presence of goal assertions with explicit activities created to work on them in parallel, we now have the capability to compare and contrast them as they work. As work progresses new *partial results* are achieved that can enable the system to reapportion its resources. A simple example of this is a system attempting to solve the goal $(\exists x)(P(x) \wedge Q(x))$. The system in parallel attempts to find assignments to x that will make one of the predicates P and Q true. If it succeeds in finding one such assignment (say P(a)), then it should allocate more resources to working on a derivation of Q(a). Similarly, if it discovers $\neg P(b)$, it should certainly stop working on the goal of showing Q(b).

Chapter IV Activities

4.1 Creating Activities for Goals

The simple consequent sprite developed in figure 2 on page 12 has one obvious problem; there is no way to stop work on the subgoal when the main goal has been achieved. For example, if the original goal assertion was `(GOAL (MAMMAL FIDO))` and it had been achieved, i.e. `(MAMMAL FIDO)` was broadcast, there is no mechanism that would prevent work contributing to the solution of `(GOAL (HUMAN FIDO))` begun by the consequent sprite from continuing. Our solution to this problem will be the introduction of the concept of an *activity*. An activity is a *locus of control with some purpose*. We would like to create two activities, each containing all work on each of the two subgoals. The example in figure 2 is redone using activities as shown in figure 4.

Fig. 4. Simple Consequent Sprite With Activities

```
(when (GOAL (MAMMAL =x)) =activity                    ;If you want to show x is a mammal,
      (let ((subgoal (new-activity)))                 ;Create a subgoal activity.
           (broadcast (GOAL (HUMAN x)) subgoal)       ;Try to show x is human in this activity.
           (when (HUMAN x)                            ;If you show x is human,
                 (broadcast (MAMMAL x)))              ;broadcast x is a mammal.
           (when (MAMMAL x)                           ;If you learn x is a mammal,
                 (broadcast (STIFLE subgoal)))))      ;stifle the subgoal activity.
```

There are a couple of new syntactic constructs used in this example. You will notice that sprites (such as the main one) can take two elements in the pattern instead of one. Also notice that broadcasts (such as the first one in the body of this sprite) can take two arguments. The second argument in both cases is the *activity marker*. The main sprite will trigger if an assertion has been broadcast that matches its pattern *and* that assertion is part of a currently active *activity*. The main goal, if it is to enable this sprite, should be part of such an activity. The function *new-activity* creates a new activity that becomes a sub-activity of the current activity. The new activity (bound to `subgoal`) becomes the activity of the new subgoal of the main goal. The goal assertion, representing this subgoal, is broadcast with this activity as a second argument. As before, a sprite is created that watches for the the result of the subgoal to appear and then broadcasts the main result. An additional sprite is created that waits for this main result to appear, and if so, broadcasts a `STIFLE` assertion. These cause work on the created activity to halt. `STIFLE` is an Ether primitive.

At first glance it would seem that the two sprites created (the one that checks for the result and the one that does the stifling) could have been combined into one. Why have they been separated? Remember this is a *parallel* problem solving system. There may be, concurrently running with this solution attempt, other such activities with the same overall purpose. Or it may be that this fact is learned by the system in some fortuitous unexpected way. It doesn't matter. If ever the result is achieved, *regardless of how*, the activities created to achieve them will stop working.

4.2 General Schemas for OR and AND subgoals

Traditional problem solving theory presents two standard techniques of backward chaining [12] based on whether one or all of a collection of subgoals must be satisfied for its parent goal to be considered satisfied. We will present Ether templates for doing these two kinds of reasoning where all subgoals are attempted in parallel.

4.2.1 OR Subgoals

If we wanted to determine if an object is a living thing it would suffice to determine either that the object is a plant or an animal. We can say this in Ether by creating two sprites, each watching for a **LIVING-THING** goal to be broadcast as shown in figure 5. One broadcasts an **ANIMAL** goal and the other a **PLANT** goal. Appropriate continuation sprites are also created to broadcast the **LIVING-THING** assertion if either of the subgoals are achieved.

Fig. 5. Simple Or Subgoals

```
(when (GOAL (LIVING-THING =x)) =activity                ;If you want to show x is a living thing,
      (let ((subgoal (new-activity)))                       ;Start a subgoal activity.
           (broadcast (GOAL (ANIMAL x)) subgoal)            ;Try to show x is an animal.
           (when (ANIMAL x)                                 ;If x is an animal,
                 (broadcast (LIVING-THING x)))              ;broadcast x is a living ...ing.
           (when (LIVING-THING x)                           ;If you learn x is a living thing,
                 (broadcast (STIFLE subgoal)))))            ;stifle the subgoal.

(when (GOAL (LIVING-THING =x)) =activity                ;If you want to show x is a living thing,
      (let ((subgoal (new-activity)))                       ;start a subgoal activity.
           (broadcast (GOAL (PLANT x)) subgoal)             ;Try to show x is an plant.
           (when (PLANT x)                                  ;If you learn x is a plant,
                 (broadcast (LIVING-THING x)))              ;broadcast x is a living thing.
           (when (LIVING-THING x)                           ;If you learn x is a living thing,
                 (broadcast (STIFLE subgoal)))))            ;stifle the subgoal.
```

Notice that the activities working on each of the subgoals stop if the main goal is satisfied, regardless of how. We could create a third such consequent sprite and insert it in the system and these would still behave properly.

4.2.2 AND Subgoals without Variables

If we wanted to determine whether a person was a bachelor it would be sufficient to determine that he was male and unmarried. This could be accomplished by the following:

This consequent broadcasts the two and subgoals simultaneously and establishes continuations awaiting the results. When they are received, the individual subactivities are stifled. Another continuation sprite awaits the successful completion of both subactivities and broadcasts the main result.

Fig. 6. Simple And Subgoals

```
(when (GOAL (BACHELOR =x)) =activity                    ;If you want to show x is a bachelor,
      (let ((subgoal1 (new-activity))                   ;Start one subgoal activity,
            (subgoal2 (new-activity)))                  ;and another subgoal activity.
           (broadcast (GOAL (MALE x)) subgoal1)         ;Try to determine if x is male.
           (when (MALE x)                               ;If you learn x is male,
                 (broadcast (STIFLE subgoal1)))         ;stifle the subgoal activity.
           (broadcast (GOAL (UNMARRIED x)) subgoal2)    ;Try to determine if x is unmarried.
           (when (UNMARRIED x)                          ;If you learn that x is unmarried,
                 (broadcast (STIFLE subgoal2)))         ;stifle the subgoal activity.
           (when ((MALE x)                              ;If you learn that x is male,
                  (UNMARRIED x))                        ;and that x is unmarried,
                 (broadcast (BACHELOR x)))))            ;broadcast that x is a bachelor.
```

Activities form a tree by a subactivity relationship. When an activity is STIFLEd, it and all activities transitively related to it by the subactivity relation stop work. The patterns of these consequent sprites are followed by a pattern variable (i.e. `=activity`) to indicate the match will only occur if the goal assertion, the sprite's pattern, was broadcast in an activity that is currently functioning. Any new activities produced inside the body of this sprite become direct subactivities of the activity in which the assertion was broadcast. The next section contains a precise description of how activities are defined and their relationship to the other objects in the system: sprites and assertions. Note that if the main goal activity bound to `activity` is stifled at any time work on its subgoal subactivities `subgoal1` and `subgoal2` stop work.

4.3 How Activities Work

This section describes how activities function, what being in an activity means, and their relation to the other concepts of the system.

The Ether environment consists of assertions and sprites. The actual "work" of the system is done by executing the bodies of sprites that have been triggered by assertions. In order for a sprite to trigger, it must be part of some (non-stifled) activity. Associated with each currently active sprite is an activity. Associated with *some* assertions (those representing goals, for instance) must be one or more activities that will supply processing power to sprites matching the assertion, i.e. sprites capable of carrying out the tasks called for in the assertion.

4.3.1 The Rules

There are two cases that need be considered.

(a) If the sprite is of the form:

```
(when (pattern)
      body)
```

and is part of a currently active activity, the match will be processed. All new broadcasts and sprite activations that result from evaluating the sprite's body will happen in the same activity.

(b) If the sprite is of the form:

```
(when (pattern) *activity
      body)
```

is part of a currently active activity *and* the assertion has been broadcast with one or more activities that are currently active, the match will happen. All new broadcasts and sprite activations that result from evaluating the body will happen in a new activity that is a subactivity of *all* the activities the assertion was broadcast in. This property is *retroactive*. If the assertion associated with the parent activity is subsequently broadcast with a new activity, this activity is added to the list of parents.

When an activity is stifled, all work occuring in that activity is halted. If this activity has subactivities they are also stifled providing they do not have additional parent activities that are not stifled.

Chapter V Hypothetical Reasoning

Many procedural deduction systems contain facilities for creating separate subworlds of the then current collection of assertions (world model) to allow reasoning new deductions to be made that are contingent on this collection of assertions. New deductions made are placed within this subworld and thus the rest of the system is left unaltered.

The notion of a *situation* is introduced by McCarthy [13] as one way of accomplishing this. All n-tuple assertions are made into n+1-tuple assertions by the inclusion of a *situational tag*. For example the assertion `(HAS MONKEY BANANAS)` can be relativized to become `(HAS MONKEY BANANAS WORLD15)`. Then if all assertions are so relativized, the problem solver can reason about what would be true in `WORLD15` and can make new assertions that apply to that world without affecting the state of belief about other hypothetical worlds.

`QA4` introduced the notion of a *context* for similar reasons. Contexts are a generalization of Algol block structure. Contexts can be pushed and popped. When a context is popped, changes made in that context become invisible. `QA4` generalizes block structure by making it possible to coroutine between the various contexts; contexts form a tree structure. The `QA4` context mechanism is somewhat less general than situational tags because only one context can be current at a time. This makes it impossible to concurrently examine and manipulate several of them. Contexts do supply one additional structuring mechanism that situational tags do not. When a context is *pushed* the new context contains all the information contained in the previous context. This makes it easy to determine the implications of making a change to the current world model without making a separate copy of it.

Context and situational tag-like mechanisms are used to create *hypothetical worlds* inside the machine that can be reasoned about separately. There are two reasons for wanting such a mechanism. The first is to determine the *consistency* of a hypothesis with presently believed facts. The second is to determine the implications of making *changes* to the current world as would be done in robot planning problems, for example. We will call these two uses *additive* and *manipulative*. The name additive is used because the collection of assertions representing the new hypothetical world is a superset of the assertions in the old one. Manipulative mechanisms are more general. The assertions in the hypothetical world produced are a function of the assertions in the one it was derived from. Manipulative mechanisms are inherently more complex. Most problem solving systems place more emphasis on manipulative rather than additive hypothetical reasoning and in fact do not recognize the difference. We have found some new uses for additive mechanisms in parallel problem solving and will concentrate only on these in this paper. A discussion of manipulative hypothetical reasoning and their uses in planning will appear in a later paper.

5.1 Viewpoints

The term given to the Ether analog of context or situation is *viewpoint*. Viewpoints have the flexibility of situational tags and an inheritance mechanism something like contexts. All assertions representing facts about the world are considered to be in some viewpoint. The syntax for these assertions is a 2-tuple: the first element is the assertion itself and the second is the viewpoint. For example `((ON A B) H1)` means that `(ON A B)` is true in viewpoint `H1`. So far viewpoints look just like situational tags. When viewpoints are created, they may be declared to be *subviewpoints* of other viewpoints. When a viewpoint is made a subviewpoint of another (in an additive hypothesis) all assertions of the second *virtually* become assertions of the first; i.e. sprites will trigger on them as if they were actually broadcast in the subviewpoint. This concept is similar to Fahlman's [4] notion of *virtual copying.*†

The function used to create new viewpoints is called *new-viewpoint*. It is given as an optional argument the viewpoint‡ it is a subviewpoint of. It might be used in the following way:

```
...
  (let ((hypothetical (new-viewpoint (parent INITIAL)))
        (broadcast ((ON B C) hypothetical)))
...
```

This has the effect of creating a new viewpoint (which we will call "`HYPOTHETICAL`" for the sake of discussion) that is a subviewpoint of the currently existing viewpoint `INITIAL`. Suppose that `INITIAL` had three assertions in it at some point in time:

```
((ON C A) INITIAL)
((ON A B) INITIAL)
((MADE MOON ROQUEFORT) INITIAL)
```

Then immediately after the broadcast, `HYPOTHETICAL` would contain (at least) four assertions:

```
((ON C A) HYPOTHETICAL)
((ON A B) HYPOTHETICAL)
((MADE MOON ROQUEFORT) HYPOTHETICAL)
((ON B C) HYPOTHETICAL)
```

The contents of `INITIAL` is not affected at all. Note also that any additional assertions broadcast *at any future time* in `INITIAL` will immediately appear in `HYPOTHETICAL`; there are no race conditions between subviewpoint creation and broadcasting in superviewpoints. We will use a diagrammatic representation to describe

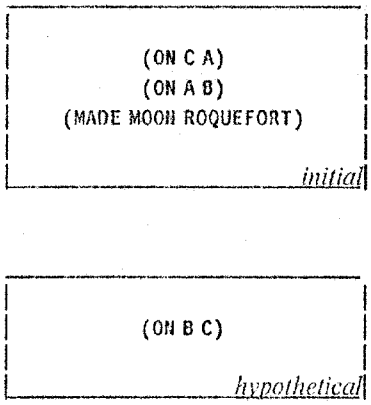
---

† The terminology here is somewhat ambiguous. We will sometimes consider **((ON A B) H1)** to be an assertion and sometimes consider it to represent the assertion **(ON A B)** in viewpoint H1. Hopefully this will not cause confusion.

‡ The viewpoint hierarchy can be more general than a tree structure. A viewpoint can be a subviewpoint of more than one other viewpoint. The subviewpoint hierarchy can form any graph without directed cycles. The assertional content of each of the parents is virtually copied into the subviewpoint. We do not have the multiple parent inheritance problem that occurs in class-structured languages. For the remainder of this chapter, though, all viewpoints will have no more than one parent.

viewpoint structures as shown in figure 7.

Fig. 7. Example of Hypothetical Viewpoint

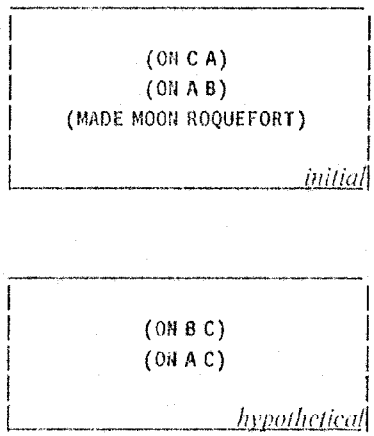


Individual viewpoints are shown as boxes of assertions with subviewpoint relations indicated by arrows. An assertion that is virtually copied from one viewpoint to another will be explicitly shown only in the viewpoint it was actually broadcast in. However, to antecedent sprites, it will appear just as if the the assertions were carried along the subviewpoint link and actually placed in the lower viewpoint. Suppose there was an active antecedent sprite such as:

```
(when {((ON =x =y) HYPOTHETICAL)                    ;If x is on y in the hypothetical viewpoint,
       ((ON y =z) HYPOTHETICAL)}                    ;and y is on z in that viewpoint,
      (broadcast ((ON x z) HYPOTHETICAL)))          ;Assert x is on z in that viewpoint.
```

Then the assertions `(ON A B)` and `(ON B C)` will be picked up by this sprite and cause `(ON A C)` to appear in `HYPOTHETICAL`. The new viewpoint structure appears in figure 8. When doing antecedent reasoning from the

Fig. 8. Hypothetical Viewpoint After Processing



assertions in a hierarchy of viewpoints, the only assertions to be actually broadcast in the subviewpoint are those that *depend* on the new assertions broadcast in the subviewpoint. This (assuming additive inheritance) is a rather trivial contribution to the study of the frame problem.

5.1.1 Antecedent Reasoning With Viewpoints

All work, including both consequent and antecedent reasoning, must occur in some activity. The emphasis throughout this work is on schemas for consequent reasoning. Code for creating activities to pursue antecedent reasoning has been, for the most part, left out of the examples. Our technique for instantiating antecedent sprites is a variation on the one used by Charniak [14]. The key idea is we have a sprite that requires an activity to trigger (just as we do with goals). The assertion this sprite triggers on indicates the viewpoint on which antecedent reasoning is to be done. This sprite creates the antecedent sprites in the new activity. In Ether code this appears as follows:

```
(when (ANTECEDENT-REASON viewpoint) =activity
      (when (antecedent1 viewpoint)
            (broadcast (consequent1 viewpoint)))
      (when (antecedent2 viewpoint)
            (broadcast (consequent2 viewpoint)))
      ...
      (when (antecedentn viewpoint)
            (broadcast (consequentn viewpoint))))
```

There may of course be many antecedent reasoning activities working on a given viewpoint. If the antecedent sprites are divided into several activities according to the semantics of the problem domain, these activities can be manipulated separately as the computation progresses.

It is the responsibility of the code that creates a viewpoint to initiate antecedent reasoning on the viewpoint.

5.2 Deduction by Antecedent Reasoning to Anomalies

One use for additive viewpoint inheritance is in doing what mathematicians call *indirect proof.* Indirect proof is a proof method in which the negation of the theorem that is desired to be proved is assumed and contradictory consequents are demonstrated. Indirect proof is used very commonly in mathematical reasoning. If you scan a typical text in Topology, say [15], it seems that more than half the theorems use indirect proof for at least part of their demonstration.

There has been some argument in recent years that indirect proof is not appropriate as a sound basis for reasoning in domains outside pure mathematics. The argument asserts that any complex reasoning mechanism must contain some mutually incompatible beliefs. If some assumption is made a contradiction will be obtainable. Thus any fact you like is derivable via indirect proof. While the basis of this assumption (the inherent inconsistency in the belief structures of a sufficiently complex reasoner) is undoubtably correct,

and strongly suggests that logic cannot be an ultimate basis for reasoning,† the plausibility of some mechanism very much like indirect proof for reasoning about *negative goals* seems still quite necessary.

The basic idea is that the reasoning mechanisms imagines the antithesis of the negative goal to be true in a separate hypothetical world that also contains facts currently known to be true. This world is then examined for anomalous conditions. If one is found, the original negative result is asserted. For example, if I told you there was an angry skunk in this room you would not believe me. How do you so quickly decide this? I propose that the reasoning goes something like: "Suppose there were an angry skunk in this room. Then there would be a horrible odor. I do not notice a horrible odor. Therefore there is no such skunk here." We have achieved a negative goal. What we have done is created a world inside our machine in which we placed all known facts plus the fact that there was an angry skunk in the room. The antecedent theorems "went to work" and quickly discovered an anomaly. This mechanism seems far more plausible than straightforward consequent reasoning. It is easier to imagine an antecedent-driven indirect proof-like mechanism for doing this than a consequent method that knows how to prove a skunk isn't in a room.

The reason this mechanism seems primarily useful for deriving negative results in "common sense reasoning" is that the technique depends on the ability to reason antecedently from the negative of the fact to be demonstrated. If the goal was to prove there *is* a skunk in the room we would have to imagine a world that contained the one additional fact of there *not* being a skunk in the room. Certainly this fact would not trigger any new facts and thus nothing can be learned; no anomalies could be found.

It is interesting to note that indirect proof, in mathematics, does not exhibit this limitation. This is because of the nature of mathematical concepts. In mathematics, if we can derive interesting facts from the proposition P, then it is also likely that P will have interesting consequents. One mathematical argument might go like "Suppose topological space T is Hausdorf. Then there is some open neighborhood U of x such that ..." or like "Suppose topological space T is not Hausdorf. Then there are two points x and y in T such that there is no open set containing y that does not contain x. ..." There does not seem to be the same asymmetry as exists with common sense reasoning.

The way we would do indirect proof-type reasoning in Ether is by creating a viewpoint that inherits from the viewpoint containing facts about the world and place in that new viewpoint the negation of the fact we are trying to deduce. In addition to doing normal antecedent reasoning on this viewpoint, special "anomaly expert" sprites are created to watch the viewpoint. In a logic theorem prover, an appropriate anomaly expert would be a sprite that checks for simultaneous existence of a fact P and a fact ¬P in the knowledge base.

---

† Hayes [16] does not discuss this objection to logic in his otherwise insightful criticism of criticisms of logic. However, in Hayes' defense, for this problem to be of important pragmatic concern would require the construction of systems much more intricate than any discussed to date in the artificial intelligence literature.

As an example of the use of indirect proof in Ether, suppose we had a viewpoint (called WORLD) with the following assertions: (⊃ Q R), (⊃ P Q), and ¬R, with a goal of ¬P. Figure 9 shows a sprite that knows how to prove negative goals via indirect proof. It does this by creating a new viewpoint and places the antithesis of the negative goal (P) in this viewpoint. Antecedent sprites working on the upper viewpoint also work on the lower one, placing all results that involve any of the assertions in the hypothetical viewpoint explicitly in it. One additional sprite is created that watches for contradictions in this hypothesis viewpoint. If they are found, the result (¬P) is broadcast in the upper viewpoint.

Fig. 9. Sprite that Initiates Indirect Reasoning

```
(when (GOAL ((¬ =x) =w)) =activity                    ;If there is a negative goal
      (let ((hypothesis (new-viewpoint w)))          ;Create a new hypothesis viewpoint
           (broadcast (x hypothesis))                ;Place the goals antithesis in this viewpoint
           (when {((¬ =y) hypothesis)                ;If a fact and its negation
                  (y hypothesis)}                    ;appear in the hypothesis viewpoint
                 (broadcast ((¬ x) w)))))            ;Broadcast the resultant theorem.
```

In order to get the ball rolling, the following would have to be executed:

```
(broadcast (GOAL ((¬ P) WORLD) ACTIVITY5)

(when ((¬ P) WORLD)
      (broadcast (STIFLE ACTIVITY5)))
```

We do not have the continuation sprite in figure 9 stifle the activity of the indirect proof activity directly. Rather, we create a separate sprite that watches for the result to be achieved and then stifles the activity. It should not matter who solved the goal, or how it was solved, for this activity to be stifled.

The GOAL assertion will be picked up by the consequent sprite shown in figure 9 and will create a viewpoint structure as shown in figure 10. Assuming we have activated an antecedent rule implementing *modus ponens*, new facts will be derived in the hypothesis viewpoint producing the viewpoint structure of figure 11. Then the sprite that was created to watch for assertions and their negations will detect R and ¬R being present in the lower viewpoint and broadcast the result to the higher viewpoint as shown in figure 12. At this point the sprite that watches for the goal (¬P) to be achieved stifles ACTIVITY5 and all work attempting this goal stops.

The purpose in our introducing techniques of indirect proof are twofold. First, it is an example of a *program in which the reasoner can concurrently reason about different world models (viewpoints) in parallel.* Any consequent directed sprites that picked up the goal of ¬P would have worked unhindered in parallel with the one attempting indirect proof. The second purpose is suggested by the skunk example above. If P does not imply any anomalies (i.e. ¬P is *not* derivable by indirect proof) then P is at least *plausible.* This is the subject of the next section.

Fig. 10. Initial Indirect Reasoning Viewpoint Structure

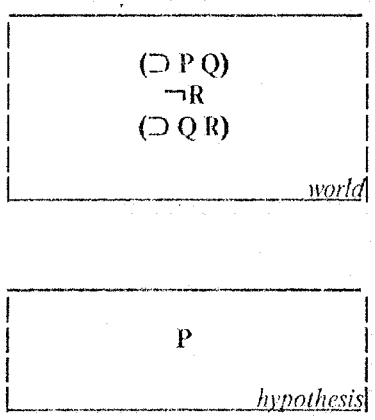
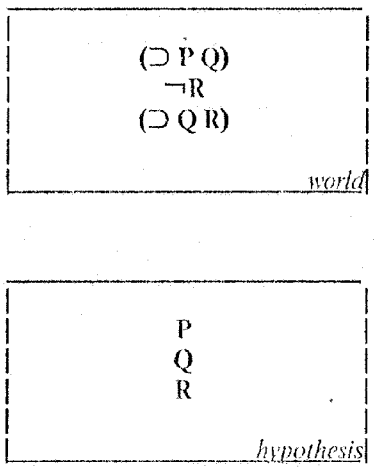


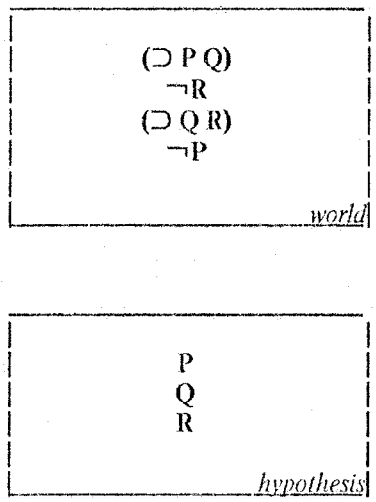
Fig. 11. Subsequent Indirect Reasoning Viewpoint Structure



5.3 Modeling Goal States and Opponents

A well known difficulty with backward chaining is that it can easily lead to exponentially widening trees of goals where many of the goals in the tree are to achieve states that are simply not true. There is a great advantage in stifling the activity working on untrue goals; every such goal is itself the root of an exponentially widening tree of (guaranteed useless!) subgoals. The strategy we will adopt in this section is to create when appropriate a model of what the world would be like *if* the goal were true and see if there are any anomalies that would indicate that the goal is unachievable. In Ether we do this by creating a viewpoint that inherits

Fig. 12. Final Indirect Reasoning Viewpoint Structure



from the viewpoint containing the world model in which the goal is broadcast. This viewpoint represents what the world would be like *if* the goal were true. We instantiate both standard antecedent sprites and anomaly detection sprites that work on this viewpoint.

In this section we will build on the example of simple OR subgoals in figure 5 on page 17 which contained two sprites that turned a goal of showing some object was a LIVING-THING into two subgoals of showing it is a PLANT or an ANIMAL. We will assume now that we have a world model containing facts about the objects in the system. In particular we may know some facts about the object we wish to prove is a LIVING-THING (call it FRED), say that it is MOBILE. This along with other facts about our world are contained in a viewpoint. We will modify the consequent sprites shown in figure 5 to create new viewpoints containing the subgoals themselves. In these subviewpoints antecedent reasoning is performed on the goal also using information contained in the world model viewpoint. In this way the consistency of the subgoal is checked. We know of only one fact so far, though the world model perhaps contains many others; that fact is that Fred is mobile. Our database contains at least the following assertion: `((MOBILE FRED) WORLD)`.

In figure 13 each of the component subgoals establishes a viewpoint that inherits from the `WORLD` viewpoint. In this viewpoint is placed the assertion of the goal. We want to do antecedent reasoning on the contents of these new viewpoints. There are presumably already antecedent sprites that are pattern matching on the `WORLD` viewpoint. We would like to extend their range of application to the newly created viewpoints. There is an Ether primitive for doing this called `SPRITE-INHERITS`. It is broadcast with two arguments, the inherited and inheriting viewpoints. It must be broadcast in a certain activity in which all the work done by these

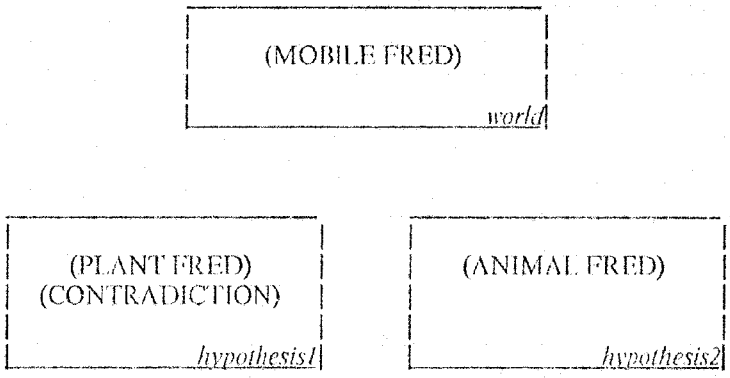
Fig. 13. OR Subgoals With Opponents

```
(when (GOAL ((LIVING-THING =x) =h)) =activity                ;If there is a goal of showing x is a living-thing
      (let ((subgoal (new-activity))                         ;create a new activity for a subgoal
            (subplat (new-viewpoint h)))                     ;and a new viewpoint for an opponent
           (broadcast (GOAL ((ANIMAL x) h)) subgoal)         ;Broadcast the new subgoal.
           (when ((ANIMAL x) h)                              ;If the subgoal has been achieved,
                 (broadcast ((LIVING-THING x) h)))           ;Broadcast x is a living thing.
           (when ((LIVING-THING x) h)                        ;If you determine x is a living thing,
                 (broadcast (STIFLE subgoal)))               ;stifle the subgoal.
           (broadcast ((ANIMAL x) subplat))                  ;Broadcast the goal to the opponent viewpoint.
           (broadcast (SPRITE-INHERITS subplat h) subgoal)   ;and start antecedent reasoning.
           (when ((CONTRADICTION) subplat)                   ;If the opponent viewpoint is contradictory,
                 (broadcast (STIFLE subgoal)))))             ;stifle the subgoal.

(when (GOAL ((LIVING-THING =x) h)) =activity                 ;If there is a goal to shown x is a living thing
      (let ((subgoal (new-activity))                         ;Create a new subgoal activity,
            (subplat (new-viewpoint h)))                     ;and an opponent viewpoint.
           (broadcast (GOAL ((PLANT x) h)) subgoal)          ;Broadcast the new subgoal.
           (when ((PLANT x) h)                               ;If the subgoal has been achieved,
                 (broadcast ((LIVING-THING x) h)))           ;Broadcast x is a living thing.
           (when ((LIVING-THING x) h)                        ;If you determine x is a living thing,
                 (broadcast (STIFLE subgoal)))               ;stifle the subgoal.
           (broadcast ((PLANT x) subplat))                   ;Broadcast the goal to the opponent viewpoint.
           (broadcast (SPRITE-INHERITS subplat h) subgoal)   ;and start antecedent reasoning.
           (when ((CONTRADICTION) subplat)                   ;If the opponent viewpoint is found contradictory,
                 (broadcast (STIFLE subgoal)))))             ;stifle the subgoal activity.
```

inherited sprites happens. If that activity becomes stifled all work done by these sprites in the inheriting viewpoint stops. We create an additional sprite watching for contradictions to be determined via antecedent reasoning. If one is found, the `subgoal` activity (which includes the antecedent reasoning on that viewpoint) is stifled. The viewpoint structure appears in figure 14.

Fig. 14. Viewpoint Structure for OR Subgoal Opponent



Antecedent reasoning will eventually determine that `MOBILE` and `PLANT` are incompatible properties and

broadcast the (CONTRADICTION) assertion.†

The general name we give to work being done to try to prove goals insoluble is *opponent activity.* The viewpoints created to look for contradictions for an opponent activity are called *opponent viewpoints.* The opponent concept is a generalization of what is usually referred to as *goal filtering.*

The most familiar example of goal filtering in the literature is the classic geometry theorem proving program of Gelernter [17]. His program used only backward chaining. A representation of the diagram was available to the program. Before it would attempt work on any goal it first checked to see if the theorem was true of the diagram. If it was not true, work on the subgoal was never begun. Otherwise the subgoal was attempted. This is analogous to our creation of a new inheriting viewpoint in which the goal is asserted and contradictions are looked for. Opponents are more general than goal filters because we do not require the opponents to always terminate in a reasonable amount of time. It would be catastrophic in a sequential system using goal filtering if even rarely the filtering procedure did not terminate. Imagine that instead of proving the *validity* of a theorem in geometry we were interested in *satisfiability.* Presence of supporting evidence in the diagram would solve the problem. Lack of supporting evidence would not be useful information. However, our opponents *would* still be useful. If a contradiction was determined then the theorem would not be satisfiable.

In the event of and subgoals we would create an opponent viewpoint that contained all the conjuncts. The example of AND subgoals in figure 6 is redone in figure 15.

Fig. 15. And Subgoals With Opponents

```
(when (GOAL ((BACHELOR =x) =h)) =activity                    ;If there is a goal of showing x is a bachelor,
      (let ((subgoal1 (new-activity))                          ;Create a subgoal activity,
            (subgoal2 (new-activity))                          ;and another subgoal activity.
            (subplat (new-viewpoint h)))                       ;Create an opponent viewpoint.
           (broadcast (GOAL ((MALE x) h)) subgoal1)            ;Broadcast a 'male' subgoal.
           (when ((MALE x) h)                                  ;If x is shown to be male,
                 (broadcast (STIFLE subgoal1)))                ;stifle that subgoal.
           (broadcast (GOAL ((UNMARRIED x) h)) subgoal2)       ;Broadcast an 'unmarried' subgoal.
           (when ((UNMARRIED x) h)                             ;If x is shown to be unmarried,
                 (broadcast (STIFLE subgoal2)))                ;stifle that subgoal.
           (when (((MALE x) h)                                 ;If x is shown to be male,
                  ((UNMARRIED x) h))                           ;and x is shown to be unmarried,
                 (broadcast ((BACHELOR x) h)))                 ;broadcast x is a bachelor.
           (broadcast ((MALE x) h))                ;Broadcast a male assertion to the opponent viewpoint.
           (broadcast ((UNMARRIED x) h))                       ;And also an unmarried assertion.
           (broadcast (SPRITE-INHERITS subplat h) subgoal1 subgoal2)   ;Antecedently reason.
           (when ((CONTRADICTION) h)                           ;If there is a contradiction,
                 (broadcast (STIFLE subgoal1))                 ;Stifle one subgoal.
                 (broadcast (STIFLE subgoal2)))))              ;Stifle the other subgoal.
```

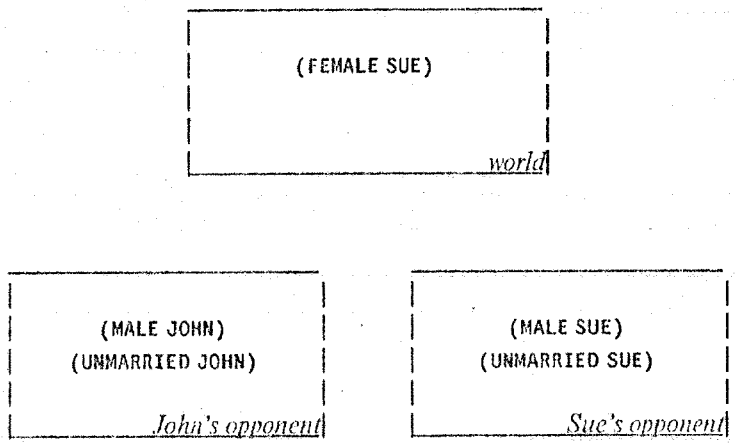
Unlike the case with OR subgoals, we require only one opponent viewpoint in which we put all the conjuncts because all must be true if the goal is to be realizable. Figure 16 shows the viewpoint structure created for a

† If it seems to you that an unreasonably large amount of antecedent reasoning must be done to support this, see section 6.3.

particular world model by the consequent sprite of figure 15 after two independent goal broadcasts:

```
(GOAL ((BACHELOR JOHN) WORLD))
(GOAL ((BACHELOR SUE) WORLD))
```

Fig. 16. AND Subgoal Opponent Viewpoint Structure



Both goals are processed concurrently with opponent activity trying to refute them. In each opponent viewpoint is a description of what the world would be like if the goal were true. One of the opponents (Sue's) rather quickly discovers a contradiction as shown in figure 17 and the activity working on `(GOAL ((BACHELOR SUE) WORLD))` is stifled.

5.4 Modeling The Goal Stack in Opponents

If there are several goals arranged hierarchically we would like the opponent viewpoints to chain together in a way that mimics the goal stack. Subgoals lower down can often be constrained by the overall purpose of the main goal. For example, if we had the goal stack of figure 18 where the `BACHELOR` goal establishes two conjunctive subgoals: `UNMARRIED` and `MALE` and the `UNMARRIED` subgoal in turn sets up a subgoal to show `FRED` does not have a `HUSBAND`. This, however, is a subgoal that is only applicable in cases where the object is a female. The subgoal should get stifled immediately. Using the methodology for opponents shown so far, this constraint would not be carried along because each opponent inherits from the viewpoints containing the world model viewpoint. Instead of having each subgoal link its opponent directly to the `WORLD` model viewpoint, we would like it to link its opponent to the opponent of the goal it is a subgoal of. For this particular example, the opponent viewpoint structure is shown in figure 19. In order to make this work, we must pass the name of the opponent viewpoint along with the goal assertion itself. The consequent sprite in

Fig. 17. Subsequent AND Subgoal Opponent Viewpoint Structure

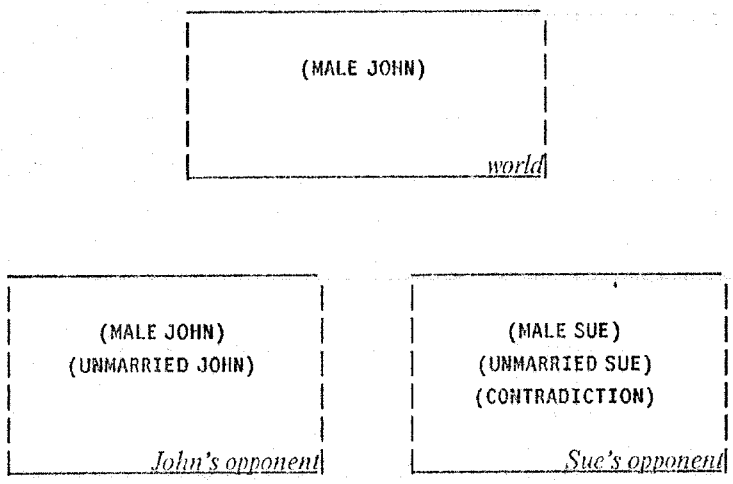


Fig. 18. Example Goal Stack

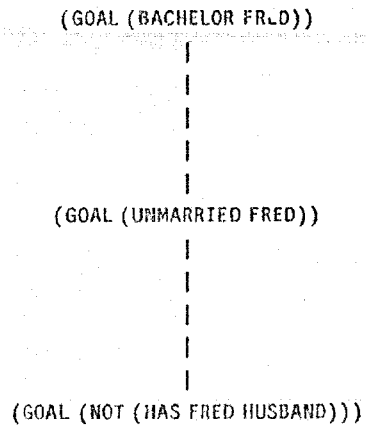


Figure 15 is redone in figure 20 with this modification. The explicit goal assertion contains an additional element: the name of the opponent viewpoint working on the goal. This allows the code in the sprite bodies to make the newly created opponent viewpoint a subviewpoint of the opponent viewpoint of its supergoal. Other than this, the code is identical.

Fig. 19. Opponent Viewpoint Structure for Goal Stack

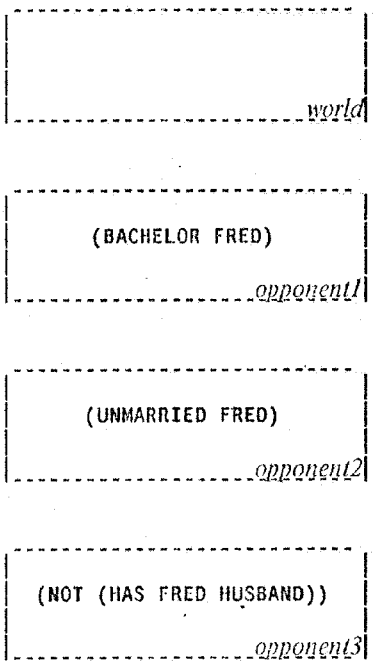


Fig. 20. Code to Create Viewpoint Goal Stack Model

```
(when (GOAL ((BACHELOR =x) =h) =opponent) =activity                ;If you want to show x is a bachelor.
      (let ((subgoal1 (new-activity))                               ;Start one subgoal activity.
            (subgoal2 (new-activity))                               ;Start another subgoal activity.
            (subplat (new-viewpoint opponent)))                     ;Create an opponent viewpoint.
           (broadcast (GOAL ((MALE x) h)) subgoal1)                 ;Broadcast a 'male' subgoal.
           (when ((MALE x) h)                                       ;If x is shown to be male,
                 (broadcast (STIFLE subgoal1)))                     ;Stifle the subgoal activity.
           (broadcast (GOAL ((UNMARRIED x) h)) subgoal2)            ;Broadcast an 'unmarried' subgoal.
           (when ((UNMARRIED x) h)                                  ;If x is shown to be unmarried,
                 (broadcast (STIFLE subgoal2)))                     ;stifle the subgoal.
           (when {((MALE x) h)                                      ;If x is shown to be male,
                  ((UNMARRIED x) h)}                                ;and x is shown to be unmarried,
                 (broadcast ((BACHELOR x) h)))                      ;Broadcast x is a bachelor.
           (broadcast ((UNMARRIED x) subplat))                      ;Broadcast 'unmarried' to the opponent.
           (broadcast ((MALE x) subplat))                           ;Broadcast 'male' to the opponent.
           (broadcast (SPRITE-INHERITS subplat opponent)            ;Start antecedent reasoning.
                      subgoal1 subgoal2)
           (when ((CONTRADICTION) subplat)                          ;If there is a contradiction,
                 (broadcast (STIFLE subgoal1))                      ;stifle one subgoal activity,
                 (broadcast (STIFLE subgoal2)))))                   ;and stifle the other subgoal activity.
```

5.5 The Relationship Between Viewpoints and Activities

In every example in this chapter we have created viewpoints and activities in parallel. Whenever there was a problem to be solved, an activity would be created to pursue the problem and a viewpoint created to serve as a "scratch pad" for the activity. This close relationship might tempt one to simplify the language somewhat by combining the two notions.

Viewpoints and activities are, however, quite distinct notions. Viewpoints are a mechanism for structuring and localizing knowledge. Activities are a way of controlling the processing that actually gets done. Less trivial programs in Ether than the examples in this paper might require the use of an activity that needed to access more than one world model (i.e. viewpoint) to accomplish its purpose. Conversely, the information in a viewpoint may be useful irrespective of the state of the activity that created it.

Chapter VI Some Further Ideas

6.1 Resource Control

We have argued that by allowing many processes to run concurrently tighter control over certain search problems can be achieved. This increase in control can come from two sources: (1) taking advantage of wide variabilities in the timings of methods, and (2) the use of opponents to prune demonstrably useless attempts. The system is able to capitalize on interactions between various running activity in ways that would be hopelessly complex to manipulate by a coroutine-like control structure. Although our system is protected from catastrophic failure when individual activities diverge, there is one sense in which we have lost control; we have no means to protect the system from getting choked with thousands upon thousands of activities, choked to the point where no activity can do anything at all!

6.1.1 The Basic Idea

The kinds of problem solving situations Ether is designed for involve substantial trees of backwardly chained goals. These problems have the character that any given approach (goal) is not likely to succeed. We have proposed one mechanism, namely opponent activity, that can achieve eventual pruning of activities working on useless subgoals. Often it will require some amount of work to be done for a goal to be pruned. We are of course most interested in pruning goals higher up in the tree. Because of the exponential growth character of expanding goal trees, there may be, in a short time after the program is started, so many running activities that none can get anything done. The system becomes choked. Even if you see this scenario as being somewhat unrealistic, it would seem that in a large problem solver some little corner of it would have this property of generating many useless activities that do not get quickly stifled. This one corner would grow in a cancer-like way and could come to dominate the entire problem solver. This is the traditional idea of a "combinatorial explosion" applied to activities instead of data. We have to provide some means of preventing it from getting out of hand.

Our solution to this problem is the introduction of the notions of *processing power* and *energy*. The machine is viewed as a finite resource usable at a constant rate in the sense that during a given interval of time the machine can do a constant amount of work. Drawing an analogy with physics, we say that a machine has a constant amount of *power* available to it that can be divided among its running activities. Each activity uses up an amount of *energy* equal to the time integral of the power available to it. When an activity creates another activity, it must give up a certain amount of its processing power to this activity. Thus processing power, in addition to being conserved globally, is conserved locally.

An analogy with tree search algorithms can be made here. Sequential programs can be said to correspond with depth-first search, and parallel programs to breadth-first search. There is a third class of tree search

algorithm known as best-first that is a generalization of depth and breadth-first search. Best-first searches can make use of available heuristic information to decide what node to examine next. If all activities are given approximately equal amounts of processing power then the control structure is similar to a breadth-first search. If only one activity (or one string of activities related by the sub-activity relation) has processing power at a time it is similar to a depth-first search. Best-first search can be emulated by using heuristic information to control the allocation of processing power.

Parallel processing with resource control is actually more general than best-first search. With best-first search, after we have picked a method, we are committed to pursuing it until we are given the opportunity to pick the next method. Parallel processing allows you to change resources allocated to running activities whenever facts are discovered that would suggest such changes; there is no concept of an atomic, indivisble action.

6.1.2 An Implementation of Resource Control

There are two kinds of resource limitations we might want to define for an activity: power and energy. A resource limitation on energy is optional; an activity with no energy limit will keep computing as long as it has something to do. All activities are power-limited, whether or not the language supplies a means of controlling it. In this section Ether primitives for dealing with these two quantities will be described. They have not been extensively used and should be considered tentative.

The machine is viewed as consisting of some constant amount of power that is divided among the running activities. For the moment we will assume the activity graph to be a tree. When an activity creates another activity it must give it a certain amount of its own processing power if this activity is to do anything. The processing power owned by an activity is distributed in some manner between its needs and those of its subactivities. The default allocation strategy is to divide power equally between an activity and each of its subactivities. With an exponentially growing tree of activities this has the property that the allocation of power falls off exponentially as the tree is traversed down from the root.

When the default scheme is not desired there must be a way to alter the power allocation assigned to a node. An activity is created to do a job. When an activity is created, its creator gives it an amount of processing power that corresponds to its notion of how important this job is to it at the time. This activity in turn divides its power in the way it sees most fit. For these reasons the primitives dealing with processing power do not deal in terms of proportions of the total machine resources that it is getting, they deal only in terms of proportions of the processing power that have been assigned to it. We think of the processing power posessed by an activity and its subactivities as summing to 1. The default scheme, then, allocates implicitly a power of $1/(n+1)$ to it and each of its subactivities. If it becomes desirable to change from the default allocation it should do:

```
(broadcast (PROCESSING-POWER activity number))
```

for subactivities it has created and for its own use:

```
(broadcast (PROCESSING-POWER-SELF activity number))
```

All subactivities that have not been explicitly allocated will divide up among themselves all the power that has not been allocated. It is an error if the sum of these numbers for an activity and its subactivities is greater than 1.

The other resource we would like to specify ways of limiting is energy. Processing energy is a quantity we are used to dealing with, sometimes expressed in the units of "CPU seconds." It is, unfortunately, implementation (and program) dependent. One would not use it in a program without having first had quite alot of experience with that program on that machine. This, unlike ideas about processing power, has received some treatment in the literature in the context of *agendas* and has been used as an integral part of at least one artificial intelligence system [18]. There are any number of things we might want to do if an activity has expended its energy limit. The following sprite stifles an activity when it has reached a prescribed energy limit:

```
(when (PROCESSING-ENERGY activity number)
      (broadcast (STIFLE activity)))
```

Other things we may want to do when an activity has reached a limit is check to see how it is doing, based on information that has been broadcast by that activity during its running. If it has been making "satisfactory progress" it should be allowed to continue, otherwise halted.

The primitives described are just that, primitives. We need to build on this a higher level language for discussing resource control that speaks in the language of problem solving rather than these low level concepts.

6.2 Quiescence

This paper discusses the desirability and possibility of doing reasoning in parallel. Emphasis has been placed on useful interaction between concurrent processing in order to limit search. In section 6.1.2 we present control structure ideas for making use of notions of variable processing power to implement depth first-like searches, characteristic of "sequential" problem solving.

One thing we can't do with the primitives presented so far in any easy way is to: "Order some methods. Try them one at a time. Only when you have exhausted *all* possible regimes for employing a given method do you go onto the next." Considering this to be the *only* natural control structure in our distant ancestor, Micro-Planner, it may seem somewhat odd that we cannot handle it! This does not really cause us concern

because in a problem solving situation it is rarely possible to be sure that *all* activity that could possibly accomplish some goal has terminated; new information may be learned that could give a *quiescent* activity, one with no work to do currently, new things to do. For most applications we believe it desirable to use the merits of what the activity has or has not done so far as a gauge on whether to allow it to continue or not. Quiescence is really a degenerate case of the much more important problem of detecting when an activity has ceased to make useful progress.

There do seem to be, however, certain kinds of problem solving situations in which it is desirable to determine whether an activity has gone quiescent. I believe one such problem is *cryptarithmetic*. A well-known example of a cryptarithmetic problem from Newell and Simon's book [19] is:

```
  D O N A L D
+ G E R A L D
-------------
  R O B E R T
```

The problem is to find an assignment of digits to letters so that this template represents a valid summation.

This kind of problem is most usefully solved by multi-stage process of *constraint propagation* and *case splitting*. Constraint propagation can be accomodated in Ether by antecedent reasoning. For example, if we learned that `(HAS-VALUE D 5)`, an antecedent sprite could assert (by examining the last column) that R must have a value greater than five and G a value less than 5. These constraints would "propagate" to other columns containing the letters R and G, and to other letters that were competing with G and R for values. In this way the problem solving space becomes constrained monotonically with time. When antecedent processing has terminated (becomes quiescent), as it must in a reasonable amount of time, either all letters will be assigned unique digits making the problem *solved*, or there will be some letters that are not yet fully constrained.

The search for a solution can continue by case splitting on the value of some digit. For example, if we know that R must be either 7 or 9, two new viewpoints can be created, inheriting from the current one, in which `(HAS-VALUE R 7)` and `(HAS-VALUE R 9)` are placed respectively. Antecedent processing continues in these viewpoints until one of three things happen: a contradiction is determined to exist in the viewpoint in which case antecedent processing activity is stifled, a solution is reached, or a quiescent state is reached. If the third possibility happens case splitting can be effected again on some other digit.

To detect quiescence, the pattern of a sprite may be the special form: `(QUIESCENT` *activity*`)`. The sprite will then trigger when the designated activity has gone quiescent. Using this the cryptarithmetic problem solver described above can be implemented in a straight forward manner.

Micro-Planner-style depth first searches can be implemented using the quiescence detection mechanism. This

is done by starting up one alternative, waiting for its activity to become quiescent, and then starting the next. This is shown in figure 21.

Fig. 21. Code for Implementing Depth First Search

```
(let ((activity1 (new-activity)))
     (broadcast (GOAL (alternative1)) activity1)
     (when (QUIESCENT activity1)
           (let ((activity2 (new-activity)))
                (broadcast (GOAL (alternative2)) activity2)
                (when (QUIESCENT activity2)
                      (let ((activity3 (new-activity)))
                           (broadcast (GOAL (alternative3)) activity3)
                           (when (QUIESCENT activity3)
                                 ... ))))))
```

6.3 Virtual Collections of Assertions

The value of pattern-directed invocation as a basis for artificial intelligence programming is in the generality different methods have for communicating with each other. The different methods communicate in a language based on the semantics of the problem domain rather than one based on the control structure of the program. It is certainly a poweful idea yet one that has met little application outside of "toy" domains. This can be attributed principally to the lack of efficiency of all extant implementations. The lack of efficiency can be traced to two sources: (1) Any assertion broadcast is potentially processable by any sprite. (2) All information flow in the program involves the creation of assertions, structures that need to be CONS'd. Discrimination nets and other technical aids ameliorate the situation somewhat, though not enough.

Compilation schemes, although attractive at first glance, do not seem very plausible in the general case. Compilation of Ether-like languages would entail converting broadcast-when interactions into function calls with arguments. If we knew that goal assertions of a certain form were only and always received by a certain set of sprites, the broadcast of this assertion could be replaced by function calls of the code associated with the sprites. However, sprites can be created while the system is running. There is no way a compiler can know from syntactic considerations when this is the case. You might imagine a scheme in which the user specifies when this more restricted condition holds. While this can certainly be done, the program writer might just as well have specified the code in terms of the function calls it would be compiled into.

It is the case that many subsections of a typical Ether-like program can be easily coded in a host language such as Lisp. We would like a scheme for such *hand-coded* methods to communicate with other hand-coded methods and with subsections of the system that are more naturally written using pattern-directed invocation. The inspiration for the method I will present comes from the object-oriented language formalisms. (Using actor terminology) an actor is described *solely* by its message passing behavior, the messages it accepts and replies with. Efficiency can be incorporated very natuarally in these systems without sacrificing program clarity. For example, a matrix is an actor that accepts two kinds of messages, one for storing new values and

one for requests as to values of its elements. Many matrices in applications are *sparse*, that is their values are zero for almost all elements. A sparse matrix is most efficiently stored as a hash table containing entries for the non-zero elements. A function that took matrices as arguments in a non-object-oriented language might have to check first to see how the matrix is represented to know how to access it. An object-oriented language allows the programmer to create a sparse matrix by specifying how it responds to the two kinds of messages mentioned. After this actor is created, it will behave functionally identically to any other matrix. The rest of the program is effectively shielded from the intricacies of how the individual kinds of matrices are represented and accessed.

A subset of a pattern-directed invocation system is exactly described by (1) a description of the assertions it is interested in responding to, and (2) a description of the assertions that get added to the database when one of the assertions that it is interested in is added to the database. Any method embodied in code that can provide these two descriptions can be interfaced to the rest of an Ether-like language *completely transparently.* This does not mean the assertions it would add to the database are actually present; only that the method supplies code for deciding if they are virtually present. A method described this way is known as a *virtual collection of assertions.*

Incorporating property (1) of a virtual collection of assertions, indicating what assertions it is interested in and what it does when they are broadcast requires no additions to the Ether language; sprites (at least in the current implementation, whose bodies can contain arbitrary Lisp code) already do exactly this. The only new facility we must supply is one to handle the queries about virtual presence in the database. For this we must specify a set of procedures that specify (1) the membership questions they are capable of answering, and (2) how to decide for an individual assertion query whether the assertion is actually present. A possible syntax for this is:

```
(when-asked (pattern viewpoint)
            --arbitrary Lisp code-- )
```

The *pattern* specifies the class of assertions this procedure can handle. The *arbitrary Lisp code* returns a list of all assertions it considers to virtually exist matching the pattern in the given viewpoint.

For example, suppose we had a virtual collection of assertions that modeled a semantic net. It should be capable, among other things, of deciding that an object known to be human is also a mammal. An entry point to the semantic net would look something like:

```
(when (HUMAN =x)
      --code to enter x into the semantic net--)
```

Then for each of the characteristics that the net will answer questions about, we have a `when-asked` as follows:

```
(when-asked (MAMMAL =x)
            --code to check if x is a mammal in the net--)
```

Then a sprite of the form:

```
(when (MAMMAL FRED)
      do something)
```

will trigger if FRED was previously known to be a human. From the point of view of the Ether program there is a very large collection of assertions available for pattern matching. However, the information is represented in the most compact and efficient way the programmer could devise.

There are many attributes a virtual collection must have to function correctly. These include ensuring proper interaction with the viewpoint mechanism and invariance of behavior with respect to the order of virtual insertion into the database and request for presence by sprites. I think this approach will make possible a synthesis of the very general but inefficient Ether mechanisms with efficient Lisp code where it is known how to construct this code. The MIT Lisp machine with stack groups and alarmclock interrupts supplies an implementation vehicle for Ether that will allow the running of Lisp code without removing the parallel behaviour of Ether programs.

Chapter VII Comparison With Other Work

7.1 Pattern-Directed Invocation Languages

Many of the concepts of pattern-directed invocation languages originate with Planner [20]. A subset of Planner known as Micro-Planner [21] was implemented. It embodied the ideas of antecedent and consequent *theorems* or procedures that were invoked automatically by the system. Micro-Planner investigated all goals by simple depth-first search with backtracking when failure points were reached.

There were a couple of interesting bugs discovered in Micro-Planner. It was not possible to distinguish between *wanting to know* if a certain fact is known to be true and *investing effort in trying to prove* it. In Ether we would say:

```
(when (INTERESTING FACT)
      (Do something))
```

if we wanted to do something only if the fact was true. If we wanted to start some work attempting to show it is true we would do:

```
(broadcast (GOAL (INTERESTING FACT)) ACTIVITY)
```

In Micro-Planner the two were lumped under the primitive `THGOAL`. Another, similar, problem with Micro-Planner was an ambiguity between *knowing something is not the case* and *not knowing if something is the case.* [11] Negation was handled via the primitive `THNOT`. `THNOT` succeeded if and only if its argument failed. For example:

```
(THNOT (THGOAL (HAS ALPHA-CENTAURI LIFE)))
```

would, in the absence of any way to prove the goal, succeed. This is the Micro-Planner equivalent of proving `(NOT (HAS ALPHA-CENTAURI LIFE))`. Languages developed subsequently, like `QA4` [22], Conniver [23], and Popler [24] made further contributions. `QA4` introduced the notion of contexts as a means of structuring the Lisp environment (property lists, variable bindings, etc.). Contexts could form a tree structure; it was possible to create a context, leave it for a while, and go back to it later on. `QA4` was found to be very inefficient and a subset of it known as `QLISP` was embedded more directly in Lisp [25]. Conniver had a context mechanism similar to `QA4`'s. Its principle contribution was a way of controlling backtracking by means of *generators* and *possibility lists.* Instead of the language implementation trying possibilities in some arbitrary order, the program could manipulate possibilities to try next as data structures, hopefully optimizing its search. Programs in Conniver become complex beyond the point of understandability. These systems largely failed because of unexpected interactions between methods that were *conceptually* unordered (i.e. running in parallel). Popler [24] was an implementation of many of the original Planner ideas in the language Pop-2. It

was the first of these languages to allow methods to be run in parallel, though it was not designed for the massive parallelism Ether is.

AMORD made use of *explicit goal assertions* (see section 3.3). The sole means of structure in the language (above the basic notions of sprites and assertions) is a backbone of justifications that are manipulated by a truth maintenance system [26]. Justifications are used to maintain the consistency of the database as well as the goal structure. While allowing some powerful new control strucutres via the justifications, AMORD has given up certain facilities that are found in other procedural deduction systems such as a viewpoint mechanism. It is the philosophy of their approach that higher level concepts such as viewpoints and activities can be simulated with justifications and truth maintenance. The view taken by the present work is that these ideas are best "unbundled" to make the semantics of what you are doing clearer. By creating the notion of an *activity* we can deal with various different approaches to solving a problem as objects that can be manipulated (stifled, speeded up, etc.). A corollary of the use of a truth maintenance system is that all visible assertions must be consistent with one another. This makes it impossible to reason about several conflicting world models concurrently.

7.2 Parallel AI Systems

The Hearsay speech-understanding system [27] makes use of decentralized parallel processing in a fundamental way. It presents many levels of description (raw input, phonological, word, phrase, etc.) that are constructed in parallel with one another. The basic philosophy of the approach is that each level is inherently noisy and incomplete, and thus the only way anything can get done is if processing at one level helps to constrain work at other levels. In this sense their approach is quite similar to ours. Theirs is a more special purpose system; Hearsay is not a programming language in which such concepts as opponents could be written.

Lenat [18] presents the most interesting use of notions of resource control that I have seen. His domain is mathematical discovery; the object is to have the program discover new mathematical concepts from ones it already knows about. Many possible avenues of discovery are explored in parallel. There is a criterion for interestingness of the potential discovery that guides the scheduler in determining what to run next, and for how long. Lenat's basic control structure is an agenda mechanism with resource limitation information based on how interesting the result would be if achieved. The important point of agreement between his work and ours is the observation that you cannot tell how successful a path will be short of trying it; for this reason many paths should be pursued in parallel to avoid having "all your eggs in one basket." Lenat's thesis inspired the concepts of processing power and energy in Ether.

Fahlman [4] discusses a special purpose language and hardware network for doing the kinds of problems

appropriate for semantic nets. He shows that many problems can be solved by doing set intersections that can be easily simulated by passing tokens through the network. He argues that conventional sequential control structures cannot do the search that seems to be required to solve these problems in real time as clearly happens with people. His system is designed to be connected to a problem solving system as a semantic net subroutine box. Fahlman's approach to combining parallelism with artificial intelligence, in contrast to ours, is to make brute force searches tractable. We have demonstrated that parallelism can be used to make searches more controlled.

Smith [28] introduces a mechanism known as the *contract net.* The problem solver itself is distributed around a resource-limited network. The nodes of the network interact with each other in a manner reminiscent of commercial systems consisting of contractors, contracts, bids, and awards. The bidding protocols result in a distribution of tasks throughout the system in a manner that utilizes the available processing resources reasonably. Contract nets, in contrast with Ether, deal with issues of task distribution on physical hardware. The reasons bids are awarded include such items as load balancing, better suitability of the processor, etc. We have concerned ourselves only with parallel language design and the uses of parallel processing for artificial intelligence. A protocol such as contract nets may well be necessary to implement Ether on parallel hardware.

Minsky and Papert [29] are developing a theory of intelligence they call the "society theory of the mind". The theory asserts the existence of an enormous number of *agents* or specialists in certain areas or points of view. Intelligence is manifest through the interaction of these agents in a massive parallel scheme. The emphasis in their scheme seems to be the presence of lots of simple computational elements all "speaking their mind"; the final behavior of the system seems to represent a compromise between the various agents. They develop the notion of a *critic* which bears some resemblance to our opponents. The society theory as it now stands is metaphorical and suggestive of how a computer system might be implemented to exhibit intelligent behavior rather than a specific technical proposal.

7.3 Languages for Parallel Processing

There is now a large literature on languages for parallel processing. There are several distinct reasons why parallel processing languages and systems have been proposed. We will list four of these and then suggest a fifth proposed by the present work.

1. *To make computers useful in an inherently parallel society.* We are used to, in our own lives, interacting with such diverse and distant information sources as banks, schools, governments, etc. If we want to integrate computers into this society, they in turn must be able to deal with these diverse sources. The sequential machine model is not applicable here. As it was realized computer systems needed these capabilities, schemes for interrupt handling were developed. These schemes naturally led into a consideration of parallel

processing languages.

2. *To provide robust computation.* Computing machines, being inherently complex, are prone to errorful performance. With current hardware trends as they are, a practical solution to this problem is to compute redundantly making use of several processors. Discrepancies and hardware signaled errors will cause some backup or reconfiguration operation to happen. This approach has been successful in such critical applications as the design of jet airplane flight control computers [30], onboard space vehicle computers [31], and remote message relay processors [32].

3. *To increase overall program speed.* The idea here is to exploit cheaply available multi-processor architectures by making it convenient to separate certain tasks to be performed in parallel. Friedman and Wise [33] note that "applicative" languages, such as pure Lisp, can have function execution transparently done in parallel. They advocate a scheme in which one processor is given charge of the evaluation; as it runs across subtasks to be handled they are farmed out to other processors. Baker [34] develops the notion of a *future*. Futures give the program writer explicit control over what activities are farmed out.

4. *To increase program understandability.* As more familiarity is gained with concepts of parallel programming several researchers have discovered that certain tasks are more easily described as parallel algorithms. These points are stressed by Hoare [35] and Kahn and MacQueen [36].

Ether has been used to explore what is perhaps a fifth use of parallel processing: *combinatorial implosion.* Useful interaction between running processes can occur that simplify the overall computational effort. These ideas have applicability in artificial intelligence. Possible application to other areas is suggested by chapter 2 although I know of no other clearly useful algorithm than the one in that chapter.

7.3.1 Synchronization and Communication

The principle means of communication between processes discussed in the literature is by *shared data structures* modified by programs embodying *critical regions.* The first development in this area was the *semaphore* by Dijkstra [37]. Improvements on the semaphore led to the *monitor* by Hoare [38] and subsequently the *serializer* by Atkinson and Hewitt [39] improved upon in [40]. Other schemes for communication between concurrent processes require information to flow along predesignated topologies. The Communicating Sequential Processes of Hoare [35] is of this kind. The applicative schemes mentioned above allow information to flow along the dynamically created paths of expression evaluation only.

Ether presents a model of parallel computation that allows information flow in arbitrary ways *without* having shared data structures manipulated inside critical regions. It is instead based on the notion of *broadcasting* information that interested parties have the option of intercepting. We have argued already why arbitrary

information traffic between different activities is desirable.

Certainly, at an implementational level, Ether must support interprocess synchronization. Ether is an alternative language level formulation that when usable is superior in its ability to suppress unwanted detail. There are, of course, many problems that require synchronization (such as the "airline reservation system") and, as such, cannot be handled by the existing Ether system. We have made plausible that useful communication between parallel processes can be done without synchronization.

Chapter VIII Bibliography

[1] Kornfeld, William, *Using Parallel Processing for Problem Solving*, S.M. thesis MIT, 1979.

[2] Kornfeld, William, Hewitt, Carl, *The Scientific Community Metaphor*, work in preparation, 1980.

[3] Moravec, Hans P., *Intelligent Machines: How to get there from here and What to do afterwards*, Computer Science Dept., Stanford University, September 3, 1977.

[4] Fahlman, Scott E., *A System for Representing and Using Real-World Knowledge*, MIT PhD, Dept. of Electrical Engineering and Computer Science, 1977.

[5] Imai, Masaharu, Yoshida, Yuuji, Fukumura, Teruo, *A Parallel Searching Scheme for Multiprocessor Systems and its Application to Combinatorial Problems*, Sixth International Joint Conference on Artificial Intelligence, August 1979.

[6] Ruth, Gregory R., *Automatic Design of Data Processing Systems*, MIT Laboratory for Computer Science TM-70, February 1976.

[7] Hayes, P. J., *Computation and Deduction*, Proceedings MFCS Symposium, Czech Academy of Sciences, 1973.

[8] Hewitt, Carl, *How to Use What You Know*, Fourth Internation Joint Conference on Artificial Intelligence, 1975.

[9] de Kleer, J., Doyle, J., Steele, G., Sussman, G., *Explicit Control of Reasoning*, MIT AI memo 427, June 1977.

[10] Kalish, Montague, *Logic, Techniques of Formal Reasoning*, Harcourt Brace, 1964.

[11] Moore, Robert C., *Reasoning from Incomplete Knowledge in a Procedural Deduction System*, MIT Artificial Intelligence Laboratory TR 347, December 1975.

[12] Nilsson, Nils J., *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

[13] McCarthy, J., *Programs With Common Sense*, in Minsky, ed. *Semantic Information Processing*, M.I.T. Press, 1968.

[14] Charniak, Eugene, *Toward a Model of Children's Story Comprehension*, MIT Artificial Intelligence Laboratory TR-266, 1972.

[15] Munkres, James R., *Topology*, Prentice Hall, 1975.

[16] Hayes, P. J., *In Defence of Logic*, Fifth International Joint Conference on Artificial Intelligence, 1977.

[17] Gelernter, H., *Realization of a Geometry-Theorem Machine*, in Feigenbaum and Feldman, eds., *Computers and Thought*, 1963.

[18] Lenat, D., *An AI Approach to Discovery in Mathematics as Heuristic Search*, Stanford AI Lab Memo

AIM-286, 1976.

[19] Newell, Alan, Simon, Herbert A., *Human Problem Solving*, Prentice Hall, 1972.

[20] Hewitt, Carl, *Description and Theoretical Analysis (using schemata) of PLANNER*, Artificial Intelligence Lanoratory TR-256, April 1972.

[21] Sussman, G., Winograd, T., Charniak, E., *Micro-Planner Reference Manual*, MIT Artificial Intelligence Laboratory memo 203, 1970.

[22] Rulifson, John F., Derksen, Jan A., Waldinger, Richard J., *QA4: A Procedural Calculus for Intuitive Reasoning*, Stanford Research Institute Artificial Intelligence Center Technical Note 73.

[23] McDermott, Drew, Sussman, Gerald, *The CONNIVER Reference Manual*, Artificial Intelligence Laboratory memo 259a, January 1974.

[24] Davies, Julian, *POPLER 1.5 Reference Manual*, School of Artificial Intelligence, University of Edinburgh, TPU Report no. 1, May 1973.

[25] Wilber, Michael B., *A QLISP Reference Manual*, Stanford Research Institute Artificial Intelligence Center Technical Note 118.

[26] Doyle, Jon, *Truth Maintenance Systems for Problem Solving*, MIT Artificial Intelligence Laboratory TR-419, January 1978.

[27] Lesser, Victor R., Erman, Lee D., *A Retrospective View of the Hearsay II Architecture*, Fifth International Joint Conference on Artificial Intelligence, 1977.

[28] Smith, R. G., Davis, R., *Distributed Problem Solving: The Contract Net Approach*, Proceedings of the Second National Conf. of the Canadian Society for Computational Studies of Intelligence, July 1978.

[29] Minsky, Marvin, Papert, Seymour, *Society of the Mind*, work in progress, 1979.

[30] Wensley, John H., Green, Milton W., Levitt, Karl N., Shostak, Robert E., *The Design, Analysis, and Verification of the SIFT Fault Tolerant System*, Proceedings of the International Conference on Software Engineering, San Francisco, October 1976.

[31] Hopkins, Albert L. Jr., *A Fault-Tolerant Information Processing Concept for Space Vehicles*, IEEE Transactions on Computers, 1971.

[32] Ornstein, S. M., *Pluribus - A Reliable Multiprocessor*, Proceedings of the National Computer Conference, 1975.

[33] Friedman, Daniel P., Wise, David S., *The Impact of Applicative Programming on Multiprocessing*, Proceedings 1976 International Conference on Parallel Processing IEEE Cat. No. 76CH1127-0C.

[34] Baker, Henry G. Jr., *Actor Systems for Real-Time Computation*, MIT Laboratory for Computer Science, 1973.

[35] Hoare, C. A. R., *Communicating Sequential Processes*, Communications of the ACM, August 1978.

[36] Kahn, Gilles, MacQueen, David B., *Coroutines and Newtorks of Parallel Processes*, IFIP Congress Proceedings, 1977.

[37] Dijkstra, E. W., *Cooperating Sequential Processes*, In *Programming Languages* (Ed. F. Genuys), Academic Press, New York, 1968.

[38] Hoare, C. A. R., *Monitors: An Operation System Structuring Concept*, Communications of the ACM, October 1974.

[39] Atkinson, Russell, Hewitt, Carl, *Specification and Proof Techniques for Serializers*, MIT Artificial Intelligence Laboratory Memo 438, August 1977.

[40] Hewitt, Carl, Attardi, Giuseppe, *ACT1: A Language for Distributed Systems*, work in progress, 1980.