

# ETHER -- A Parallel Problem Solving System

William A. Kornfeld  
MIT Artificial Intelligence Laboratory  
545 Technology Square  
Cambridge, Mass. 02139  
arpanet: BAK@MIT-AI

ETHER is a new pattern directed invocation language for problem solving applications. ETHER programs allow arbitrarily much processing to happen in parallel. For example, when a goal generates several subgoals they are all pursued in parallel. ETHER introduces the concepts of an *activity* which is a generalization of *process*, and *platform*, a generalization of *context*. Two uses for parallelism are discussed: parallel evaluation of subgoals and the use of *opponents*, concurrently running activities that attempt to show a subgoal is unattainable. If the opponent succeeds then work on the subgoal halts.

## 1. Introduction

ETHER is an attempt to form a synthesis of ideas about parallel programming with concepts of pattern-directed invocation. There have been several researchers proposing parallel AI systems [1, 2] as well as systems that make use of agendas and similar mechanisms [3, 4] to get parallel-like behavior. This paper is concerned with the motivation and basic concepts of parallel pattern-directed invocation using ETHER. Due to space limitations it is sketchy in many areas; a much more complete discussion can be found in [5].

## 2. Phylogeny of Planner-like Languages

In the early 1970's there was great interest in developing languages for Artificial Intelligence. This began with PLANNER [6] which was implemented as MICRO-PLANNER [7]. Experience with the ideas of PLANNER and earlier theorem proving systems led to the development of several more sophisticated systems at different centers: CONNIVER [8] at MIT, QRA [9] at SRI, and POPLER [10] at Edinburgh. The general contribution these languages have made to programming are greater flexibility in representing and accessing data and the ability to gain access to procedures, not by their name, but by an indication of what they can do (so called *pattern-directed invocation*) [6]. If there was a problem to be solved, the routine charged with solving it could instantiate a goal.

(GOAL ("My problem is ..."))

The implementation would then match the statement of the problem against statements of what each procedure could do, and procedures that are relevant would be invoked.

The original focus of PLANNER was the writing of programs that manifested intelligence by the combined effect of loosely coupled procedures. In MICRO-PLANNER it was possible to create independent units that said "If you want to accomplish the following result, try the following:..." The interpreter was charged with the task of combining this knowledge. It treated the theorems of MICRO-PLANNER as any standard language interpreter would treat subroutines. However, because nothing specified the order theorems were tried, MICRO-PLANNER programs followed a wasteful, undirected depth-first search, using chronological backtracking.

CONNIVER tried to amend this situation by putting the control structure in the hands of the user [8]. Programs had control over the generation and selection of problem solving strategies. Unfortunately, these new abilities destroyed much of the expressive ability inherent in the original MICRO-PLANNER modularity [11].

Our view is that the failure of MICRO-PLANNER programs to control their search was because *program control had to be located at one point in the program at any given time*. This assumption is implicit in the so-called Von Neumann machine model that underlies virtually all current hardware and software systems but has come under attack recently by several researchers. We introduce the notion of an *activity* and show how it can be used to control reasoning in a highly parallel system. The context mechanism introduced in [9] is generalized to *platforms* all of which may be concurrently manipulated.

## 3. Pidgin-ETHER

This paper describes a language called ETHER that allows the user to construct problem solving systems embodying the philosophy discussed above. The analog of the PLANNER theorem is called in ETHER a "sprite". Sprites have two principle parts, a pattern and a body. Once created, a sprite watches for an assertion to appear that matches its pattern, and, if this occurs, its body is executed. The body may contain commands to create new sprites or commands to *broadcast* new assertions that other sprites may trigger on. This section develops a subset of the ETHER language known as pidgin-ETHER. The ideas in pidgin-ETHER are quite similar to those in [12]. To illustrate the difference between ETHER and MICRO-PLANNER, consider the following PLANNER-like consequent theorem:

```
(to-prove (BACHELOR ?X))
  (GOAL (UNMARRIED ?X))
  (GOAL (MALE ?X)))
```

This says "If you want to determine if X is a bachelor try to show that he is both male and unmarried." A MICRO-PLANNER interpreter would first try all possible ways of showing that X is unmarried, and then if it achieves success, all possible ways of showing X is male. The analog to this in pidgin-ETHER might be written:

```
(when (GOAL (BACHELOR =x))
  (broadcast (GOAL (UNMARRIED x)))
  (broadcast (GOAL (MALE x)))
  (when& ((UNMARRIED x)
    (MALE x))
    (broadcast (BACHELOR x))))
```

This example makes use of *explicit goal assertions* [11, 12] instead of primitive consequent rules. When this sprite receives a matching assertion (such as "(GOAL FRED)") simultaneously three things occur: a goal assertion is broadcast that causes work to begin on determining whether x is unmarried, a goal assertion is broadcast that causes the system to begin work on determining

whether  $x$  is male, and a new sprite is created that waits for these assertions to appear in the database. When (and if) they finally appear the BACHELOR assertion is broadcast satisfying the goal.

In the ETHER example both goals are worked on *simultaneously*. Each of the GOAL assertions (requests to work on the goals) may each be picked up by several sprites serving as consequent theorems and in turn may themselves cause several subgoals to be pursued concurrently.

#### 4. Activities and Stifle

This example has one problem. When a goal has been solved, computation attempting to achieve it will not stop. For example, even if FRED has been shown to be a BACHELOR, consequent sprites working on the problem of determining if FRED is a BACHELOR will continue doing so. It is desirable to be able to stop this additional processing so that system resources can be reclaimed.

To allow this kind of behavior we will introduce the concepts of *activity* and *stifle* to the pidgin-ETHER so far described. An activity can be thought of as a *locus of control with some specific purpose*. When the work being carried out by an activity is found to be no longer necessary the activity is *stifled*. This causes all computation involving sprites that are part of this activity to cease. An activity may be created to achieve some goal. If the goal has been achieved then the program should stifle this activity.

To illustrate the use of activities is redone the above example using them. Activities are initiated by giving broadcast a second argument of the name of the new activity. If a sprite receives this broadcast, newly created sprites are placed in the new activity. Suppose we wish to work on the goal of (BACHELOR FRED). The following broadcast is done with GOAL33 being the name of the activity created to work on this goal.

```
(broadcast (GOAL (BACHELOR FRED)) GOAL33)
```

There may be a number of different ways to accomplish a goal (each by a different consequent sprite). One way, as was done above, is to demonstrate FRED is both MALE and UNMARRIED.

Because this is one of perhaps many ways to accomplish a goal, its investigation should occur in a distinct activity.

```
(when (GOAL (BACHELOR x))
  (let (subgoal (new-activity))
    (broadcast (GOAL (UNMARRIED x)) subgoal)
    (broadcast (GOAL (MALE x)) subgoal)
    (when& ((UNMARRIED x)
            (MALE x))
      (broadcast (BACHELOR x)))
    (when (BACHELOR x)
      (broadcast (STIFLE subgoal))))))
```

The code given is like the previous example with some additions to show the use of activities. The function *new-activity* creates a new activity. The name of this activity is returned and becomes the value of *subgoal*. All work begun by this sprite occurs in this activity. The last sprite occurring in its body watches for (BACHELOR FRED) to be broadcast, indicating successful completion. If this is ever broadcast, the sprite broadcasts a special assertion instructing the system to stifle the subgoal activity and any activities it might have created. A stifled activity ceases to work on the goal. It does not matter how the goal was finally achieved for the stifling to occur, only that it was achieved.

If the same goal assertion is broadcast twice with different activity markers then only one new activity rather than two will be created. This new activity will be a subactivity of both parents. Thus, if there are two different goals that create a common subgoal, the work on this subgoal will not be duplicated.

#### 5. Platforms and Opponents

Planner-like languages frequently need to create little worlds or *contexts* inside the machine in which to reason. Contexts are usually defined so that only one is visible to the problem solver at a time. A parallel problem solving system must not contain such a restriction because different activities may be set up to concurrently reason about several incompatible world models. We have generalized the context concept to the notion of a

*platform*. All platforms are accessible to all sprites at all times. We [5] have found several uses for many concurrently accessible platforms.

Gelernter, in his classic geometry theorem proving program [13], introduced the concept of a *goal filter*. His program worked by using strict backward chaining from the theorem. Before attempting to work on a subtheorem it would check a diagrammatic representation of the subtheorem to see if the subtheorem was consistent. If it was not, the goal was immediately pruned. The concept of checking a subgoal for plausibility is an important one in maintaining coherency in the problem solving process. In Gelernter's program the test was easy to make and would be guaranteed to terminate with an answer in a short span of time. In many problem solving situations goal filters can make use of the full capabilities of the problem solver and may not be guaranteed to terminate. The use of goal filters still may be justified because of the desirability of cutting off exponentially growing trees of useless backward chaining.

One common mode of reasoning that people use to filter goals is to *assume* the object of a goal is true and then seeing if there is an anomaly. For example in a geometry theorem proving program we may have a goal of demonstrating satisfiability of the statement "Line AB is longer than line AC" in the following:

```
(line A B)
(line B C)
(line A C)
(right-angle A B C)
```

We create an activity and broadcast in this activity:

```
(goal (greater (line A B) (line A C)))
```

As in the previous examples this will cause consequent processing to happen on this platform attempting to prove this goal. Sprites doing consequent processing will not realize that the goal is unattainable though this is easily apparent to humans. Consequent reasoning sprites will implement backward chaining from this unattainable subgoal.

A method people use for seeing this is by determining the obvious consequents of the proposition that line AB is longer than line AC and seeing that an anomaly arises.

In ETHER, data assertions are grouped in specific *platforms*. We create a new platform that inherits from the platform containing the description of the figure and in it we broadcast:

```
(greater (line A B) (line A C))
```

Many antecedent sprites are created to allow the system to determine the implications of this assumption. We also instantiate for this platform sprites that can detect contradictions. In this particular example, the assumption has one easily seen consequent, that angle ACB is greater than angle ABC. This implies angle ACB > 90 degrees, and that triangle ABC has internal angles that sum to more than 180 degrees, an anomaly. A sprite watches for the occurrence of anomalies and then stifles the activity attempting to do consequent reasoning on this goal.

We concurrently do consequent processing attempting to prove the goal and antecedent processing to determine if the goal is not realizable. The activity attempting to achieve the goal is known as a *proponent*. The activity in which antecedent processing is being done on the goal to determine if anomalies exist is an *opponent*. If the goal is achieved, the result is broadcast and both the opponent and proponent activities are stifled. If the opponent succeeds in finding an anomaly, both activities are stifled.

The inherent parallelism of ETHER is necessary to allow use of opponent platforms. In a sequential problem solving system, the problem solver must *choose* to do one or the other first. It cannot allow opponent processing that is not guaranteed to terminate in a reasonable amount of time. Goal filters can be implemented in our scheme by creating a platform that executes the task of the goal filter, stifling the goal proponent if the filter is successful.

The full power of the problem solver can be brought to bear on the task of filtering goals.

The inheritance mechanism is such that assertions appearing in the parent of a hypothesis platform *virtually* appear in the hypothesis platform itself. The program may inherit antecedent sprites from a parent to a child platform. When this is done, the only new assertions that get generated and placed in the hypothesis platform are only those that depend on the assertions broadcast in this hypothesis platform. Thus the creation of an opponent platform is relatively inexpensive.

## 6. Further Ideas

The ETHER language is much more extensive than the kernel described here. We briefly summarize some of the other features that are discussed in [5].

**As a parallel language.** The semantics of the language have been carefully designed so that it is implementable on highly parallel hardware without introducing bottlenecks or the possibility of deadlock caused by the use of synchronization primitives. We believe that many of the choices made in this regard are also the right choices from the point of view of language design for modular programming; the language makes it easy to design programs devoid of the timing errors that plague most parallel systems.

**Processing power.** The concept of an *agenda* has been proposed for incorporating *resource-limited computation* in AI systems. We feel this is too low level a concept because the individual parts of the system must worry about their own scheduling. Imagine how much harder it would be to write programs for a time-sharing system if the individual programs had to schedule each time quantum for themselves. In ETHER the individual activities are truly parallel. To incorporate resource-limited computation we introduce the notion of *processing power*. If some activities seem to be more likely to achieve their purposes than others they are given a larger share of processing power. The user is given a set of primitives in which to build in heuristic knowledge to improve the performance of the program.

**Objects and coreference.** Most PLANNER-like languages allow variables to appear in goals. ETHER contains a generalization of the notion of an *anonymous object*. The program can place assertions on a platform causing the object to be coreferential with another object on that platform. We show that variables can be replaced with these objects resulting in solutions to many problems that are much more tightly controlled than a solution using variables. Objects also have important applications in planning.

**Manipulative inheritance.** The inheritance mechanism discussed for platforms above will inherit all assertions from a parent to a child. While this is valuable for the creation of opponents it is not generally useful in planning. Planning situations usually require the ability to make certain assertions invisible through inheritance links. For example, if a block A is taken off of a block B in a hypothetical situation we would like the assertion (ON A B) to be invisible in the hypothesis' platform even though it is present in the higher platform. We employ the use of *inheritance filters* and *justifications* in our solution to the frame problem. The user may tag assertions with markers and allow the system to automatically compute dependency information for newly created assertions. A platform inheritance link can specify a filter that indicates the class of assertions that can be "seen" through the inheritance link.

**Virtual Collections of Assertions.** A common complaint of pattern-directed invocation systems is that they are too inefficient. We would like to supply a facility that allows the user to interface efficient Lisp code to the ETHER system in a transparent manner, i.e. from the point of view of the ETHER program this efficient subprogram looks as if it is implemented in sprites. For each virtual collection the program must specify two items: the assertions that it would like to know about if they are asserted in the database, and the kinds of assertions for which it can decide virtual presence in the database. We envision a facility like this

being used to tie together many relatively large, special purpose, programs so that they can interact in a useful fashion with one another.

## 7. Conclusions

ETHER is a language that allows arbitrarily much processing to occur in parallel. We believe this feature allows programs to maintain modularity and independence of individual methods while still allowing programs to carefully control their search. We have developed two kinds of generalized control structures. In ETHER it is possible to run many methods in parallel and arrange that computation that becomes unnecessary to be stifled. ETHER makes possible a generalization of the concept of a goal filter in which the filtering computation runs in parallel with the attempted attainment of the goal. ETHER code is clear and simple and counters the commonly held belief that parallel programs are hard to design and understand.

ETHER has been implemented by the author as part of his masters thesis research. I am currently improving the implementation and modifying it for use on the MIT Lisp Machine.

## Bibliography

- [1] Lesser, V. R., Erman, L. D., *A Retrospective View of the Hearsay-II Architecture*, IJCAI77.
- [2] 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.
- [3] Lenat, D., *AM: An AI Approach to Discovery in Mathematics as Heuristic Search*, Stanford AI Lab Memo AIM-286, 1976.
- [4] Bobrow, D. G., Winograd, T., *An Overview of KRL, A Knowledge Representation Language*, XEROX PARC publication CSL-76-4, 1976.
- [5] Kornfeld, William, *Using Parallel Processing for Problem Solving*, S.M. thesis MIT, May 1979, forthcoming technical report.
- [6] Hewitt, C., *Description and Theoretical Analysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in a Robot*, MIT AI TR-258, 1972.
- [7] Sussman, G., Winograd T., Charniak, E., *Micro-Planner Reference Manual* MIT Artificial Intelligence Laboratory memo 203, 1970.
- [8] McDermott, Drew, Sussman, Gerald, *The CONNIVER Reference Manual*, Artificial Intelligence Laboratory memo 259a, January 1974.
- [9] 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.
- [10] Davies, Julian, *POPLER 1.5 Reference Manual*, School of Artificial Intelligence, University of Edinburgh, TPU Report no. 1, May 1973.
- [11] Hewitt, C., *How to Use What You Know*, IJCAI 75.
- [12] de Kleer, J., Doyle, J., Steele, G., Sussman, G., *Explicit Control of Reasoning*, MIT AI memo 427, June 1977.
- [13] Gelernter, H., *Realization of a Geometry-Theorem Machine*, in Feigenbaum, Feldman, *Computers and Thought*, 1963.