CONCEPTS IN PARALLEL PROBLEM SOLVING

by

William Arthur Kornfeld

M. S., Massachusetts Institute of Technology
(1979)

B. S., Massachusetts Institute of Technology
(1975)

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS OF THE
DEGREE OF

DOCTOR OF PHILOSOPHY IN
COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1982

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science
October 26, 1981

Certified by [illegible]
Carl Hewitt
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Departmental Graduate Committee

CONCEPTS IN PARALLEL PROBLEM SOLVING

by

WILLIAM ARTHUR KORNFELD

Submitted to the Department of Electrical Engineering and Computer Science
on October 26, 1981 in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy in
Computer Science

ABSTRACT

The Ether language, a language for parallel problem solving, is used as an implementational vehicle for two systems. Several aspects of these systems are novel and depend on the presence of parallelism.

One of these is a system that solves cryptarithmetic puzzles. This system is capable of pursuing several alternative hypotheses about assignments of letters to digits in parallel. The resources given to these various activities can be changed asynchronously with their running. Strategies for resource allocation are described. It is argued that parallel search offers greater flexibility in controlling the search process than can classical tree search algorithms.

The second is a program synthesis system that takes as input a description of the relationship between inputs and output of a Lisp function in a variant of first order predicate logic. The system synthesizes a Lisp program from this description. The system was designed to test the notion of pursuing multiple implementation strategies in parallel with skeptic strategies. The skeptic strategies attempt to show that certain proposed implementation strategies cannot possibly work. If these succeed, work on the successfully refuted implementation strategy is halted. Sometimes one skeptic can simultaneously refute many possible implementations that would otherwise have to be searched individually.

Several implementation details are discussed. The most significant of these is the notion of a "virtual collections of assertions." The discrimination net, common to implementations of pattern-directed invocation systems, has been completely replaced by a scheme that compiles the assertions and data-driven procedures into much more efficient message passing code. The form of this code is specified by the programmer for classes of assertion types and is suggested by their semantics. The technique significantly improves the flexibility and efficiency of this class of languages.

Ether code can be freely mixed with Lisp code while maintaining effective parallelism. Techniques of implementation that make this possible are discussed. The ability to freely mix Lisp code that is interpreted in the conventional manner allows us to build programs of significant size on conventional machines.

Thesis Supervisor: Professor Carl Hewitt

Title: Associate Professor of Electrical Engineering and Computer Science

CONTENTS

1. Introduction 6

2. A Theory Of Parallel Problem Solving 11

2.1 Falsificationist Philosophy 11
2.2 What This Has To Do With Problem Solving 12

3. The Ether Language 15

3.1 Activities 15
3.2 Sprites and Assertions 17
3.3 Goals 22
3.4 Viewpoints 24
3.5 More On Sprites 25
3.6 Historical Antecedents 26

4. A Cryptarithmetic Problem Solver 28

4.1 Description of The Problem 30
4.2 Relaxation 31
4.3 A Simple Depth-first Solution 33
4.4 The Basic Parallel Solution 37
4.5 Controlling The Search 41
4.6 Comparing Tree Search and Parallel Search 48

5. Synthesis Of Programs from Descriptions of Their Behavior 55

5.1 Our Domain 55
5.2 How We Describe Functions to Synthesize 57
5.3 Overview of How The Synthesis System Works 68
5.4 Generation and Refutation of Termination Clauses 74
5.5 Proposing Simple Constructors 79
5.6 Pursuing The Function Definition 83
5.7 Conditional Introduction 86
5.8 Skeptics By Progressive Testing 93
5.9 Related Approaches To Program Synthesis 94

6. How We Get Parallelism On Top Of Lisp 97

6.1 Message Passing Languages 98
6.2 Implementation of Activities 101
6.3 Lexical Environment Management 106
6.4 Sending Messages 107
6.5 Implementation of Processing Power 108
6.6 Modifying The Activity Structure 113
6.7 Other Mechanisms 114

7. The Implementation of Assertions and Sprites 117

7.1 A Review of the Properties of Sprites and Assertions 117
7.2 Virtual Collections of Assertions 119
7.3 Objects and Object Oriented Programming 122
7.4 A Very Simple Virtual Collection 127
7.5 The Length Virtual Collection 129
7.6 The Member Virtual Collection 140
7.7 Implemention of Cryptarithmetic 146
7.8 Miscellaneous Virtual Collections 148
7.9 Comparison With Lexical Retrieval Schemes 152

8. Epilogue 159

8.1 Nondeterminism and Computation 159
8.2 A Comparison With Constraint Networks 164
8.3 What We Need To Make Faster Ethers 169
8.4 Problems With Ether 173

9. Conclusions 178

10. Bibliography 180

FIGURES

Fig. 1. Code For Depth First Solution 34
Fig. 2. Basic Parallel Cryptarithmetic Program 38
Fig. 3. Heuristic Allocation Strategy Code 43
Fig. 4. Tree Search vs Parallel Search 50
Fig. 5. General Knowledge Sprites 70
Fig. 6. code to initiate test 71
Fig. 7. Goal For Equality 73
Fig. 8. Code For Proposing Termination Clauses 75
Fig. 9. Code For Disproving Termination Clause 77
Fig. 10. Function To Propose Possible Constructors 80
Fig. 11. After Finding A Termination Clause 83
Fig. 12. Code To Propose Possible Accumulators 85
Fig. 13. Test Proposing Function for Equal First Element 88
Fig. 14. Shell of Lisp Code 101
Fig. 15. State of an Ether Computation 103
Fig. 16. Definition Of An Event Record 104
Fig. 17. Definition of an Activity 105
Fig. 18. Code to Modify Absolute Processing Power Allocations 113
Fig. 19. Hierarchy of Concepts 118
Fig. 20. Length Assertion Replacement Procedure 120
Fig. 21. Length Sprite Replacement Procedure 121
Fig. 22. When Replacement Procedure for Contradictions 127
Fig. 23. Assert Replacement Procedure for Contradictions 128
Fig. 24. Length Assertion Replacement Procedure Repeated 130
Fig. 25. Length Assertion Message Handler 131
Fig. 26. Replacement Procedure for Length Sprites 133
Fig. 27. Length Sprite Message Handler 134
Fig. 28. Implementation of Length Sprites With Sequence Variable 137
Fig. 29. Replacement Procedures for the Member Virtual Collection 140
Fig. 30. Sprite Handler for Member Assertions 141
Fig. 31. Member Assertion Handler 144
Fig. 32. Letter Handler For One Of Assertions 147
Fig. 33. Sprite Virtual Collection for Universal Quantification 150
Fig. 34. Universal Quantification With Binding 151
Fig. 35. Sprite Replacement for Stifled Assertions 152
Fig. 36. Three Terminal Node Sprites 165
Fig. 37. Resistor Sprites 166
Fig. 38. Series Resistors 167
Fig. 39. Parallel Resistors 169

Chapter I Introduction

This work is concerned with the use of parallelism in the solution of problems in artificial intelligence. We are not concerned *per se* with the speed increase that can be attained by gainfully employing more than one processor in the solution of a single problem -- although we hope that the ideas about parallel *languages* described in this work will offer some direction to researchers concerned with parallel *hardware* architectures. A point we wish to stress from the beginning is that parallel *programs* can be talked about quite independently of any assumptions about the hardware these programs are run on. The programs we describe have been implemented and executed on a single processor machine. Chapter 6 is devoted to a discussion of the techniques of this implementation.

The work is divided into three main parts. The first part (chapters 2 and 3) consists of an overview of our main ideas, both conceptual and linguistic. Chapter 2 presents a theory of problem solving based on the work of certain philosophers of science, principally Karl Popper and Imre Lakatos. They develop an epistemological theory to account for the growth of scientific knowledge that is very "operational" in character. Their theories have motivated some new ideas about control structures for problem solving systems. These control structures are inherently parallel. Chapter 2 presents the theory of problem solving in the abstract and describes its relationship to other approaches to problem solving. Later chapters, through two example systems, make use of these control structures in solving problems.

Chapter 3 contains a brief description of the Ether parallel problem solving language. The material in this chapter is an elaboration of earlier work on Ether [30]. Ether has several attributes that make it a convenient base to develop large parallel systems. There is an **assert** statement in Ether by which assertions (representing facts or goals) can be made. There is also a procedural construct known as a *sprite*. Sprites have patterns that are capable of matching assertions. If an assertion has been made, and a sprite *activated* that is capable of matching it, the sprite will *trigger*. When a sprite triggers a new environment is formed as the result of pattern matching and the body of the sprite (arbitrary Ether code) is evaluated in that environment. Sprites and assertions obey certain properties that make them convenient to use in a highly parallel environment. Assertions obey a *monotonicity* property. Once an assertion has been made, it cannot be erased or overwritten. Sprites and assertions satisfy a *commutativity* property. Of a sprite and assertion that are capable of triggering, the effects of evaluating the body of the sprite after triggering do not in any way depend on the order of creation of the sprite and assertion. We argued extensively in [30] that these properties make it easy to construct large parallel systems without the possibility of "timing errors"; these properties are made use of here as well. The earlier work on Ether [30] introduced a *viewpoint* mechanism that makes it possible to introduce hypotheses and reason about them. Hypothetical reasoning is not possible in a monotonic system without some construct to grouping assertions, derived from a hypothesis, together. The viewpoint mechanism has been carried over essentially unchanged except for some new syntax that allows the code to be more concise.

There are significant differences between the current Ether language and the one reported earlier in [30]. Most of these differences have to do with the semantics and implementation of assertions and sprites. The assertions of the original Ether language (and, indeed, most assertion-oriented languages) are treated by the pattern matcher as uninterpreted syntactic forms. In the current system the semantics of assertions and sprites are understood by the system in a deep way. This, of course, requires designers of Ether systems to define the semantics of the various types of assertions. We have postponed a discussion of this until chapter 7. The language description and later examples should be understandable without entering into this discussion. Indeed, the effect on our example programs of our current treatment of assertions and sprites is to make them much simpler than they would otherwise be.

Chapter 3 also discusses the *activity* mechanism, the Ether analogue to a "process." When we have several tasks we wish to pursue in parallel, we create several activities for this purpose. The extension here over earlier work is the introduction of a *resource control* mechanism.. It is possible to make activities run at different rates with respect to one another. The allocations of resources to the various activities can be made asynchronously with their running.

The second major division of this document describes two example systems developed in chapters 4 and 5. These systems highlight different aspects of both the theory of problem solving developed in chapter 2 and the Ether language.

Chapter 4 describes a problem solver for a kind of mathematical puzzle known by the name *cryptarithmetic.* It is used to illustrate the concept of a *sponsor* that is introduced in our theory of problem solving. The function of a sponsor is to "watch" the progress of several competing solution methods, each searching for the answer, and modify resource allocations accordingly. The Ether resource control mechanism is used to implement the sponsors. The chapter develops several different algorithms for resource control based on aspects of the particular problem. Section 4.6 of that chapter compares the parallel search methodology we use with more conventional tree search techniques. It argues that there is no way of reorganizing the parallel search program as a tree search that will allow us the same degree of control.

Chapter 5 describes a toy program synthesis system. The system synthesizes a few simple Lisp programs from descriptions of their input/output relationships. The main motivation for developing this example is to demonstrate the techniques of *proposers* and *skeptics* developed in chapter 2, the theory of problem solving. Here we *propose* solutions (or pieces of solutions) to a problem and then test these using *skeptics*. Those proposals that survive the tests of the skeptics are accepted. An additional motivation for developing this example was to have a large system involving many sprites, assertions, activities, and viewpoints. Our techniques for implementing sprites and assertions are quite unique and required us to construct a system of significant size to test them.

While our system is certainly a "toy" and of no practical value, we believe the ideas used in its construction may be of some use in the construction of more practical program synthesis systems or systems to aid program development. A comparison with other program synthesis systems and suggestions for how the methods we make use of might be employed are the subject of section 5.9.

The third major division of this document consists of chapters 6 and 7 and describe the implementation of Ether. At the lowest level Ether is based on the actor theory of computation. All computation involving sprites and assertions compile into actor-style message passing code.

Chapter 6 is concerned with the "low level" techniques of this implementation -- the implementation of message passing and its interface to the normal Lisp environment. The activity notion that we use throughout the example systems is implemented as a mechanism for grouping events in the message-passing implementation. We explain how messages are sent, environments are maintained, and resource control is implemented. The material of this chapter should be of interest to those interested in message passing languages irrespective of the higher level code that compiles into it in our system. Another major theme of the chapter concerns our techniques for blending message passing with ordinary Lisp function calling in a way that preserves the parallelism that message passing provides while allowing us to build fairly large programs on conventional machines. The resulting system, although more cumbersome in certain ways than a pure actor language, has allowed us to construct the nontrivial problem solving systems described in the earlier chapters.

Chapter 7 concerns the techniques we use for the implementation of *sprites* and *assertions*. The techniques differ markedly from other assertion-oriented languages. In most languages, the storage of assertions and the triggering of sprites are accomplished by purely syntactic techniques that are not accessible to the user of the language. Our system allows the Ether subsystem implementer to design the implementation of storage and retrieval mechanisms making use of the semantics of the assertional types. The techniques are, in effect, a way of compiling assertion-oriented code into straight message passing code. Whenever an assertion is made or a sprite activated, what actually happens is a message is sent to an object that acts as a clearinghouse for the necessary information. Special message handlers are written for the individual assertional types that know how to encode the information in semantically meaningful ways. We show how our mechanisms maintain the properties of commutativity and monotonicity that were made use of in the example systems. There are a number of advantages with respect to both efficiency and expressibility of this approach. These are summarized in section 7.9.

Chapter 8 is a "wrap up" and covers a few different topics. The Ether programs we have written are nondeterministic, that is they can produce different (valid) results on different runs. Both the reasons for this, and some insights this gives us concerning nondeterminism are discussed in section 8.1. Section 8.2 contains some remarks on the relationship between Ether and "constraint networks." We show that the conceptual idea behind constraints can be more effectively implemented using Ether-like techniques than the usual network-style implementation. Following this are two sections that remark on the shortcomings of the current Ether language and its implementation.

About reading this document: Various sections of this document can be read independently. We strongly urge the reader to look at chapter 2, the theory of problem solving. It is short and supplies a framework for the rest of the document. The only "must reading" for comprehending later chapters is chapter 3, the discussion of the language. All the other chapters build make use of constructs introduced in it. Either of the two chapters containing example systems can be read independently of one another. The two chapters on implementation techniques can be read with only a brief glance at the previous two chapters from which the examples used derive.

On the use of code in this document: Code is an integral part of this work. Throughout the document code fragments are presented and explained. This code is included for a reason; it is not just filler. A careful attempt is made in the text to explain how the code works (and what point we are making by putting it there). Only a tiny fraction of the Ether code used to implement the two problem solving systems has been included. If we had included it all the size of this document would easily double. None of the Ether implementation has been included (save one very elegant function in figure 18); if that had been included the size would triple. Whenever code is included it is there to demonstrate some important feature. Almost all the code used in this thesis was "removed from service" so to speak. The exceptions to this are the examples in chapter 3 and section 8.2. The only "doctoring" of the code to improve readability that was done involves the use of certain programming constructs to get lexical binding to function properly. This is described in section 6.3.

Chapter II A Theory Of Parallel Problem Solving

The theory we are building upon in this chapter has been reported by Hewitt and myself under the title *The Scientific Community Metaphor* [32]. In that paper we construct a theory of problem solving meant to mimic some of the higher level aspects of the the kind of problem solving that is characteristic of scientific research. Much of the inspiration for this theory derives from the work of the espitemologist Karl Popper [49, 50]. Popper was interested in how knowledge could be gained through the methods of scientific research.

2.1 Falsificationist Philosophy

The question of how science comes to realize knowledge has been asked by philosophers in modern times since Descartes. There evolved two schools of thought known by the names *rationalism* and *empiricism*. Modern rationalism, though beginning with Descartes *Meditations*, has intellectual roots that go back to Euclid. The hallmark of rationalist philosophy is that there is a *core of irrefutable knowledge* from which all other knowledge is deduced. Science, according to the rationalists, consists of a process of deduction of new facts from already deduced facts. The work of Whitehead and Russell [69] and the set theorists early in this century represent a serious attempt to cast the whole of mathematics in this mold.

Empiricism, pioneered by Hume [28], disputes the claim that there can be a core of irrefutable knowledge from which scientific facts can be derived. He proposed instead that knowledge is gained by repeated observation. We see the sun rise every morning and so we come to the conclusion that the sun will continue to do so. He believed that all knowledge was of a similar kind, gained by repeated observation.

Popper rejects both of these traditions as being both logically unsound and not consistent with the historical development of science. He coined the term *justificationism* to encompass both classical rationalism and empiricism. He develops instead the doctrine of *falsificationism*. The falsificationist doctrine asserts that what we believe, we believe not because we have a justification for it (a proof or set of observations) but because *we have tried to falsify it and failed in the attempt.* Both deduction and observation play a role in his theories, but not the same roles they play in justificationist philosophy.

To Popper, science, or the advancement of science, consists of two aspects: *conjectures* and *refutations.*[†] Conjectures of scientific law are put forth by scientists *without any epistemological justification whatsoever.* He uses the term "bold conjectures" to emphasize this point. Once a conjecture is put forth it then becomes subject to the refutation process. From a theory (conjecture) deductions can be made and then the results of these compared with observation. If the consequents of the theory conflict with observation then the theory is *falsified* and must be discarded.

This basic doctrine of falsificationism later became known as *naive falsificationism.* It was obvious to everyone, including Popper, that theories don't ordinarily get discarded every time an anomaly is discovered concerning them. *Naive* falsificationism was replaced by *methodological* falsificationism which was extensively developed by Lakatos [36]. Methodological falsificationism augments the naive version in two principle ways:

1. When anomalies are discovered, theories are not discarded; rather, they are *adjusted* in a way that preserves as much of the original character of the theory as possible, yet does not imply the anomaly. Theories, then, instead of being isolated points form clusters where new theories are adjustments of old ones that account for some new observation. These clusters of theories Lakatos refers to as *research programmes.*

2. Research programmes do not exist in isolation. Rather there are many existing concurrently, each trying to account for the same group of phenomena. The degree to which a theory (or research programme) is accepted depends on how it fares in comparison with others.

2.2 What This Has To Do With Problem Solving

Looked at as a theory of problem solving this is an inherently parallel one. Research programs proceed in parallel and within each program attempts at falsifying and adjustment happen in parallel. The observations of these philosophers of science imply a definite "control structure" for problem solving. The problem solver consists of three components:

1. **Proposers** suggest new theories for evaluation.

---

† Hence the name of Popper's book, *Conjectures and Refutations* [49]. Lakatos presents a beautiful example of the process of conjecture and refutation in the discovery of mathematical theorems and their proofs in his book *Proofs and Refutations* [37].

2. **Skeptics** explore the *implications* of those theories and compare them with observations or beliefs. The findings of skeptics can be used by proposers to generate new theories. Skeptics were suggested for problem solving systems by Hewitt [22] and used by me in earlier work with Ether [30].

3. **Sponsors** compare different approaches to the same problem and adjust resources to their activities according to their relative merits.

There are many AI paradigms that bear a resemblance to the one we are proposing here. The classical control structure of "hypothesize and test" is very similar in spirit although is usually applied in domains where the forms of the hypothesis and the test are simple and uniform. The "debugging" theories of Goldstein [15] and Sussman [65] are perhaps the closest to ours. Their schemes both propose programs and adjust the programs based on bugs encountered in running them. The key addition of our approach is the idea of running many in parallel and having the ability to "stand back" and watch their respective progress. If you have only one "research program" in existence, there is the danger that it will box itself into a corner with a bug fix that was a mistake. Indeed Sussman comments in his thesis about the relationship between his own programming abilities and those of his program: "[I say to myself when I notice my bug fixes not improving the program] 'This is getting to be an ugly kludge and it is time to rewrite it.' I then reorganize the whole structure [of the program] ... HACKER just doesn't have anything like that ability."

The theory we propose differs in certain fundamental ways from some traditional approaches to problem solving. A hallmark of traditional problem solving ideas is the notion of having two places in the search space, where you are now (or, equivalently, your knowledge about where you are now) and where you would like to be, the "goal state." The problem solver in some sense tries to build a bridge between these two places. The search space consists of lots of little stepping stones across which the bridge is constructed. We don't know enough to construct a top view of this space so the bridge-building activity must proceed only on local knowledge. We know various ways of getting from stone$_x$ to the stones that are near enough to it that a bridge can be directly constructed between them. It is hoped that "heuristic information" is available to suggest which local paths are likely to be most fruitful. The seminal work from which this metaphor derives is the GPS program of Newell and Simon [47]. Explicit tables were used to show the degree of interconnection between nodes in the search space. There are two important assumptions that are made by the GPS model. They are:

1. That there is a symmetry with respect to two different ways of solving a problem. We can build

bridges forward from that current world state or backwards from the goal state.

2. That we can trust the "atomic bridges" as being invariably correct. If we can succeed in finding a way of connecting start to finish we can be absolutely certain of the correctness of the answer.

Robinson [53] introduced the resolution approach to theorem proving in 1965. Because of its great success from a theoretical perspective† it generated considerable interest until it was realized that resolution could only solve the most trivial of problems due to its thoroughly syntactic nature. The later group of pattern-directed invocation languages descended from the Planner language developed by Hewitt [20] were largely a reaction to the lack of controllability brought about by this syntactic orientation. Planner (as well as its implemented subset, Microplanner [64]) and subsequent languages still preserved the dual nature of starting states and goal states. There are two kinds of theorems: *antecedent* and *consequent*, the first of which moves from given facts to their consequents, the second from goal states to facts that imply them. The two kinds of reasoning are duals of one another. It was hoped that some proper mix of the two kinds of reasoning would yield systems that converged on an answer.

The theory of problem solving we propose does not view these two kinds of reasoning as duals. When we have a problem to solve, we *propose* a solution (or rather, as we will see, classes of solutions) and then use our forward reasoning capabilities to determine the implications of our proposals. When these implications conflict with what we know to be true, we know the original proposal was incorrect. Those proposals that remain uncontested we accept.

We have not yet said anything about how our proposers are constructed. Our theory of proposers involves two basic ideas. The first is that we look at *simple examples*‡ and then see if we can find theories that work on these simple examples. These theories, then, are tested on larger classes of examples and adjusted (or flushed) as appropriate. The second basic idea is the notion of a prototypical situation. Minsky's *frames* [45] and Schank's *scripts* [55] are both relevant here. They are both attempts to formalize the process of recognition, but the notions carry very naturally over to theory formations. They propose general *templates* with slots that need to be filled in to make a specific theory.

† Resolution is a theorem prover for the first-order predicate calculus that is provably complete and consistent.
‡ When we use the term *example*, we do not necessarily mean examples in the ordinary sense. Within the context of the program synthesis system developed in chapter 5, for example, we may be interested in the behavior of a function on a list consisting of only a single element, but we may not care to specify any characteristics of this element. These might be called *generalized examples*.

Chapter III The Ether Language

The language that is used throughout this work to implement our parallel programs is known as *Ether*. The language was first described by Kornfeld [30, 31] in 1979. Since that time it has been considerably extended and improved to the point where large, non-trivial parallel programs are possible. Ether is an extension of the Lisp Machine dialect of Lisp. The interpreter for Ether is an extension of the normal Lisp interpreter. Normal Lisp functions can be freely intermixed with Ether code. The Lisp code should not be viewed as an "escape mechanism." The programs we present makes use of the Lisp metaphor (functional and operational programming), the Ether metaphor (declarative, data driven programming), and, as we will see in chapter 7, the metaphor of object-oriented programming. The current generation Lisps suffer from certain shortcomings that make this mix somewhat difficult and inefficient. These shortcomings are discussed in sections 8.3 and 8.4. However, these shortcomings can be overcome, and we believe the combination of the these metaphors is quite practical and we hope will point the way towards practical AI languages in the future. In this chapter we present the constructs of the Ether language "as the naive user sees them." Of course, the host language, Lisp, is not a parallel language and has no facility for insuring that *sprites* (to be defined shortly) will function as they should. A rather extensive translation is done with the Lisp macro facility to transform the Lisp code into an implementation which is effectively parallel. How this translation is accomplished is the subject of chapters 6 and 7. The reader should not worry about this translation process and simply accept the programs, for now, at face value.

3.1 Activities

Activities are the basic parallelism construct of Ether. Whenever any code is executed, it is executed under the auspices of some activity. Activities are returned by the function `new-activity`. Activities form a directed acyclic graph. Each activity (save one, the *root activity*) has at least one activity preceding it in the graph known as its *parent activity*. When the function `new-activity` is called with no arguments, its parent activity is the one in which the `new-activity` function was evaluated. Alternatively, it can be handed an argument that indicates its parent(s). We tell Ether to evaluate code in an activity by using the `within-activity` function. An iterator function is supplied known as `continuously-execute` that keeps calling its argument again and again. So if we evaluated the following code:

```
(let ((a (new-activity))
      (b (new-activity)))
  (within-activity a
    (continuously-execute (print 'foo)))
  (within-activity b
    (continuously-execute (print 'bar))))
```

We would create two distinct activities running concurrently with one another, whose sole purpose would be to print rather boring messages on the terminal. It would look something like:

```
FOO
FOO
BAR
FOO
BAR
BAR
BAR
```

The exact sequence of prints is nondeterminate, yet the expected number of FOOs and BARs would be about equal. If the above code were evaluated at top level the parents of the activities bound to both a and b would be the special activity `the-root-activity`.

Activities have a parameter associated with them known as their *processing power.* The processing power of an activity represents the speed with which it can run. An activity with twice as much processing power as another will get about twice as much processing done in a given time quantum as the other. Processing power is a conserved quantity within the system. When an activity creates other activities it must give them some processing power for them to run. The processing power assigned initially to `the-root-activity` is arbitrarily chosen to be have the value 1. All of its subactivities that are runnable are assigned some fraction of this and at all times the total processing power for all activities within the system will sum to 1. Processing power decisions are made by certain defaults when the code does not explicitly specify otherwise. If an activity creates some number of subactivities they will all by default receive equal shares of processing power.

There are a number of functions that allow the user to override this default. One such function is `support-in-ratios` which tells the system to divide processing power in specified proportions. For example, if we modify the above code to:

```
(let ((a (new-activity))
      (b (new-activity)))
  (support-in-ratios
    children  (list a b)
    ratios    '(1 4))
  (within-activity a
    (continuously-execute (print 'foo)))
  (within-activity b
    (continuously-execute (print 'bar))))
```

we would be telling the system that we wanted activity b to get 4 times as much processing power as activity a. `Support-in-ratios` takes two arguments, a list of activities which must be children of the current activity, and a list of equal length containing numbers.† Processing power is divided proportionately with these numbers. What we would see on our console after executing this code would be an endless sequence of `FOOs` and `BARs` with about four times as many `BARs` as `FOOs`. The processing power assigned to the children of any activity can be modified at any time, completely asynchronously with the running of the activities. After a change in processing power allocations, the future running of the activities will be closely in accordance with these proportions.

There is one additional operation we can perform on an activity: we can *stifle* it. A stifled activity simply ceases to execute. Any processing power that was assigned to it is returned to be subdivided among its parent activity and that activity's children. An activity can be stifled by calling the function `stifle` with the activity as an argument.

3.2 Sprites and Assertions

Many Ether programs depend very heavily on their ability to manipulate objects that can be given a *declarative* interpretation. One primitive, `assert`, is used for making *assertions* (statements of one kind or another). Another construct, known as a sprite, has a pattern that can match classes of assertions. When an assertion has been made, and a sprite created with a pattern that can match that assertion, the sprite will at some point be *triggered* and code contained in the body of the sprite will be evaluated. The environment in which the body of the sprite is evaluated may be augmented with variable bindings from the pattern match. If we had assertional types `foo` and `bar`, and had executed the following:

† `Support-in-ratios` takes *keyword* arguments. A convention used in the code in this document is that keyword arguments will be placed in an italic font.

```
(assert (foo 20))

(when {(foo 20)}
  (assert (bar 15)))
```

We would expect (bar 15) to be asserted. Sprites can be made to match classes of assertions, rather than just the single assertion above, by using *variables* in the pattern. A variable is a symbol prefaced by an "=" that will match anything in that position. When the body of the sprite is evaluated, the variable becomes bound to the item that was in the respective position of the assertion. If we had written the following code:

```
(assert (foo 20))

(when {(foo =n)}
  (assert (bar →(- n 5))))
```

we would again expect (bar 15) to be asserted. When the sprite was triggered, n is bound to 20 and then (assert (bar (- n 5))) causes (bar 15) to be asserted because (- n 5) evaluates to 15. The Ether assert primitive currently obeys a *quasi-quote* convention [51]† in which subexpressions are normally unevaluated. Those subexpressions preceded by the symbol "→" are evaluated and this value replaces the subexpression in the final assertion.

It is worth noting that the order in which the assertion is made and sprite created is irrelevant to the final outcome. The sprite will be triggered in either case and the body will be evaluated in the identical environment in either case. This is a critical property for the proper working of the Ether language known as *commutativity.* It is this property that makes it possible to have highly parallel programs that function together in predictable ways. Often the producer of some information (the one doing the assert) and the consumer of that information (the sprite) will be in different sections of code running asynchronously with one another. The order that they will *actually* be executed in is not necessarily knowable from reading the program and may actually vary from one run of the code to the next. In future implementations of Ether-like languages on truly parallel machines the order in which they actually get executed may not even be knowable *in principle.*‡ The commutativity property enables our parallel programs to support subcomponents capable of interacting with one another in a productive manner.

† But see section 8.4.4 concerning quasi-quote.
‡ Special relativity puts certain theoretical limits on our ability to order events in time that happen at physically distinct locations.

When a sprite is activated, it becomes associated with the activity in which it was activated. As long as the activity has processing power the sprite will remain capable of triggering on assertions. When a sprite is triggered, its body is evaluated in the activity that the sprite is associated with.

We will illustrate the interaction of sprites and activities with some simple examples. Suppose we execute:

```
(let ((a (new-activity)))
  (when {(foo 100)}
    (stifle a))
  (within-activity a
    (when {(foo =n)}
      (assert (foo →(+ n 1)))))
  (assert (foo 1)))
```

The above creates a new activity, and in it creates a sprite that will cause a never ending sequence of assertions of the form `(foo 1)`, `(foo 2)`, `(foo 3)`, to be made. We have also created a sprite that waits for `(foo 100)` to appear and when it does the activity generating the foo assertions will cease functioning. The semantics of the language do not tell us precisely which will be the last assertion to get made; the activity may generate `(foo 102)` or `(foo 103)` or so before it is actually halted. We only know that it will stop soon after `(foo 100)` appears.

We will look at a few more examples.

```
(let ((a (new-activity))
      (b (new-activity)))
  (within-activity b
    (when {(bar =n)}
      (assert (bar (+ n 1)))))
  (within-activity a
    (when {(bar 100)}
      (stifle b))
    (when {(foo =n)}
      (assert (foo →(+ n 1)))))
  (assert (foo 1)))
```

Here we have two activities, each one generating increasing sequences of `foo` and `bar` assertions. However, in activity `a` we create a sprite that watches for `(bar 100)` to appear. When it does, the activity `b` is stifled and the output of `foo` assertions will be halted. Since the activities `a` and `b` have the same amount of processing power we would expect the highest `foo` assertion to be generated would be in the vicinity of `(foo 100)`.

The next example is a little trickier:

```
(let ((a (new-activity))
      (b (new-activity)))
  (within-activity a
    (when {(foo 100)}
      (stifle b))
    (when {(foo =n)}
      (assert (foo →(+ n 1)))))
  (within-activity b
    (when {(bar 100)}
      (stifle a))
    (when {(bar =n)}
      (assert (bar →(+ n 1)))))
  (assert (foo 1))
  (assert (bar 1)))
```

Here we have created two activities, one generating increasing sequences of `bar` assertions, and the other increasing sequences of `foo` assertions. However in each activity we have a sprite waiting for a specific assertion that is to be generated by the other. When it sees this assertion *the other* activity is stifled. What will happen when we execute this code? The result is nondeterminate and there are three possible outcomes.

1. The assertion `(foo 100)` will appear first causing the sprite in activity `a` to stifle activity `b` before `(bar 100)` is produced. In this case activity `a` will continue running and result in the generation of an endless sequence of `foo` assertions.
2. The assertion `(bar 100)` will appear first causing activity `a` to be stifled and resulting in an endless sequence of `bar` assertions.
3. Both critical assertions get generated sufficiently close in time to one another that both activities will be stifled. (Each sprite fires and succeeds in stifling the activity containing the other sprite.)

We could, of course, stack the deck by giving the activities bound to `a` and `b` different amounts of processing power as in the following:

```
(let ((a (new-activity))
      (b (new-activity)))
  (support-in-ratios
    children (list a b)
    ratios    '(1 4))
  (within-activity a
    (when {(foo 100)}
      (stifle b))
    (when {(foo =n)}
      (assert (foo →(+ n 1)))))
  (within-activity b
    (when {(bar 100)}
      (stifle a))
    (when {(bar =n)}
      (assert (bar →(+ n 1)))))
  (assert (foo 1))
  (assert (bar 1)))
```

making activity `b` run four times as fast. This will virtually assure that activity a will get stifled and activity b will forever generate `bar` assertions.

The above examples are silly and use the `stifle` primitive in ways that we would not use it in practice; they are meant only to make clear what it means for a sprite to be in an activity. The way we will be using the stifle primitive in subsequent chapters is when we have a *proof* that the sprites in a given activity are not going to produce any information that will be of use in solving the overall problem we will stifle that activity. There are several ways this can occur as the examples in subsequent chapters will make clear.

In addition to the *commutativity* property associated with sprites, assertions satisfy a *monotonicity* property. Once an assertion has been generated it cannot be erased. It is these two properties that allow us to give sprites a declarative interpretation. We can interpret them as implementing a forward chaining implementation of modus ponens. For example, if we have the logical statement:

$$\text{Human}(x) \supset \text{Mortal}(x)$$

we could embody this knowledge in the sprite:

```
(when {(human =x)}
  (assert (mortal →x)))
```

As long as this sprite were in some activity with processing power, and `(human Fred)` were asserted, we would be assured that the assertion `(mortal Fred)` will eventually appear.

The way sprites and assertions are implemented in the current Ether language differs significantly from the implementation of similar constructs in other languages. These new implementation techniques have enormous implications both in the power of the language and the efficiency of the Ether program. This is the subject of chapter 7.

3.3 Goals

Sprites, as we have just seen, give us a natural way to implement forward chaining. We also need the ability to do backward chaining; in other words, we may desire to know something and wish to initiate an activity for this purpose. At first glance we would think that the normal function calling ability of Lisp would solve this problem for us. For example, if we wanted to know if two objects were equal to one another, we could start up an activity and evaluate a function in that activity which contains Ether code that can determine if two objects are indeed equal. We would say:

```
(let ((a (new-activity)))
  (within-activity a
    (goal-equal object1 object2))
  (when {(equal →object1 →object2)}
    -- Whatever we would like to do knowing they are equal --))
```

The call to `goal-equal` is an ordinary function call and contains the necessary Ether code to determine whether or not two objects are equal. If the activity succeeds in its quest, an `equal` assertion will be made which will be detected by the following sprite.

The above solution has the following serious flaw. Suppose several concurrent activities in the system decide they want to know whether or not the same two objects are equal. They will each execute the function `(goal-equal object1 object2)` and the exact same work will be duplicated (needlessly). To avoid this, we have created a special *goal* facility. What the user would say, instead of the above code, would be:

```
(let ((a (new-activity)))
  (goal (equal object1 object2) a)
  (when {(equal object1 object2)}
    -- Whatever we would like to do knowing they are equal --))
```

A method has been written that knows how to determine if two objects are indeed equal. The skeleton for the code looks like:†

† The curious reader can skip ahead to figure 7 where the goal handler for `equal` assertions used by the program synthesis system is given.

```
(defgoal equal (x y) act
     --Code that can determine if two objects are equal--)
```

The goal mechanism has several features that make it easy to use. The first time the goal is invoked, i.e. the first time:

```
(goal (equal object1 object2) activity1)
```

is executed, a new activity is automatically created (we will call it for purposes of discussion the "`goal-equal-activity`") and the body of the `defgoal` is executed in this activity. This new activity is a subactivity of `activity1`. If at some later point another

```
(goal (equal object1 object2) activity2)
```

is executed, activity `activity2` is added to the list of parents of `goal-equal-activity`. Whatever processing power it has is added to the processing power of `goal-equal-activity`. The goal may be invoked any number of times and each time it causes processing power to be added to `goal-equal-activity` which is already in progress working on the goal. The parent activities (i.e. `activity1` and `activity2`) may change their processing power allocations for the goal and this change is then reflected in appropriate changes to the processing power allocation of `goal-equal-activity`. If any of the parent activities are stifled, they are removed from the list of parents. If *all* the parents of `goal-equal-activity` are stifled, its processing power allocation is reduced to 0, causing it to halt work. However, some future invocation of

```
(goal (equal object1 object2) activity3)
```

occurs, the `goal-equal-activity` will get processing power from `activity3` allowing it to continue work.

`Defgoal` methods can often determine that their services aren't needed. For example, if it is either definitely known that `(equal object1 object2)` or `(not-equal object1 object2)` there is no point in the activity continuing operation and it should be stifled. We would write the `defgoal` in the following manner:

```
(defgoal equal (x y) act
   ...
   (when {(equal →x →y)}
     (stifle act))
   (when {(not-equal →x →y)}
     (stifle act))
   ...)
```

The variable `act` is bound to `goal-equal-activity`. If `(stifle act)` is executed, this activity will be stifled. Special code is associated with these special goal activities that ensures that if they are stifled *from within themselves*, all their immediate parent activities are stifled. This will allow processing power allocated for this goal to be reclaimed automatically by the creator of the goal.

3.4 Viewpoints

All our discussion thus far has presupposed a global collection of assertions. We will be making heavy use of *hypothetical reasoning* in our example systems. Hypothetical reasoning is not possible in a system that requires assertions to be monotonic without some mechanism to make them relative to the hypotheses from which they were derived. Once an assertion is made it is not retractable and we are stuck. Our solution to this is a *viewpoint*† mechanism. All assertions and all sprites are created within some viewpoint. A sprite will trigger on an assertion only when it has been asserted in the viewpoint of the sprite or in a viewpoint that the sprite's viewpoint inherits from. The function `new-viewpoint` returns a new viewpoint, initially free of any assertions. Optionally `new-viewpoint` can be handed an argument that specifies parent viewpoint(s). If `v1` is a parent viewpoint for `v2` then all assertions present in `v1` appear also in `v2`. We use the function `within-viewpoint` to specify in the code which viewpoint the sprites and assertions actually happen in. `Within-viewpoint` takes as its first argument a viewpoint and then any number of forms to evaluate within this viewpoint. If we had the following code:

```
(let ((v1 (new-viewpoint))
      (v2 (new-viewpoint)))
  (within-viewpoint v2
    (when {(foo =n)}
      (assert (bar →n))))
  (within-viewpoint v1
    (assert (foo 5))))
```

The sprite would not trigger because it was activated in a different viewpoint than the one in which `(foo 5)` was asserted. If, however, we let viewpoint v2 inherit from v1 as in the following:

† The viewpoint mechanism we have currently use is quite simplistic. Barber [2] is developing a much more sophisticated viewpoint mechanism than the one presented here. The virtue of our mechanism is that it is clearly implementable and is of adequate generality for our purposes here.

```
(let ((v1 (new-viewpoint))
      (v2 (new-viewpoint  inherits-from v1)))
  (within-viewpoint v2
    (when {(foo =n)}
      (assert (bar n))))
  (within-viewpoint v1
    (assert (foo 5))))
```

the assertion `(bar 5)` would appear in the viewpoint bound to v2.

3.5 More On Sprites

All the sprites discussed so far have exactly one pattern. A sprite can have any number of patterns enclosed between the curly brackets as in the following:

```
(when {(foo =n)
       (bar →(+ n 1))}
   -- body -- )
```

the sprite will trigger iff a `foo` assertion is present and a `bar` assertion is present with a number 1 greater than that contained in the `foo` assertion. Such sprites are semantically equivalent to a nesting of sprites, as in:

```
(when {(foo =n)}
  (when {(bar →(+ n 1))}
    -- body -- ))
```

In fact, in the implementation, `when` is a macro that, when handed multiple patterns, expands into nested sprites like the one above. We introduce the concise notation because we often wish to check for many things at once and this page (barring margins) is only 6.5 inches wide.

There is one class of sprite pattern that implements a restricted form of universal quantification that is worth mentioning at this point. If we have the following sprite:

```
(when {(∀ n in    list-of-numbers
             check {(foo n)})}
  -- body --)
```

where `list-of-numbers` is bound, say, to the list `(1 3 5 7 9)`. The sprite will trigger iff the following assertions are made:

```
(foo 1)
(foo 3)
(foo 5)
(foo 7)
(foo 9)
```

Remember that this sprite, as all sprites, satisfies the property of *commutativity.* The order in which the assertions are made and the sprite activated is immaterial. It also, of course, does not matter what order the elements of `list-of-numbers` are in. The identical behavior would be gotten if the list were `(3 1 9 5 7)` instead. We will explain in section 7.8.1 how these sprites are implemented on top of Lisp.

We have now introduced the basic constructs and mechanisms of Ether. There is more to the story of programming Ether than what we have mentioned here. This chapter should serve as enough of an introduction that the examples in chapters 5 and 4 can be understood. The other aspects to programming in Ether will be explained in chapter 7.

3.6 Historical Antecedents

The seminal work in the field of pattern-directed invocation languages was Hewitt's Planner [21]. A subset of it implementing antecedent and consequent theorems was implemented as Microplanner [64]. This spawned several efforts at various sites. An excellent overview of this crop of languages through 1974 is given by Bobrow [4]. QA4 [54] introduced the concept of multiple *contexts*, much like our viewpoints, but did not allow concurrent access to them. There is a nontrivial amount of time required to switch from one context to another. QA4 was very heavily engineered so that ordinary Lisp properties (e.g. variable bindings) could be made context dependent. QLisp [70] took a subset of the ideas in QA4 and integrated them so that code could be made to run using the standard Interlisp control structure. Conniver [44] had many similar mechanisms to QA4 and introduced the concept of *possibility lists* to allow ordering or pruning of possibilities for backtracking. Amord [71] was the first language to consistently make use of the properties of commutativity and monotonicity.†

The most direct antecedent of the current Ether language was the original Ether language [30]. Here were introduced the notions of activities, viewpoints, assertions, and sprites. The notion of processing power, although mentioned in a "future work" section, was never implemented. The most significant difference between the current Ether and the original is in how assertions and sprites are implemented. The implementation techniques we use have important implications for both the efficiency and *expressivity* of the language. We have replaced a syntactic retrieval mechanism with one that is primarily

† There is an unfortunate confusion of terminology here. Amord is normally thought of as being *non-monotonic.* This is, however, a feature of its sequentiality. Amord does not allow multiple viewpoints. It is monotonic (in our sense) in that no information is ever thrown away, and is also commutative in our sense.

semantic. This idea is very important and will be discussed in chapter 7.

Amord [72] introduced the concept of using explicit Goal assertions. The original Ether language [30] borrowed this concept and somewhat extended it. The nature of this extension was somewhat of a hack and served to make activities work out correctly. It served this purpose, but at the expense of some modularity. The current solution fixes this problem. The reason for wanting to have "explicit goals" is so the reasoner can reason about them. We would like to have the ability to know what the goals are so we can control resources of activities working on them, and also be able to create activities attempting to refute them. We would also like to maintain the following capabilities in the goal mechanism:

1. The proposer of the goal need not necessarily know of the name of the procedure that works on it, nor what other activities are also interested in the goal.

2. It should be possible that all control for working on a particular goal *pass through one point*, i.e. that there should be one activity assigned to all work on the goal, and one place in the code that orchestrates work on the goal. We would like to be able to establish resource control procedures to control work on the goal that must know about all subactivites working towards it.

3. There may be several procedures for working on the goal which do not necessarily have to be known (at program writing time) to the handler of the goal.

The Goal mechanism presented satisfies all three of these constraints in a fairly clean manner.

Chapter IV A Cryptarithmetic Problem Solver

The problem-solving system in this chapter was picked to highlight the use of *sponsors* to compare different approaches and allocate resources accordingly. As explained in chapter 3 we have the ability to manipulate the relative rates (that is, their *processing power*) at which different activities run in parallel concurrently with their running. The idea we hope to demonstrate is that with the ability to manipulate the processing power of activities we can "guide" the problem solver to a solution more efficiently than could otherwise be done. Concurrently with running alternative methods of solution, we will also run an activity whose task is to *compare* the rates at which different methods are approaching the solution† and reallocate processing power based on these comparisons.

In addition to being a testbed for the use of heuristic information to guide a search via the manipulation of processing power, there are three other major themes that we will investigate with the system described in this chapter.

First we would like to use this example to make a point concerning *search strategies.* When most writers use this term, they are referring to different techniques for searching a *tree* using an inherently sequential algorithm. We will argue that *parallel* search is of a fundamentally different sort. We use the cryptarithmetic example because it appears superficially to be amenable to a tree search algorithm. After explaining the various parallel search strategies we will demonstrate that these algorithms can not, in any easy way, be written as "tree search" algorithms. This is the subject of section 4.6.

Secondly we wish to constrast two ideas in programming languages. The program synthesis system makes use of the *data-driven* programming metaphor where the user writes programs that consist of sprites that "watch" for new information to be learned. Part of the program implementing the system in this chapter makes use of a different metaphor known by the name of *constraints.* In a constraint-based system the world is conceptualized as a graph of nodes that are repositories for some local piece of information about the problem. The nodes are connected in a network. When new information is learned about the attributes of a node that might effect neighboring nodes in the network, the appropriate information is passed to them. Some other systems that employed this style of programming are described by Borning [5] and Steele and Sussman [59]). As we will see in chapter 7 the

† In order for this to be successful, there must of course be a metric by which the alternative methods can be compared. Our solution method, as we will see, lends itself quite naturally to such a metric.

implementation of our constraint network is but a special case of the implementation of sprites. These observations will become important when we come to compare the the issues of *expressive power* and efficiency of the two formalisms in section 8.2.

Lastly we wish to demonstrate a system that combines two common architectures for problem solving systems: *relaxation* and *hypothesize and test.*

In a system based on relaxation, internal data structures represent (implicitly or explicitly) potentially acceptable points in the search space. Computation proceeds in narrowing down these possibilities by employing knowledge of the domain in the structure of the computation. A classic example of the use of relaxation is the vision program of Waltz [66]. In his system there were various possible interpretations for the parts of a visual scene. For *pairs* of parts of the scene, there were only certain allowable consistent labelings. By local propagation of information about possible labelings for the individual parts of the scene, the system was able to relax to a single consistent labeling for the entire scene. The notion of relaxation is closely coupled with that of constraint networks because it is an obvious computational mechanism for implementing relaxation. One point we wish to emphasize about pure relaxation is that at any time the internal data structures will be consistent with any solution to the problem. Thus, if more than one solution is possible, pure relaxation will be unable to select only one of them. Further, even if a unique solution exists, a relaxation-based system may not be able to find it.

The hypothesize-and-test methodology allows the program to make assumptions that narrow the size of the search space; there is no guarantee that the assumption is consistent with any solution to the problem. The program continues to make hypotheses until a solution is located or it has been determined that no solution is possible with the current set of assumptions. There is no requirement that any hypothesis be correct and so mechanisms must be available that prevent commitment to any hypothesis until it has been demonstrated to be acceptable. The most commonly available mechanism is known as *backtracking*. Backtracking allows the program to return to an environment that would exist had that assumption not been made. The ability to create multiple activities gives us much more flexibility in designing control structures than backtracking allows. In fact, as we will see shortly, "backtracking" is but a special case of a whole family of search strategies that can be created.

As long as the search space is enumerable (a very weak assumption) hypothesize-and-test can be easily seen to be theoretically more powerful. If there are several consistent solutions, a pure constraint propagation system has no way to establish preference for one of them. Even if only one solution is possible a constraint propagation system will not necessarily find it; this will be demonstrated later by example. The proponents of constraint propagation point out that hypothesize-and-test is grossly inefficient in situations where constraint propagation can function. The example in this paper bears out this claim, although one recent study by Gaschnig [14] suggests there are situations in which pure backtracking is more efficient than constraint propagation.

One can, however, imagine a composite system that has aspects of both relaxation and hypothesize-and-test. In such a system, relaxation can be used to prune the search space, yet allowing hypothesize-and-test to continue the search where constraint propagation is not able to. A constraint language that can support the creation of such systems has been constructed by Steele [62]. Steele allows assumptions to be made and backtracking performed. The current work discusses another such system in which the hypothesize-and-test methodology allows more than one assumption to be pursued concurrently. This is made possible by the use of viewpoints and activities. We can create new viewpoints to hold the *results* of relaxation-type processing based on hypothetical assumptions. The work doing the propagation in these viewpoints will be contained in separate activities. The amount of processing power we give these activities depends on how likely we are to get useful results out of the assumption(s) represented by the viewpoint.

The research described in this chapter has a highly empirical character. We experimented with several different strategies for the reallocation of resources. Part of our message in this chapter is that not only are the parallel programs easy to create, but they are also easy to *tune* to take into account heuristic information that is available. The system described here is a demonstration that parallelism provides certain flexibility in the design of algorithms that make it convenient to make use of heuristic knowledge in ways that would be difficult or impossible otherwise.

The content of this chapter has already been reported in an abridged form in [34].

4.1 Description of The Problem

Cryptarithmetic problems, of the sort we are studying, were made famous in the AI literature by Newell and Simon [46]. Their interest in these problems was one of producing psychologically motivated models. We are only interested in them as abstract puzzles that involve searching through relatively large spaces; no "psychological validity" is expressed or implied in the descriptions our algorithms.

We are given three strings of letters, e.g. "DONALD", "GERALD", and "ROBERT" that represent integers when substitutions of digits are made for each of the letters. There is at least one possible assignment of digits for letters so that the numbers represented by the first two ("DONALD" and "GERALD"), when added, yield the number represented by the third ("ROBERT"). Any one of these assignments is a solution. In the problems we will be looking at, each will contain exactly ten letters. A solution consists of a bijection from these ten letters to the ten digits 0 through 9.

4.2 Relaxation

To understand how the relaxation process works for cryptarithmetic we will examine the problem mentioned in the previous section:

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

What can we say just by looking at it? By examining the second column from the left we can conclude that $E = 0 \lor 9$. With a little thought we discover that no other constraints can be learned about any of the other letters or digits without more information.

In Newell and Simon's original formulation of the problem for their production system model [46], they gave the system the additional constraint that $D = 5$. Many facts can now be derived. We list a few of them along with the reasons for the derivation.

1. $D = 5$. Given.
2. $E = 0 \lor 9$. column2.
3. $T = 0$. #1 and column6.
4. $E = 9$. #2 and #3.
5. $A = 4$. #4 and column4.
6. Carry-in(column5) = 1. #1 and column6.
7. R is odd. column5 and #6.
8. $R > 5$. column1 and #1.
9. $R = 7 \lor 9$. #8 and #7.
10. $R = 7$. #9 and #4.

We can go on like this, and in fact solve the whole problem this way. We are able to do this without making any additional assumptions beyond #1 above. We aren't always this lucky. The point to going through this is to realize that each of our deductions is centered around three kinds of objects: *columns*, *letters*, and *digits*. We can make certain deductions by examining what we know about the values of letters in a column and their carries in and out. When new constraints are learned about the values of letters, these constraints can be propagated to columns containing those letters. Similarly we can make use of the fact that no two letters can be the same digit and no two digits the same letter.

There are three different kinds of Ether objects in our cryptarithmetic problem solver. They are `columns`, `letters`, and `digits`. There are a number of assertions we can make:

`(possible-digits →letter →digit-list)` if asserted means that the only possible digits that `letter` can be are those given in the list `digit-list`.

`(possible-letters →digit →letter-list` if asserted means that the only possible letters that `digit` can be are those given in the list `letter-list`.

`(carry-in →column →n)` if asserted means the carry in of the column `column` is known to be `n`. `n` must be either 0 or 1.

`(carry-out →column →n)` if asserted means the carry out of the column `column` is known to be `n`. `n` must be either 0 or 1.

`(contradiction)` if asserted means some letter or some digit had no possible assignment.

Code that actually implements the constraint propagation is presented in section 7.7. Unlike the program synthesis system the kinds of processing that we do within each viewpoint is precisely the same. We have no need to define sprites that can be selectively activated in different viewpoints; the program can be written directly in the lower level message passing sublanguage to be introduced in chapter 6. We will have more to say about the relationships between the two styles of programming in section 7.7 and then again in section 8.2.

We can observe some things about the ability of a purely constraint-based system to satisfactorily derive a unique solution. First, if there is more than one possible solution it will not find any of them. Since the letter and digit assignments of each possible solution are certainly possible assignments, they will appear on the possibility lists attached to each node. Even if there is only one possible solution (or no possible solutions) the system may not find it (or discover that no solutions exist). For example, the "DONALD" + "GERALD" = and "ROBERT" puzzle has only one solution; the relaxation system described will quiesce before finding it. Nevertheless, the knowledge can be said to be "present" in the network; if the nodes of the network are instantiated with an assignment of leters to digits, the network will assert a CONTRADICTION iff the assignment is not a solution. In order to solve these problems in general we will have to augment our relaxation-based system with the ability to make and test assumptions in seperate viewpoints.

4.3 A Simple Depth-first Solution

The first cryptarithmetic problem solver we will present is one that does hypothesize and test by a kind of *depth-first* search. The depth-first search is implemented by a backtracking control structure. We present this first as an exposition of our basic methodology of hypothesize and test and then go on to show how parallel solutions are but simple variants of the more conventional depth-first approach. The complete code for the depth-first solution is shown in figure 1. We begin the search by evaluating the function `initiate-depth-first`. This creates a new activity (called `start-act`) and a new viewpoint (called `start-vpt`). It is in this viewpoint that we will learn whatever we can about the solution by relaxation *without* making any assumptions. We activate the following sprite in this viewpoint:

```
(when {(contradiction)}
  (Print "Problem not solvable.")
  (stifle start-act))
```

If a contradiction happens in this viewpoint (meaning that there is some letter for which there is no possible digit or digit for which there is no possible letter) then the problem, as given, is not solvable. When the activity *quiesces*, i.e. the system has relaxed as much as it can given the initial configuration, the following sprite in the figure triggers:

```
(when {(quiescent →start-act)}
  (if (total-solution (quiescent-letter-constraints start-vpt))
      (report-solution)
      (let ((minpair (select-forking-pair
                         (quiescent-letter-constraints start-vpt))))
       (depth-first (car minpair) (cadr minpair) start-vpt))))
```

The function `quiescent-letter-constraints` can be called with a viewpoint as an argument and returns a list of letters and their possible values in that viewpoint. Naturally, the viewpoint must be quiescent for this information to be well-defined. The predicate `total-solution` checks to see that

Fig. 1. Code For Depth First Solution

```
(defunc initiate-depth-first ()
  (slet ((start-vpt (new-viewpoint))
         (start-act (new-activity)))
    (within-activity start-act
      (within-viewpoint start-vpt
        (initiate-relaxation)
        (when {(contradiction)}
          (Print "Problem not solvable.")
          (stifle start-act))))
    (when {(quiescent →start-act)}
      (if (total-solution (quiescent-letter-constraints start-vpt))
          (report-solution)
          (let ((minpair (select-forking-pair (quiescent-letter-constraints start-vpt))))
            (depth-first (car minpair) (cadr minpair) start-vpt))))))
```

```
(defunc depth-first (letter alternatives parent-viewpoint)
  (if (null alternatives)
      ;If there are no viable alternatives, the there is no consistent assignment possible
      (within-viewpoint parent-viewpoint (assert (contradiction)))
      ;Otherwise, pick one letter and test it in a new viewpoint
      (slet ((v (new-viewpoint  parent parent-viewpoint))
             (a (new-activity)))
        (within-viewpoint v
            ;Make the assumpution in the newly created viewpoint.
            (assert (one-of →letter (→(car alternatives))))
            ;Let the implication of the assumption via relaxation happen in the newly created activity.
            (within-activity a
              (initiate-relaxation)))
            (when {(contradiction)}
              ;If we know the assumption to be incorrect, we should note that fact in the parent activity
              (within-viewpoint parent-viewpoint
                (assert (cant-be →letter →(car alternatives))))
              ;and we stifle the activity.
              (stifle a)
              ;We then recursively call the procedure on the remaining alternatives.
              (depth-first letter (cdr alternatives) parent-viewpoint)))
        ;If the activity has quiesced, we must first check if the problem has been solved; if so, we are done.
        ;Otherwise we must pick a new branch to go down in a depth-first fashion.
        (when {(quiescent →a)}
            (if (total-solution (quiescent-letter-constraints v))
                (report-solution)
                (let ((minpair (select-forking-pair (quiescent-letter-constraints v))))
                  (depth-first (car minpair) (cadr minpair) v)))))))
```

its argument, the list of possible digits assignable to each letter, contains precisely one possibility for each letter. This qualifies as a solution and we are done. More often, most letters will have many possibilities. We now wish to make assumptions about particular letters being particular digits. The search is "depth-first" in the following sense: After making one assumption, we will again allow the system to relax; when processing in *this* new viewpoint has relaxed we again see if we have a solution, and if not make another assumption of a letter being a specific digit. Eventually, after making some number of assumptions, we will reach a state in which each letter has only one digit. At any level in this chain of assumptions it is possible that a `(CONTRADICTION)` will occur. A `(CONTRADICTION)` is asserted in a particular viewpoint if it is discovered that there is a letter for which there is no possible digit assignment or a digit for which there is no possible letter assignment in that viewpoint. In this case we must "backtrack." In our program this will correspond to picking another possible assignment to the letter and pursuing it in a new viewpoint. If we run out of all possible digit assignments for a letter, then the viewpoint which decided those were then only possible assignments (i.e. the one next up in the chain) must be inconsistent. A `(CONTRADICTION)` is then asserted there. Note that no special code had to be written to get this behavior in the superior viewpoint. When the viewpoint was first created, a sprite was activated looking for contradictions in that viewpoint. It matters not whether the information the caused the contradiction was derived solely from manipulations in that viewpoint (or its superiors) or from facts learned in inferior viewpoints.

The question remains yet as to *which* letter to make assumptions about. At the end of the relaxation process of `initiate-depth-first` we must pick one. It seems to make the most sense to pick a letter with the fewest number of possibilities as this also represents the choice that engenders the least number of possible *failures* (that will cause unwanted backtracking). We cannot, of course, pick letters that have only one possibility (since no new assumptions can be made here). We thus must pick the letter with the smallest number greater than 1. If there are ties then one letter is picked arbitrarily. This selection is made by the code:

```
        (select-forking-pair (quiescent-letter-constraints start-vpt))
```

The function `select-forking-pair` returns a pair consisting of the letter and the list of possible alternative digits. Next we call the function `depth-first`, giving it the letter, the list of alternatives, and the current viewpoint as arguments.

We will now read through the the code for `depth-first` in figure 1 to see how the above strategy is implemented. The first thing we check is that there is at least one possible alternative. This is accomplished by:

```
(if (null alternatives)
    (within-viewpoint parent-viewpoint (assert (contradiction)))
    ...)
```

If there are no alternatives the parent viewpoint must be inconsistent and we assert this fact. Otherwise we create a new viewpoint (and activity) to pursue one of the possible alternatives. Within the viewpoint we assert:

```
(assert (one-of →letter (→(car alternatives))))
```

That picks the first element on the list of alternative digits and assumes the letter to have it as a value (in the newly created viewpoint). Within the newly created viewpoint we also activate a sprite watching for `(CONTRADICTION)` to appear. If one is noted, we assert in the `parent-viewpoint` that that particular letter assignment is not possible. Note that this may cause additional relaxation-type propagation to happen in the parent viewpoint. That this happens is signficant. It may be the case that learning this one new fact may lead to a contradiction being derived in the parent viewpoint. If this happens, all the work exploring the remaining alternatives still under consideration by `depth-first` is unnecessary. When the contradiction is flagged in `parent-viewpoint`, the activity exploring *it* will be stifled; since the activites created by `depth-first` are subactivities of this activity, they will also be stifled and control will shift higher in the tree.† Although we are calling this the "depth-first" solution it is not a totally sequential one; relaxation is performed in parallel although hypotheses are picked in a sequential, depth-first fashion.

If the current activity *quiesces*, i.e. it has finished the relaxation process and has not derived a contradiction, we check to see if we have found a unique solution. If so, we are done. Otherwise we must make additional assumptions and recursively call `depth-first` on them. We choose the new letter to make assumptions about, as we did before, by calling `select-forking-pair`, this time giving it te current viewpoint. The chosen letter, and its list of alternatives, are passed to depth-first.

† When we start exploring other algorithms that are more "parallel" the fact that relaxation in parent viewpoints can derive information that will be passed to subviewpoints becomes increasingly significant.

Fig. 2. Basic Parallel Cryptarithmetic Program

```
(defunc parallel-solve ()
  (slet ((start-vpt (new-viewpoint))
         (start-act (new-activity))
         (manager-activity (new-activity  name 'manager))
         (background-activity (new-activity  name 'background)))
    (support-in-ratios  parent     the-root-activity
                        activities  (list start-act manager-activity  background-activity)
                        factors     '(8. 1. 3.))
    (within-activity start-act
      (within-viewpoint start-vpt
        (initiate-relaxation)
        (when {(contradiction)}
          (Print "Problem not solvable.")))
      (stifle start-act))
    (within-activity background-activity
      (when {(quiescent →start-act)}
         (if (total-solution (quiescent-letter-constraints start-vpt))
             (report-solution)
             (let ((minpair (select-forking-pair
                                  (quiescent-letter-constraints start-vpt))))
               (parallel-fork (car minpair) (cadr minpair) start-vpt)))))
    (within-activity manager-activity
      (continuously-execute (allocation-strategy)))))
```

```
(defunc parallel-fork (letter alternatives parent-viewpoint)
  (if (null alternatives)
      ;If there are no viable alternatives, the there is no consistent assignment possible.
      (assert (contradiction))
      ;Otherwise, fork on each alternative
      (foreach
        digit
        alternatives
        (slet ((v (new-viewpoint   parent parent-viewpoint))
               (a (new-activity    parent start-act)))
          (add-current-explorers v a)
          (within-viewpoint v
            (within-activity a
              (initiate-relaxation))
            (when {(contradiction)}
              (delete-current-explorers v a)
              (within-viewpoint parent-viewpoint
                (assert (cant-be →letter →digit)))
              (stifle a)))
          (when {(quiescent →a)}
             (delete-current-explorers v a)
             (if (total-solution (quiescent-letter-constraints v))
                 (report-solution)
                 (let ((minpair (select-forking-pair (quiescent-letter-constraints v))))
                   (parallel-fork (car minpair) (cadr minpair) v))))))))
```

4.3.1 Review of Simple Depth-first

It is important that the reader understand the sense in which this *is* a depth-first search. If we find a problem with this viewpoint we move to the next higher one in the tree and select the next alternative from the list. If the list is empty then we pop back to the previous level (by asserting a contradiction in the viewpoint which is watched for by a sprite at the next higher level) and continue down the list of alternatives there. At any one time there is one viewpoint which is most detailed, i.e. it reflects the largest number of assumptions. It is not a classical tree search, in which the nodes are inactive data objects. The tree consists of viewpoints containing assumptions, each with an associated activity. Information learned in a viewpoint lower in the tree may cause new activity higher in the tree.

4.4 The Basic Parallel Solution

In the depth-first implementation discussed in the previous section, whenever an activity quiesced relaxation on a particular viewpoint we picked a new letter with a list of alternative digits from which to create new viewpoints. We went through this list one digit at a time, and waited for it to fail *before* going on to the next alternative. We could just as easily have started activities pursuing each of the alternatives in parallel. The code shown in figure 2 does just that. The code in figure 2 is actually generic for a whole family of algorithms.† The aspect of the code that distinguishes one member of the family from another is the *allocation strategy* used -- the scheme for deciding how much processing power to give to the various running activities. The two functions in figure 2, `parallel-solve` and `parallel-fork`, serve analogous roles as `initiate-depth-first` and `depth-first`. `Parallel-solve` is responsible for setting up the initial viewpoint and activity structure and calls `parallel-fork` each time it wants to sprout a new viewpoint and activity to pursue a new hypothesis. `Parallel-fork` calls itself recursively for the same purpose. You will notice in the body of `parallel-solve` the following code:

```
(within-activity manager-activity
      (continuously-execute (allocation-strategy)))
```

A special activity called the `manager-activity` is created whose sole function is to continually run a function called `allocation-strategy`. This function knows about the currently active investigations (those happening at the leaves of the tree of hypotheses) and continually modifies

† As we will see, there is a member of this family that is functionally equivalent to the depth-first program of section 4.3.

processing power allocations to the respective viewpoints based on *heuristic* information giving us an estimate of how likely the investigation is to aid the overall effort.

You will notice that `parallel-solve` creates three subactivities `start-act`, `background-activity`, and `manager-activity`. `Manager-activity` has already been explained. The parallel solution draws a distinction between the activity at the leaves of the hypothesis tree and others. Work happening at the leaves of the tree of hypotheses (the most important with respect to the entire search effort) occurs in separate activities, one for each hypothesis viewpoint-activity pair. Work on leaves higher in the tree happens all in one activity, `background-activity`. Non-leaf nodes are those that have already quiesced. The only way new work can be done in them is if a `cant-be` assertion is placed there due to a `(CONTRADICTION)` occuring in a node lower in the tree. The results from these nodes quickly propagate to the leaf nodes, so it was not deemed necessary to be able to carefully control the rates at which processing in these nodes happens; hence all processing in non-leaf nodes occurs in just one activity given a constant amount of processing at the beginning.

Activities pursuing work at the leaves are all children of the activity `start-act`. This is given a constant amount of processing power initially, but the way this processing power is divided amongst the active nodes is subject to change at any time. The activities pursuing these nodes are all children of `start-act`.

Looking back at figure 2 we will check how all this is accomplished. In the definition of `parallel-solve` we see first the creation of a viewpoint, `start-vpt`, in which the initial configuration is relaxed as was done by `depth-first`. We then create the three activities and assign them processing power. The amounts of processing power given were chosen empirically as amounts that give reasonable results. The code that allocates resources is the following:

```
(support-in-ratios
    parent      the-root-activity
    activities  (list start-act manager-activity  background-activity)
    factors     '(8. 1. 3.))
```

The three activities, `start-act`, `manager-activity`, and `background-activity` are create as children of `the-root-activity`, the highest activity in the tree. The "*factors*" argument to this function contains a list of integers and processing power is allocated in porportion to them. We see that 2/3 of the processing power goes to pursuing the leaf activities, 1/12 to the *manager-activity* which is in charge of continuously monitioring these activities, and 1/4 to the `background-activity`. Analogously with the `depth-first` example we initiate relaxation in the viewpoint `start-vpt` by executing:

```
(within-activity start-act
  (within-viewpoint start-vpt
    (initiate-relaxation)))
```

If a `(contradiction)` is found in this viewpoint, then the problem is not solvable and this fact is reported to the user. Also analogously with the code for `depth-first`, we create a sprite that watches for relaxation in the initial viewpoint to quiesce:

```
(within-activity background-activity
  (when {(quiescent →start-act)}
     (if (total-solution (quiescent-letter-constraints start-vpt))
         (report-solution)
         (let ((minpair (select-forking-pair (quiescent-letter-constraints start-vpt))))
           (parallel-fork (car minpair) (cadr minpair) start-vpt)))))
```

When quiescence has been reached (and the problem not yet solved) we again pick a letter and a list of possible digits and pass them to a function. The name of this function is `parallel-fork`, and as might be expected, this function will not pick just one of them at a time (thus implementing a depth-first search) but will concurrently begin searches on *all.* The code for `parallel-fork` is also contained in figure 2. If there is at least one possible digit, we iterate through the list of alternatives[†] and create a new activity-viewpoint pair for each. As our allocation strategy must know about the viewpoint-activity pairs that are currently active (i.e. at the leaves of the tree) we must mark them as such; this is what is accomplished by evaluating the function.

```
(add-current-explorers v a)
```

As we did in the case of the depth-first search, we initiate relaxation and watch for contradictions to appear.

```
(within-viewpoint v
   (within-activity a
     (initiate-relaxation))
   (when {(contradiction)}
     (delete-current-explorers v a)
     (within-viewpoint parent-viewpoint
       (assert (cant-be →letter →digit)))
     (stifle a)))
```

In the event of a contradiction we delete the viewpoint and activity from the current explorers and, as with the depth-first case, we make a `cant-be` assertion in the viewpoint directly above the current one: `parent-viewpoint`.

† The function `foreach` binds the variable `digit` to each of the elements of the list `alternatives` and evaluates its body.

Also, analogously with the depth-first case, we wait for quiescence and make a recursive call to `parallel-fork` if the problem is not yet solved.

```
(when {(quiescent →a)}
  (delete-current-explorers v a)
  (if (total-solution (quiescent-letter-constraints v))
      (report-solution)
      (let ((minpair (select-forking-pair
                             (quiescent-letter-constraints v))))
       (parallel-fork (car minpair) (cadr minpair) v))))
```

The reader may consider a comparison of the complete code for the depth-first program in figure 1 and and the parallel program in figure 2 at this point. The differences between them are minor. The point we wish to make via this comparison is that programming a true parallel search (where multiple incompatible hypotheses are being explored) is no more difficult than a more conventional sequential search.

4.5 Controlling The Search

The next question we should ask ourselves is: *"What is the heuristic character of the parallel search program in figure 2?"* The answer to that question depends on the behavior of the function `allocation-strategy`.

4.5.1 Trivial Strategies

One possible behavior of the function `allocation-strategy` could be to allocate all the available processing power to only one leaf processing power to only one leaf activity, say the first on the list of currently exploring activities. In this case the resultant search would be equivalent to the depth-first search of figure 1. The system would put all its processing power into one activity. If a contradiction were established in the associated viewpoint, the activity would be removed from the list of currently exploring activities and the next time `allocation-strategy` was run the next on the list of alternatives would be chosen to be the sole recipient of processing power. In the event that `parallel-fork` is called (because the running activity quiesced) the new activities are added to the front of the list of exploring activities; thus one of them would then be picked to get all the processing power.

Suppose `allocation-strategy` did nothing at all (i.e. it is a NOOP); what behavior results? In this case we would fall back on the default processing power distribution algorithm (described in section 6.6.1) which gives equal amounts of processing power to each of the activities. Each of the leaf nodes would then have equal amounts of processing power and will run equally fast. As new leaf nodes are created (because some other nodes have gone quiescent) they will be added to the list of currently exploring nodes and processing power will be redistributed so that all receive equal amounts. The character of this search would then be described as *breadth-first* because we allow the tree to grow equally fast in all directions.†

4.5.2 Manipulation of Processing Power

The reason for having a mechanism for manipulating processing power at all is so that we can make use of heuristic information to direct the search by giving more processing power to those avenues of exploration we consider most promising.

How can this be applied to the current problem? This author sees no way of looking at a partially constrained viewpoint and deciding that it is more or less likely to lead to a solution. Such a method, if available, could be used to direct processing power in a way that would cause the system to converge on a solution more quickly. This does not constitute the only criterion for deciding which activities are the most useful to pursue, however. The metric used does not consider the likelihood that this branch will eventually lead to a correct solution, rather it considers the likelihood the branch will *yield useful information with a minimal amount of processing.* Useful information, in this context, is either a solution to the problem (which is particularly useful) or a *contradiction.* Determining that a branch is bad quickly is valuable for two reasons:

1. We have successfully eliminated a wrong path from our search space. The more branches we can do this for quickly, the quicker the overall search process will proceed.

2. When a `(contradiction)` has been discovered, useful information propagates up the tree of viewpoint-activity pairs. Each time there is a `(contradiction)` asserted, a `cant-be`-type assertion is placed in the superior viewpoint. Often this will lead to more relaxation processing higher in the tree

† This statement is not literally true. Each node gets as much processing power as other nodes. Because of the nature of particular problems, the tree may grow more rapidly in some places than in others. However, this is the parallel strategy that comes closest in spirit to the standard definition of "breadth-first search."

where it is more valuable. Information learned there is then porpagated to lower viewpoints. Sometimes this will cause a `(contradiction)` in this viewpoint which will propagate information yet higher in the tree.

A reasonable measure of how likely we are to obtain either a solution or a contradiction from pursuing a particular viewpoint is one which is *high for those viewpoints that are already highly constrained* and low for viewpoints that are relatively unconstrained. After some experimentation we came upon the following formula for determining relative processing power allocations for the various different activities participating in the search:

$$((10 - n_1)^2 + \ldots + (10 - n_{10})^2)^2$$

where each n_i is the number of possible digit assignments for the letter i in the viewpoint. If the letters tend to have fewer possible digit possibilities, the sum terms $(10 - n_i)$ will tend to be large. Squaring this number, and squaring the final sum serves to accentuate the relative differences between the different viewpoints.

In order to implement this strategy all we have to do is design a function, called `allocation-strategy`, that computes this formula over all the currently exploring viewpoints (those at the leaves of the hypothesis tree), and then assigns processing power to the corresponding activities in proportion to the values resulting from the application of this formula. Figure 3 contains this implementation.

Fig. 3. Heuristic Allocation Strategy Code

```
(defunc allocation-strategy ()
   (support-in-ratios
     parent     start-act
     activities currently-exploring-activities
     factors    (forlist
                  vpt
                  currently-explored-viewpoints
                  (let ((status (quiescent-letter-constraints vpt))
                        (sum 1))
                    (foreach
                      pair
                      status
                      (increment
                        sum
                        (expt (- 10. (length (cadr pair))) 2)))
                    (max (expt sum 2) 1)))))
```

As the purpose of this function is to adjust processing power to the respective activities, the body consists only of a call to `support-in-ratios`. All the viewpoints and activities that represent leaf nodes in the hypothesis tree are stored in the lists `currently-explored-viewpoints` and `currently-exploring-activities` respectively. They are arranged in an order such that the nth element of `currently-exploring-activities` is an activity in which processing for the nth viewpoint in `currently-explored-viewpoints` occurs. It is the function of the two functions used in the definition of `parallel-fork`, `add-current-explorers` and `delete-current-explorers` in figure 2 to assure that this is so. We iterate through each of the viewpoints, and for each one, evaluate the function `quiescent-letter-constraints` that returns a list of pairs each consisting of a letter and a list of those digits that are possible assignments for this letter in this viewpoint. The rest of the code of figure 3 is merely a Lisp implementation of the above formula.

We recall that this function is evaluated in a separate activity known as the *manager-activity*. In the definition of `parallel-solve` in figure 2 we evaluated the code:

```
(within-activity manager-activity
      (continuously-execute (allocation-strategy)))
```

This will cause `allocation-strategy` to be called again and again asynchronously with the running of the other activities in the system. The frequency it gets called is governed by the amount of processing power allocated to `manager-activity`. Processing power is implemented in such a way that the percentage of time actually spent executing this function will be close to the processing power allocation with the maximum deviation from this being the time it takes to run the function once. This is the subject of section 6.5.

Implementing this resource allocation strategy caused a substantial gain in average performance over the simplest parallel strategy, the one in which `allocation-strategy` was the null function, implementing the parallel analogue of a "breadth-first" type search.

4.5.3 Concurrency Factors

We have observed in the allocation strategy discussed thus far that even though activities are running with different amounts of processing power that are related to our estimate of the utility of getting useful information back from them, there still seems to be so many activities running that they tend to thrash against one another. We would like to limit the amount of concurrency so that the running activities can get something done. For this purpose we introduce the notion of a *concurrency factor*. Instead of letting all runnable activities run, we pick the n most promising activities (using the metric above), where n is the concurrency factor, and give only those activities processing power and in the ratios defined by the metric. The optimal value for the concurrency factor is picked experimentally and is discussed below.

The value of the concurrency factor that yields the best result is a reflection of two aspects of the problem:

1. Many problems will have more than one valid solution. Thus, at any one time, we may be exploring several paths that will lead to valid solutions. In the event that this is the case for a given problem, it *may* still be to our advantage to explore both paths concurrently. The reason is that the difference between the convergence rates of the different branches may be sufficiently great that running both in parallel will ensure that we get the result of the quickest, even at the expense of wasting some time on the other one. None the less, we don't want to be exploring too many valid branches simultaneously.

2. The second aspect is related to the quality of our heuristic knowledge and the distribution of computational expense for picking bad branches in the search. Obviously if our heuristic knowledge were perfect, i.e. it could always point to the correct branch to explore next, the optimal concurrency factor would be 1 -- it should simply explore this best branch. If we are less sure we are about which is the best, more branches should be explored. Also, if the computational cost of exploring a bad branch is always small, a small concurrency factor would be appropriate. If, however, the cost of a bad branch can be very large we would want to use a larger concurrency factor. With a small concurrency factor we increase the probability that the problem solver will become stuck for a very long time. A limiting case of this is with a search space that is infinite (introducing the possibility of a bad branch that never runs out of possibilities) and a concurrency factor of 1. If the problem solver happens to pick one of these branches it will diverge.

For these reasons we wish to limit the total number of branches being explored simultaneously. The function `allocation-strategy` is modified to implement this strategy. Each time it is run, we pick the n most promising viewpoint-activity pairs (using the above defined metric) and then assign them processing power in proportion to the values of the metric on the respective viewpoints. The most reasonable value for the concurrency factor can only be picked experimentally. It depends on the presence of the two factors above in the "space" of possible problems handed to the system to solve.

4.5.4 Estimating Which Assumptions Are Most Valuable

Our strategy so far has been to use hypothesize-and-test on *one letter only* in each viewpoint. We sprout one new viewpoint and activity to test the hypothesis that that letter is each one of the digits it could possibly be in the parent viewpoint. This is not necessarily the best strategy. By hypothesizing a letter is a certain digit we may learn a lot or a little. We have "learned a lot" if we (1) discover quickly that a viewpoint is contradictory, or (2) cause a lot of constraint propagation activity that significantly increases our evaluation of the new viewpoint. One thing we have observed is that the amount we learn from assuming a letter is a particular digit *does not significantly depend on which digit we use.* In other words, if we assume the letter N is 2 and discover a contradiction, then we are likely to either discover a contradiction or signficantly constrain our solution by assuming N is any other digit on its list of alternatives. To take advantage of this phenomenon the program remembers what happened when it makes particular assumptions. When it creates a new viewpoint to study the result of assuming a letter is a particular digit the result is recorded in the parent viewpoint when it has completed. There are two possible results. If it led to a contradiction this fact is recorded. If it led to a quiescent (but consistent) state it records the difference of the evaluation metric applied to the parent viewpoint and the evaluation metric on the quiescent viewpoint -- our estimate of the amount of reduction that is likely to be obtained by assuming this letter to be a digit. Our new evaluation metric attempts to take this information into consideration. When assuming a letter L is a specific digit we use the old evaluation metric if we do not have have never assumed L to be a particular digit from this viewpoint; otherwise, we use the average of the evaluations for each of the resultant viewpoints. We then multiply this figure by the factor 1 + .5 * n where n is the number of letters that we have assumed L to be and determined that they lead to contradictions.

Now that we have a mechanism for taking advantage of information learned by making different assumptions we would like to ensure that a variety of choices are tried at each branching point. We will slightly modify the technique for picking the activities to be run at any given time (in accordance with the concurrency factor). Where c is the concurrency factor, we use the following algorithm to pick the c activities to run at a given time:

1. The activity with the highest evaluation is scheduled.

2. If n < c activities have been selected for running, the n+1st activity is (a) the one with the highest metric if it does not duplicate any of the first n activities in terms of which letter it is making an

assumption about for a given viewpoint, or (b) the highest rated non-duplicated activity unless the highest rated activity has a rating at least three times higher in which case we use the highest rated activity. The factor three was picked experimentally and is based on the following argument. There is a certain advantage in having a diversity of letters being tested because this gives us a greater chance to discover assumptions that will cause significant shrinkage by constraint propagation. However, there is also an advantage to running the activity that we have estimated will give us the best result. The factor three is the ratio of estimates for expected gain for which we would rather run the higher estimated test than one that will increase our diversity.

4.5.5 An Experiment

In order to test for the existence of a speed-up with concurrency we timed 10 problems using the final parallel algorithm described above for several concurrency factors. The problems tested are:

```
1) DONALD + GERALD = ROBERT
2) CRIME + TRIAL = THIEF
3) POTATO + TOMATO = VEGIES
4) MIGHT + RIGHT = MONEY
5) FUNNY + CLOWN = SHOWS
6) FEVER + CHILL = SLEEP
7) SHOVEL + TROWEL = WORKER
8) TRAVEL + NATIVE = SAVAGE
9) RIVER + WATER = SHIPS
10) LONGER + LARGER = MIDDLE
```

They were picked by a trial-and-error process of selecting possible problems and then running them to see if they have a solution. It is not known whether they have one or more than one solution. The program finishes when it has found one solution. These tests were run on the MIT Lisp machine, a single user machine designed for efficient execution of Lisp programs. The times represent processor run time only and are adjusted for time lost due to paging. The manager activity, which continually monitors the state of the search activities and readjusts processing power accordingly, receives a processing power allocation of .1. We tested with concurrency factors between 1 and 7. Numbers 2 through 7 each gave some improvement with 4 being the best. Here we report the results for concurrency factors 1 and 4. Times reported are in seconds:

	concurrency factor = 1	concurrency factor = 4	ratio
1)	377	140	2.69
2)	85	153	.56
3)	167	192	.87
4)	79	246	.32
5)	663	227	2.92
6)	2868	348	8.24
7)	241	112	2.15
8)	78	335	.23
9)	1920	554	2.55
10)	474	212	2.24
	----	----	----
total:	6952	2519	2.76

With a concurrency factor of 1 the algorithm becomes, functionally, the "depth-first" search described earlier. A concurrency factor of 4 represents the value which yields least average run time for the problems examined. Concurrency factors larger and smaller yield higher average values. We caution the reader not to take the numbers too seriously. We only wish to demonstrate that there is value in having a non-unity concurrency factor.

Some interesting facts can be learned by examining the data. Although the parallel solution beat out the sequential solution in only 6 of the 10 cases, these six cases are the ones for which the sequential solutions take the longest. In particular, problems 6 and 9 have show by far the longest times for the sequential solution and the time saving of the parallel solution is considerable. Similarly, for the cases in which the sequential solution finished quickly, the parallel solution tended to take longer. This phenomenon is fairly easy to explain. The parallel solution supplies "insurance" against picking bad branches in the search space. If the sequential solution happened to pick a bad branch (or several bad branches) there was no recourse but to follow it through. Similarly, if the sequential program found a relatively quick path to the solution, the extra efficiency of the parallel solution was not needed.

4.6 Comparing Tree Search and Parallel Search

The material contained in this chapter was presented in abridged form at the Second Workshop and Distributed AI, and the Seventh Internation Joint Conference on Artificial Intelligence during the summer of 1981. At both places some confusion resulted in the ensuing discussion as to the relationship between the *parallel* search methodology of this thesis, and the conventional and well-researched *tree* search algorithms. The question arose as to whether there is anything new in parallel search at all. In this section we will attempt to show parallel search, although close in spirit to tree search, is a richer programming formalism and allows the programmer to design algorithms with more flexible control than would be possible otherwise. Figure 4 diagramatically shows the relationship that we wish to explore. Refer to part A of the diagram. Although there are numerous different tree search strategies, they all have certain commonalities. There is always a tree consisting of nodes that represent some *state* of the problem solving process. These nodes are *static* -- they are data structures, not procedural in any sense -- and *homogeneous*, each being drawn from some well-defined space of possible nodes. Along with the tree, there is an associated algorithm for *picking the next node to be added to the graph*. In other words, the set of possible nodes to add is a datum available to the program that must pick them based on some metric. Different tree search strategies restrict the kinds of computation of this metric that can be done (usually for efficiency reasons). From the point of view of this discussion, we will assume the system has the capability to recompute the metric on each potential new node each cycle through the algorithm.

Part B of the diagram is a schematic representation of parallel search. In parallel search there are a number of *activities* running side by side. Each activity in that diagram is represented as two small loops† with a "throttle" below it. The throttle controls the amount of processing power given to the activity. Each activity is running an *arbitrary program* that can be similar in kind or entirely different from other activities. Information flows freely between activities; one activity can make use of information learned through its labors as readily as information learned through the labors of any other activity. In addition, activities can create new activities (and eliminate extant ones) at any time. There is a resource control algorithm that decides, based on current knowledge, how to allocate processing power to the various activities.‡ The resource control algorithm runs asynchronously with any of the other activities and can change its mind on resource allocation at any time.

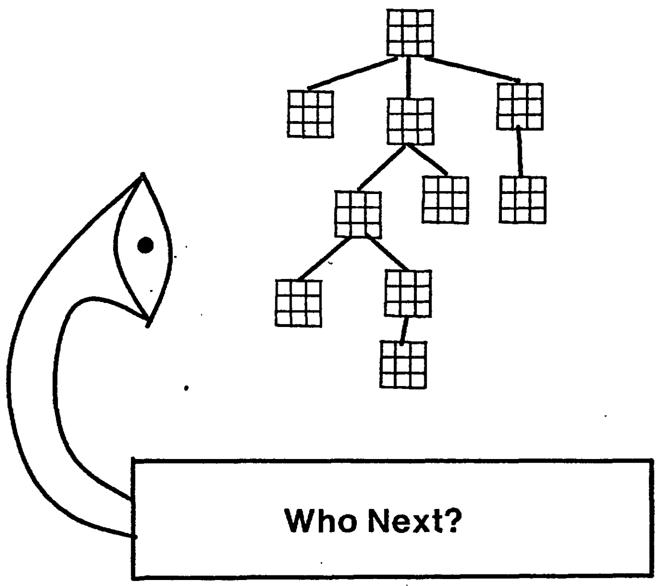
While the parallel search concept certainly appears richer than the concept of tree search, there still is the danger that we are making "much ado about nothing," that the problem could just as easily be phrased as a tree search. The remainder of this section is devoted to explaining why this is not so. The argument is one of programming practicality, not theoretical limitation. It will not be analogous to a proof that there exist context free grammars that are not parsable by any finite-state machine. It will be closer to an

† The reason we have used two loops instead of just one is to emphasize the fact that even within one activity there can be concurrency.

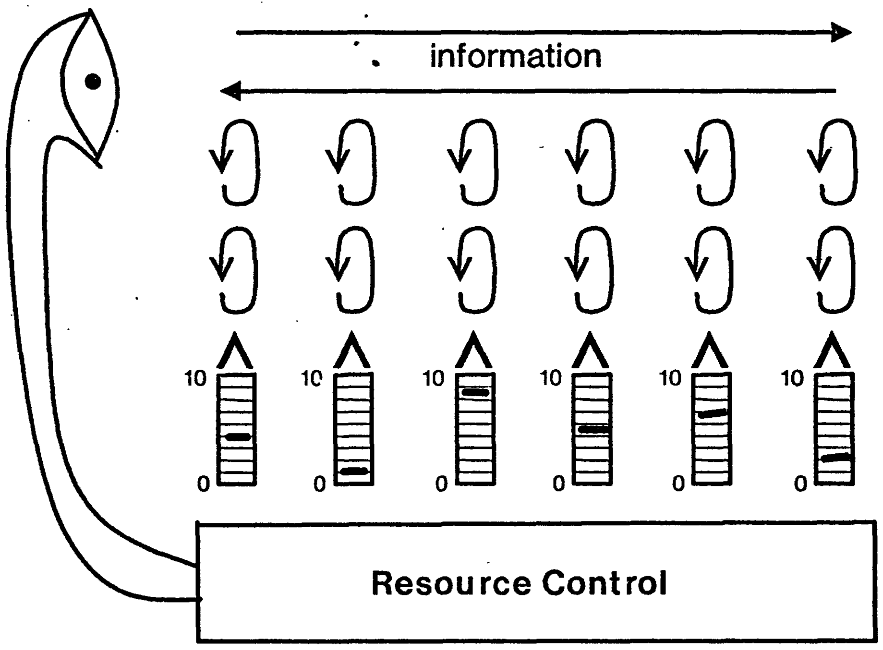
‡ In the cryptarithmetic example, and in the diagram, there is only one resource control procedure operating. This is a simplification of the most general state of affairs. There is a tree of activities, and each activity in this tree can use a different strategy for allocating processing power to its children. It happened in our strategy for controlling resources that the activity tree is essentially flat.

Fig. 4. Tree Search vs Parallel Search

(A) Tree Search:



(B) Parallel Search:



argument like "An Ether interpreter could not be implemented on a Turing machine." There is a theoretical result which flatly contradicts this, yet no programmer would consider taking on such a task. What we are arguing is that parallel techniques of the sort we have been using is a superior base from which to build good search algorithms, just as Lisp is a better base from which to build Ether rather than a finite state automaton, a read-write head, and an infinite tape. Before proceeding with this argument it is worth pointing out one other potential limitiation of tree search algorithms -- that they are inherently sequential; there is no way that parallel hardware could ever be put to use without modifying the concept in some way.

In order to view this problem as a tree search we must at least have a *tree* with nodes that represent different states of the problem. We were, in the solution to this problem, growing a tree of viewpoint-activity pairs. Could this be the "tree" of our tree search algorithm? The answer is clearly: *no.* The tree is not one of static objects, rather it consists of running programs. The programs are constantly advancing and only occasionally do new nodes of the tree get produced. Furthermore, resource control is in effect at all times. While several activities are running, changes of resource allocations can be made without any modifications to the structure of the tree at all. In a tree search algorithm the only way that heuristic information can be considered is in the design of the algorithm inside the "Who Next?" box -- the algorithm that adds new nodes to the tree.

If we are going to find a way of looking at this thing as a tree search algorithm, we must find more atomic objects that serve the purpose of the nodes of the tree. They must be static data objects. There are two choices for what these possible nodes might be: they might be momentary states of individual viewpoints (between constraint applications) or states of the entire collection of viewpoints. Both possibilities present problems.

Suppose that we have picked the first of these two options -- that the nodes of the tree are "snapshots" of individual viewpoints during relaxation. From a given node there are two kinds of "next" nodes that could be grown off of it:

1. New assumptions (what we did when we sprouted new viewpoint-activity pairs).

2. Application of individual constraints without making assumptions.

In order for there to be a tree search algorithm there must be a function to assign numerical values to nodes that could be grown on the tree so that one can be picked as "best." We will refer to this evaluation function as "F". Nilsson [48] gives examples of kinds of evaluation functions: "Attempts have been made to define the probability that a node is on the best path; distance or difference metrics between an arbitrary node and the goal set have been suggested; or in board games or puzzles a configuration is often scored points on the basis of those features that it possesses that are thought to be related to its promise as a step toward the goal." These are many different ideas, but they are similar at some level of abstraction: They imply a means of evaluation of the merits of expanding a node *based only on the characteristics of that particular node.*

In order that we get similar search behavior for our parallel algorithm recast as a tree search, we can make certain assumptions about what F must look like. It is always (we have supposed) better to apply constraints rather than make new assumptions. This is expressed in the parallel algorithm by our waiting for the activity to quiesce before making new assumptions. F must weight nodes that represent continued constraint propagation higher than assumption making. There are, however, many constraints that could be applied at once. Which one do we choose to apply *first.* No metric comes to mind for this, but we will suppose we have one or simply give all constraints that could be applied in parallel equal weight. Suppose now one of them is applied, generating a new node. Something very funny happens here (from the point of view of this being a tree search); after expanding one of those nodes *we will never want to expand any of the others.* This is because the same constraint can always be applied with greater advantage to the most newly generated node, not the old node. Because of our concepts of monotonicity and commutativity the constraints can be applied in any order with the same result. This was our reason for doing all of the reasoning in one viewpoint. We simply wish to learn all the facts that could be learned by local deduction. The end result of this is the following: Sections of the graph that represent relaxation within a viewpoint will always appear as simple linear chains -- every node will have an outdegree of one. Such a "computation" does not have any of the character, nor does it gain anything, from being looked at as a tree search.

We now consider the only possible place we would want to have branching in the tree, at places where we make assumptions. At each one of these places, as we have just learned, we will have long linear strings of nodes descending, but at least they will be descending in parallel (giving some credibility to this being a "tree"). In the original Ether algorithm we deemed it desirable to be pursuing the various hypotheses (now corresponding to parallel, descending chains of nodes) in parallel, but at different rates depending on the value of a metric. We now ask the question: "Is there a reasonable function F that will give us this behavior?" Remember that F is an evaluator of the desirability of expanding static nodes. It will be computing values for the tips of each of these descending chains and producing a number *solely on its own merits.* Because expanding a node via a constraint can only make the node seem more desirable than it was previously, it is hard to see how the other branch will ever get to run. Any function *F* that would yield the desired behavior would be unrelated to the desirability of expanding the node, and thus totally unintuitive.

Even if such an *F* could be devised, there is one very useful aspect of the original parallel program which would not fit this scheme in any reasonable way. Whenever a contradiction was discovered in a viewpoint, a `cant-be` assertion was placed in a superior viewpoint. This could lead to further relaxation-type processing in that viewpoint, the results of which would propagate to the leaves of the hypothesis tree. The nodes of a tree in classical tree search are *static.* We cannot change them once they are laid down. We cannot make a change high in the tree and have the results percolate down. To fit this inside a tree search, we would have to scrap the entire line(s) of reasoning below the node that we would have put the `cant-be` assertion in and build anew with this new information. This is so wasteful of information already learned that it would be of doubtful value.

Earlier in this discussion we mentioned that there were *two* possibilities for what the nodes of the tree could represent: states of individual viewpoints, and states of the entire computation. Does this latter possibility give us some hope? We can quickly see, by the argument given earlier, that constraint-type changes being made to the database will result in strict changes of new databases. We can also see by the argument in the previous paragraph, that in order for us to make use of information back-propagated, and without throwing away already-learned results, we can only allow one node to be expandable at any point in time -- any branching would force us to throw away information. We are able to handle the back propagation effect of the previous paragraph, but the resultant tree *is one long chain with no branching.* It makes no sense to talk about a "tree search" when the only possible trees that could be generated are simple chains. What this actually would look like would be a trace of the message-passing behavior of the Ether implementation looked at from the lowest level, the point where there cease to be separate concurrent activities as described in section 6.2. And the *F* function would not look anything like a metric of the desirability of expanding a node viewed in isolation.

I apologize for the tediousness of this long argument. It was to draw attention to what should now seem obvious: that parallel problem solving is a more flexible metaphor for programming a search than classical tree search. We spell it out in such detail for the benefit of those readers whose only model for search is classical tree search. New concepts sometimes seem strange and irrelevant without a tedious examination of the ways they compare to more familiar concepts. This is not to say that the metaphor of tree search was a mistake -- only that it was a first step and we may now be taking another step in the right direction. We have used the terms "depth-first" and "breadth-first" to describe kinds of parallel algorithms and these concepts are analogous to those found in the tree search literature (from which they were borrowed). Parallel search with research control might be considered a generalization of "best-first" search, and indeed there are parallel analogues to the classical A* algorithm as well.

There are other AI systems, not usually presented as tree searches, that have characteristics similar to tree search with respect to this discussion. Systems such as the Hearsay speech-understanding system [12], Lenat's system for mathematical discovery [40], and Davis' meta-rule formalism for rule-based systems [9, 10] are all examples. Each incorporates a resource control mechanism, but one that must make decisions about *each individual event.* Each involves some form of a "Who Next?" box. The ability to abstract away from the individual event level and apply resources to *activities* has been useful to us here, and may well be useful in these other domains.

Chapter V Synthesis Of Programs from Descriptions of Their Behavior

This chapter develops the second of the two example systems of this thesis. It is a program that generates Lisp programs from descriptions of their behavior using sprites.[†] These sprites have a declarative interpretation and can be thought of as a simple translation of a concise formula in first order logic.

The primary purpose in developing this system was to exemplify the techniques of *proposers* and *skeptics* developed in chapter 2. We were not specifically interested in advancing the art of program synthesis and the class of programs the system can generate in its current state of development is quite restricted. The contribution of this chapter is its explication of the use of concepts of parallel problem solving, not the power or generality of the resulting synthesis system.

This system does, however, lend itself to some interesting comparisons with other program synthesis systems in the literature. The bases for these comparisons are largely an artifact of the theory of problem solving we developed in chapter 2. Many of the mechanisms we develop as part of the synthesis system give us the ability to *reason* about programs and about specifications. These mechanisms might be used as subcomponents of program development systems. Remarks about these comparisons and possibilities are contained in section 5.9.

Our other major reason for developing the system described in this chapter was to exercise the linguistic concepts of Ether. We wanted to develop a system containing many assertional types with a rich semantics. The needs of the system in this chapter motivated many of the design considerations discussed in chapters 6 and 7.

5.1 Our Domain

The domain for the example in this chapter is program synthesis, but can be more generally thought of as "engineering design." That is, we have *specifications* for what we would like the final behavior of our engineering system (in this case, a computer program) to be. The system reasons about these specifications and produces a program as a result.

[†] At the time of this writing all of the system has been implemented with the exception of the skeptic activities described in section 5.8. The difficulties we had in implementing that aspect of the system are described in that section.

The general scheme we use for program synthesis is one of starting with *templates* that represent general plans for a program. The templates have *slots* that must be filled in with code fragments. Our original plan was to have a number of templates available, and have activities attempting to instantiate each template explored in parallel. Programming considerations, however, limited us to considering just one template, called *iterative accumulation*, that will be explained in section 5.3. The process of conjecture and refutation will be applied to filling in the slots of the template. A similar approach to program synthesis was taken by Goldstein [15] in the synthesis of programs to draw pictures from descriptions of their appearance.

The domain of expertise for our system consists of simple Lisp programs. In order to reason about programs, we must first have a language for doing so. The language we use is based on the formalism of assertions and sprites introduced in chapter 3. There are three general kinds of objects that we can talk about in our domain: sequences, atoms, and numbers. Sequences are the objects that get implemented as Lisp lists. When we *reason* about lists, we will think of them as sequences (that is objects having n positions, each filled by another object) rather than as CONS's (which have a CAR and a CDR). The code we generate will, of course, use CARs and CDRs. Atoms are objects that have no internal structure. The only things we can know about them is that they may or may not be equal to other atoms. Numbers are like atoms, but we can say a few more things about them (because of their total ordering). The Ether program synthesis sublanguage is "weakly typed" in that we can create an object that is not of any particular type. We can then, in some viewpoint, assert that object to be of some specific type and to have properties appropriate for that type.

There are a number of assertional types that can make statements about these objects. We define them below:

`(equal →object1 →object2)` means `Object1` is known to be equal to `object2`. `Object1` and `object2` can be any kind of object.

`(not-equal →object1 →object2)` means `object1` is known to be not equal to `object2`. `Object1` and `object2` can be any kind of object.

`(length →object →n)` means `object` is known to be a sequence of length `n`.

`(not-length →object →n)` means `object` is known to be a sequence with a length of something other than `n`.

(member →object1 →object2) means object2 is known to be a sequence that has at least one element, object1. Object1 may be of any type.

(not-member →object1 →object2) means objert2 is known to be a sequence, and Object1 is known to *not* be a member of it.

(sequence-element →sequence1 →object1 →n) means sequence1 is known to be a sequence, and it is known that object1 is in its nth position.

(less →number1 →number2) means both number1 and number2 are known to be numbers, and it is known that number1 is less than number2.

(not-less →number1 →number2) means both number1 and number2 are known to be numbers, and it is known that number1 is not less than number2.

There are several more assertional types that are used and will be introduced as needed.

5.2 How We Describe Functions to Synthesize

Each function to be synthesized by the system is described to the system by a set of sprites that state facts about the relationship between the input(s) and the output of the function. Because these sprites have an obvious declarative interpretation, the reader can think of them as having been generated by a process of "macro-expanding" a much more concise description in first-order logic. The logical description that led to the sprites is given along with the sprites that were actually input.† A macro-expander could have been written, but for the relatively small number of examples we are considering the effort would not have been worthwhile. We also give the logical description because it gives us a basis for comparison between our system and some others (that have similar logical descriptions as input).

† We do not state, unlike certain writers (e.g. Kowalski [35]) that sprites, or any other computational mechanism, in any sense "implement" logic. A logic is a formal system consisting of a language in which statements can be made and a proof procedure that can deduce certain statements from others. Logic is not any particular computational mechanism. The declarative interpretation of sprites allows the programmer to make reasonably certain that all new assertions produced by the sprite will be provable in the logic. Nevertheless, there are statements provable in the logic that the sprites derived from sentences in the logic will not, or *cannot* produce. There are meta-statements one can make about the class of provable sentences, such as the logic's completeness or consistency, that are not in any sense derivable or accessible through the sprites. It is important to keep the distinction between a logic and a computational mechanism consistent with that logic in mind.

It is important to understand the relationship between our means of problem specification and problem solution, and the remarks about the philosophy of science given in section 2.1. We are not interested in learning facts about the "real world." We *are* interested in producing an artifact (a program) that satisfies our specifications. In science we propose models for aspects of the real world and test our models against observations in the real world. In engineering design we propose models and test our models against the specifications. If our model suggests some relationship between the input and output, these relationships can be tested with the sprites that implement the specifications. The sprites are also used by the proposers. The way the proposers work is by looking at what the program does to simple examples and then hypothesizes program fragments that can handle those simple examples. The proposed code fragments are then tested against more complex examples by skeptics.

There are several functions that we have studied as part of this research. They are described in turn.

5.2.1 Reverse

The first one we will look at is the standard nondestructive reverse function common to all Lisps. An example is:

```
(reverse '(a b c d))  →   '(d c b a)
```

The function we will describe with our sprites takes one input called `input` of type sequence. The output of the function (called `output`) is of similar type.

A first order description of the reverse function is:

$$\begin{aligned}&\text{length(input)} = \text{length(output)}\\&\quad\wedge (\forall x)\ \text{member}(x,\text{input}) \leftrightarrow \text{member}(x.\text{output})\\&\quad\wedge (\forall n)\ \text{sequence-element}(\text{input},x,n) \leftrightarrow \text{sequence-element}(\text{output},x,\text{length(input)} - n + 1)\end{aligned}$$

The first conjunct states that the length of the input is the same as the length of the output. The second that something is a member of the input iff it is a member of the output; in other words, that one is a permutation of the other. The third conjunct gives the nature of this permutation as required by reverse.

There were six sprites that were actually input to the system, two for each conjunct.† The first conjunct expresses that the lengths of `input` is the same as the length of `output`. The two sprites that represent this information are:

```
(when {(length →input =n)}
  (assert (length →output →n)))

(when {(length →output =n)}
  (assert (length →input →n)))
```

They each wait for the length of the `input/output` to be known and when it is assert that it the length of the `output/input` is the same value.

The next conjunct says that the elements that are members of one are exactly those elements which are members of the other. Here, again, we represent this by creating two sprites. When one learns that there is some element that is a member of one it asserts it is a member of the other.

```
(when {(member =x →input)}
  (assert (member →x →output)))

(when {(member =x →output)}
  (assert (member →x →input)))
```

The last conjunct is expressed by the following two sprites:

```
(when {(sequence-element →input =x =n)
       (length →output =m)}
  (assert (sequence-element →output →x →(- m n -1))))

(when {(sequence-element →output =x =n)
       (length →output =m)}
  (assert (sequence-element →input →x →(- m n -1))))
```

The Lisp code produced from these specification was (effectively) the following:

```
(defun reverse (input) (reverse1 input nil))
(defun reverse1 (input accumulant)
  (cond
    ((equal input nil) accumulant)
    (t (reverse1 (cdr input) (cons (car input) accumulant)))))
```

There are a number of other functions that we have worked with that are described in turn through the

† One of the features of Prolog that is often praised by its proponents is its ability to have the same code reason in two directions. This is because functions are defined by binary relations, where one of the arguments represents the input and the other the output. We are free to unify a ground term with either of the two, thus causing it to compute in one direction or the other. In Ether we do not have a syntactic unfication mechanism and must create two sprites, one for each direction.

remainder of this section. The reader may wish to glance at them to get a feel for the breadth of the system, but there is no necessity to read through them thoroughly. The discussion continues with section 5.3 on page 68.

5.2.2 Intersection

This function finds the intersection of the elements of two lists. For example:

```
(intersection '(a b c d e f g) '(x g c d y)) → (c d g)
```

Inputs: `input1` and `input2` of type *sequence*

Output: `output` of type *sequence*

Formal description of its behavior:

$$(\forall x)\ \text{member}(x,\text{output}) \leftrightarrow \text{member}(x,\text{input1}) \land \text{member}(x,\text{input2})$$

Something is a member of the output iff it is a member of both inputs.

The sprites actually input to the program synthesis system were the following:

```
;If x is a member of both input1 and input2, then it is a member of the output.
(when {(member =x →input1)
       (member →x →input2)}
  (assert (member →x →output)))
;If x is a member of the output, then it is a member of both input1 and input2
(when {(member =x →output)}
  (assert (member →x →input1))
  (assert (member →x →input2)))
```

The code produced from these specifications was the following:

```
(defun intersection (input1 input2) (intersection1 input1 input2 nil))
(defun intersection1 (input1 input2 accumulant)
  (cond
    ((equal input1 nil) accumulant)
    ((member (car input1) input2)
     (intersection1 (cdr input1) input2 (cons (car input1) accumulant)))
    ((not-member (car input1) input2)
     (intersection1 (cdr input1) input2 accumulant))))
```

5.2.3 Setdifference

This function takes two lists as input and returns a list of all elements of the first which are not elements of the second. For example:

```
(setdifference '(a b c d e) '(b d e f g))  → (a c)
```

Formal description of its behavior:

Inputs: `input1` and `input2` of type *sequence*

Output: `output` of type *sequence*

$$(\forall x)\ \text{member}(x,\text{output}) \leftrightarrow \text{member}(x,\text{input1}) \wedge \neg\text{member}(x,\text{input2})$$

The sprites actually input to the program synthesis system were the following:

```
;If x is a member of the output, then it is a member of input1,
;and not a member of input2.
(when {(member =x →output)}
  (assert (member →x →input1))
  (assert (not-member →x →input2)))
;If x is a member of input2, then it is not a member of the output
(when {(member =x →input2)}
  (assert (not-member →x →output)))
;If x is a member of the input1, then it is a member of the output iff
;it is not a member of input2.
(when {(member =x →input1)}
  (when {(member →x →input2)}
     (assert (not-member →x →output)))
  (when {(not-member →x →input2)}
     (assert (member →x →output))))
```

The code produced from these specifications was the following:

```
(defun setdifference (input1 input2) (setdifference1 input1 input2 accumulant))
(defun setdifference1 (input1 input2 accumulant)
  (cond
    ((equal input1 nil) accumulant)
    ((member (car input1) input2)
     (setdifference1 (cdr input1) input2 accumulant))
    ((not-member (car input1) input2)
     (setdifference1 (cdr input1) input2 (cons (car input1) accumulant)))))
```

5.2.4 Unique-members

This function takes one list as an argument and returns a list containing exactly the unique members of the input. For example:

```
(unique-members '(a b a c c d a b))  → (a b c d)
```

Inputs: `input` of type *sequence*

Output: `output` of type *sequence*

Formal description of its behavior:

$(\forall x)\ \text{member}(x,\text{input}) \leftrightarrow \text{member}(x,\text{output})$

$\quad \wedge (\forall x)(\forall i)(\forall j)\ \text{sequence-element}(\text{output},x,i) \wedge \text{sequence-element}(\text{output},x,j) \supset i = j$

The first conjunct says that something is a member of the input iff it is a member of the output; the second conjunct that if an element appears in one position it cannot appear in another position of the output.

The sprites actually input to the program synthesis system were the following:

```
;If an object is a member of the input, then it is a member of the output
(when {(member =x →input)}
  (assert (member →x →output)))
;If an object is a member of the output, then it is a member of the input
(when {(member =x →output)}
  (assert (member →x →input)))
(when {(sequence-element →output =el =i)
       (sequence-element →output →el =j)}
   (assert (equal →i →j)))
```

The code produced from the specifications was the following:

```
(defun unique-members (input) (unique-members-1 input nil))
(defun unique-members-1 (input accumulant)
   (cond
      ((equal input nil) accumulant)
      ((member (car input) accumulant)
       (unique-members-1 (cdr input) accumulant))
      ((not-member (car input) accumulant)
       (unique-members-1 (cdr input) (cons (car input) accumulant)))))
```

5.2.5 Delete

This is the usual non-destructive delete function found in all Lisp dialects. An example is:

```
(delete 'a '(b a b c c a d b a b))  → (b b c c d b b)
```

Inputs: `atom`, of type *atom*, and `inlist` of type *sequence*

Output: `output` of type *sequence*

Formal description of its behavior:

$$(\forall x)\ \text{member}(x,\text{output}) \leftrightarrow \neg(x = a) \land \text{member}(x,\text{inlist})$$

The sprites actually input to the program synthesis system were the following:

```
;If an object is in the output, then it must also be in the inlist and not equal to the atom
(when {(member =element →output)}
  (assert (member →element →inlist))
  (assert (not-equal →element →atom)))
;If an element is in the inlist that is not equal to the atom, then it is also in the output
(when {(member =element →inlist)
       (not-equal →element →atom)}
  (assert (member →element →output)))
;If an element is in the inlist that is equal to the atom, then it is not in the output
(when {(member =element →inlist)
       (equal →element →atom)}
  (assert (not-member →element →output)))
```

The code produced from these specifications was the following:

```
(defun delete (atom inlist) (delete1 atom inlist nil))
(defun delete1 (atom inlist)
  (cond
    ((equal inlist nil) (reverse accumulant))
    ((equal atom (car inlist))
     (delete1 atom (cdr inlist) accumulant))
    ((not-equal atom (car inlist))
     (delete1 atom (cdr inlist) (cons (car inlist) accumulant)))))
```

5.2.6 Greatest

This function takes a list as an argument and returns the greatest number on that list. An example is:

```
(greatest '(3 5 9 7 6))  →  9
```

Inputs: `input` of type *sequence*

Output: `greatest` of type *number*

Formal description of its behavior:

$$\text{member(greatest,input)} \wedge (\forall n)\ \text{member(n,input)} \supset \neg(\text{greatest} < n)$$

The sprites actually input to the program synthesis system were the following:

```
;If an element is not equal to the greatest, then it is less than it.
(when {(member =element →input)
       (not-equal →element →greatest)}
  (assert (not-less →greatest →element)))
;The greatest element is in the input
(assert (member →greatest →input))
```

The code produced from these specifications was the following:

```
(defun greatest (input) (greatest1 input 0))
(defun greatest1 (input accumulant)
  (cond
    ((equal input nil) accumulant)
    ((less (car input) accumulant)
     (greatest1 (cdr input) accumulant))
    ((not-less (car input) accumulant)
     (greatest1 (cdr input) (car input)))))
```

5.2.7 Merge

This function takes two lists of numbers as input that it assumes are themselves ordered and outputs a list which contains all the elements of these two lists merged so that the order is preserved. An example is:

```
(merge '(1 4 8 11 13) '(2 5 6 15 16) → (1 2 4 5 6 8 11 13 15 16)
```

Inputs: `input1` and `input2`, both of type *sequence*

Output: `output` of type *sequence*

Formal description of its behavior:

$$
\begin{aligned}
(\forall x)\ \text{member}(x,\text{output}) \leftrightarrow\ & \text{member}(x,\text{input1}) \lor \text{member}(x,\text{input2}) \\
& \land\ \text{ordered}(\text{input1}) \\
& \land\ \text{ordered}(\text{input2}) \\
& \land\ \text{ordered}(\text{output})
\end{aligned}
$$

where the order relation is defined as follows:

$$\text{ordered}(\text{list}) \leftrightarrow (\forall x,y,i,j)\ [\text{sequence-element}(\text{list},x,i) \land \text{sequence-element}(\text{list},y,j)] \supset [x < y \leftrightarrow i < j]$$

The sprites actually input to the program synthesis system were the following:

```
;If an element is in input1, then it is in the output
(when {(member =element →input1)}
  (assert (member →element →output)))

;If an element is in input2, then it is in the output
(when {(member =element →input2)}
  (assert (member →element →output)))

;If an element is in the output, but not in input1, then it is in input2
(when {(member =element →output)
       (not-member →element →input1)}
  (member →element →input2))

;If an element is in the output, but not in input2, then it is in input1
(when {(member =element →output)
       (not-member →element →input2)}
  (member →element →input1))

;The elements of the list input1 are ordered
(when {(sequence-element →input1 =el1 =i)
       (sequence-element →input1 =el2 =j)}
  (when {(less →i →j)}
    (assert (less →el1 →el2)))
  (when {(less →el1 →el2)}
    (assert (less →i →j))))

;The elements of the list input2 are ordered.
(when {(sequence-element →input2 =el1 =i)
       (sequence-element →input2 =el2 =j)}
  (when {(less →i →j)}
    (assert (less →el1 →el2)))
  (when {(less →el1 →el2)}
    (assert (less →i →j))))

;The elements of the output list are ordered.
(when {(sequence-element →output =el1 =i)
       (sequence-element →output =el2 =j)}
  (when {(less →i →j)}
    (assert (less →el1 →el2)))
  (when {(less →el1 →el2)}
    (assert (less →i →j))))
```

The code produced from these specifications was the following:

```
(defun merge (input1 input2) (merge1 input1 input2 accumulant))
(defun merge1 (input1 input2 nil)
  (cond
    ((and (equal input1 nil) (equal input2 nil)) (reverse accumulant))
    ((equal input1 nil) (merge1 input1 (cdr input2) (cons (car input2) accumulant)))
    ((equal input2 nil) (merge1 (cdr input1) input2 (cons (car input1) accumulant)))
    ((less (car input1) (car input2))
     (merge1 (cdr input1) input2 (cons (car input1) accumulant)))
    ((not-less (car input1) (car input2))
     (merge1 input1 (cdr input2) (cons (car input2) accumulant))))))
```

5.2.8 Union

The union function takes two lists as input and returns a list that is the union of the two. If the same element happens to appear in both the lists it will appear only once in the output. An example is:

```
(union '(a b c d) '(b c e f))  →  (a b c d e f)
```

Inputs: `input1` and `input2` of type *sequence*

Outputs: `output` of type *sequence*

Formal Description of its behavior:

$(\forall x)$ member(x,output) $\leftrightarrow$ member (x,input1) $\vee$ member(x,input2)
$\wedge$ $(\forall x,i,j)$ sequence-element(output,x,i) $\wedge$ sequence-element(output,x,j) $\supset i = j$

The first conjunct expresses the fact that something is an element of the output iff it is a member of either input. The second conjunct says that each element of the output is unique, that is, if the same element appeared in both input1 and input2, there would be only one copy of it in the output. The is how the `union` function differs from `append`.

The sprites actually input to the program synthesis system were the following:

```
;If x is a member of input1, then it is a member of the output
(when {(member =x →input1)}
   (assert (member →x →output)))
;If x is a member of input2, then it is a member of the output
(when {(member =x →input2)}
   (assert (member →x →output)))
;If x is a member of the output, then it is a member of at least one of input1 and input2
(when {(member =x →output)}
   (when {(not-member →x →input1)}
      (assert (member →x →input2)))
   (when {(not-member →x →input2)}
      (assert (member →x →input1))))
;There cannot be duplication of elements in the output
(when {(sequence-element =x →output =i)
       (sequence-element →x →output =j)}
   (assert (equal i j)))
```

The code produced from these specifications was the following:

```
(defun union (input1 input2) (union1 input1 input2 nil))
(defun union1 (input1 input2 accumulant)
  (cond
    ((and (equal input1 nil) (equal input2 nil)) accumulant)
    ((equal input1 nil)
     (union1 input1 (cdr input2) (cons (car input1) accumulant)))
    ((equal input2 nil)
     (union1 (cdr input2) input1 (cons (car input2) accumulant)))
    ((member (car input1) accumulant)
     (union1 (cdr input1) input2 accumulant))
    ((not-member (car input1) accumulant)
     (union1 (cdr input1) input2 (cons (car input1) accumulant)))
    ((member (car input2) accumulant)
     (union1 (cdr input2) input1 accumulant))
    ((not-member (car input2) accumulant)
     (union1 (cdr input2) input1 (cons (car input2) accumulant)))))
```

5.3 Overview of How The Synthesis System Works

All the examples are of a certain kind that we might call "iterative accumulation." My original goal for this system would be that there be several such plans and that, based on characteristics of the program to be synthesized, some would be proposed and then the various aspects of the plan filled in by the process of conjecture and refutation. As it turned out, the programming difficulties in getting all the many levels of Ether working were of such magnitude that it precluded more breadth in this particular area. Never the less, after coding the system to know about more that one program schema we could (at worst) run them all in parallel and wait for one of them to work. Better, we could use knowledge about the applicability of different schemata to control resource allocations and, perhaps, prove that some schema could not possibly be made to work and reclaim all the resources given to it. Some recent work by C. Rich [52] suggests program schemata that are similar to the one we use as an example.

5.3.1 General Form of the Solution

An iterative accumulation is a simple loop in which we start with an `accumulant` that is initially the null list or 0, depending on whether it is a list or a number. The inputs (one or two) are lists, atoms, or numbers (but at least one must be a list) and we go down the list(s) and *accumulate* results in the `accumulant` variable. Although these are iterative programs, we have presented them in a tail-recursive form because they are easier to comprehend this way. All the generated code follows a schema something like the following:

```
(defun foo (input)
  (foo1 input accumulant-initial))
```

This merely establishes a call to the tail recursive secondary function generated. The original argument(s) is passed with one additional argument, the initial value of the `accumulant` (either `nil` or `0`). The definition of the tail recursive function is of the form:

```
(defun foo1 (input accumulant)
   (done-test accumulant)
   (null-input-test(s) recursive-call)
   (condition recursive-call)
   (not-condition recursive-call))
```

The *done-test* is a `null` test one or both of the inputs. When it is true the accumulant (representing the accumulated result) is returned. There may follow one or more *null-input-tests* for cases not covered by the *done-test*. These represent one or the other of the inputs becoming null when the *done-test* is:

```
(and (equal input1 nil) (equal input2 nil))
```

Following this are pairs of clauses containing one or more tests on the input(s) and its negation. Each of these clauses contain a recursive call. The form of the recursive call is quite restrictive. The inputs are either passed to the recursive call as is, or the CDR is passed. The last argument in the recursive call (the `accumulant`) contains an accumulating function that must be synthesized.

5.3.2 Setting Up A Working Environment

Throughout this work we will be creating viewpoints in which to test theories about partial programs. Whenever a viewpoint is created, it automatically inherits knowledge from its parent viewpoint(s); any fact present in the parent viewpoint will be available in this new viewpoint. There are, however, no sprites that automatically appear watching for assertions in the viewpoint. In order for any processing at all to happen in the viewpoint, there must (1) be sprites activated in that viewpoint, and (2) an activity with processing power in which the sprites have been activated. Whenever there is a new viewpoint created there is in fact a fair quantity of *active* knowledge (sprites) that we would like to be present. We would like each of the sprites that represent our definition of the function (the ones defined above) to be present. We would also like some sprites that represent *general knowledge* to be present. This set of sprites is shown in figure 5. We have defined a function called `activate-knowledge` that, when executed in a viewpoint, establishes these sprites. Our general procedure throughout the system is to, when a new viewpoint is created, execute this function.

In addition, there are several *prototype* viewpoints that are routinely established because they will be used so often for testing. The philosophy of our approach is that we wish to test simple theories on simple examples. These examples are derived from the prototype viewpoints. When an argument to a

Fig. 5. General Knowledge Sprites

```
;When two objects are known to be equal, many things can be deduced.
(when {(equal =object1 =object2)}
  (if* (not (eq object1 object2))  ;No need to do this if they are physically the same objects
        (when {(length →object1 =n)}
          (assert (length →object2 →n)))
        (when {(length →object2 =n)}
          (assert (length →object1 →n)))
        (when {(member =item →object1)}
          (assert (member →item →object2)))
        (when {(member =item →object2)}
          (assert (member →item →object1)))
        (when {(not-member =item →object1)}
          (assert (not-member →item →object2)))
        (when {(not-member =item →object2)}
          (assert (not-member →item →object1)))
        (when {(sequence-element →object1 =x =n)}
          (assert (sequence-element →object2 →x →n)))
        (when {(sequence-element →object2 =x =n)}
          (assert (sequence-element →object1 →x →n)))
        ;Two sequences cannot be both equal and not-equal
        (when {(not-equal →object1 →object2)}
          (assert (contradiction)))
        (when {(greater →object1 =obj)}
          (assert (greater →object2 →obj)))
        (when {(greater →object2 =obj)}
          (assert (greater →object1 →obj)))
        (when {(less →object1 =obj)}
          (assert (less →object2 →obj)))
        (when {(less →object2 =obj)}
          (assert (less →object1 →obj)))
        (when {(value →object1 =n)}
          (assert (value →object2 →n)))
        (when {(value →object2 =n)}
          (assert (value →object1 →n)))))
;If a sequence is of length 1, then everything that is a member of it must be equal
(when {(length =sequence 1)
       (member =object1 →sequence)
       (member =object2 →sequence)}
  (assert (equal →object1 →object2)))
;If a sequence is known to be of length 0, then it is equivalent to the special sequence NIL.
(when {(length =sequence 0)}
  (assert (equal →sequence →nill)))
```

function is a sequence, some of the examples we will want to look at include ones where the sequence has no elements, i.e. it is equal to `nill`.† Other possible configurations we will want to look at include ones where the sequence contains a list of one elements or two elements. Since these are prototype viewpoints we won't indicate in them anything about the nature of the objects known to be members of these sequences. For a prototype containing a list of two elements we won't, for example, know whether the elements are themselves atoms or sequences, or whether they are equal to one another, or anything else about them. In further "experiments" executed by the program synthesis system, we will create new

† We use "`nill`" to represent the object normally found in Lisp systems, `nil`. Our reason for doing this is that `nill` is not just a symbol, it is a variable bound to a complex object the nature of which will be discussed in chapter 7. It is not possible in Lisp to bind the symbol `nil`.

viewpoints that inherit all the information from these prototype viewpoints and may assert additional facts. For our purposes, we create the following viewpoints:

1. If the function being synthesized has only one argument, or has only one argument that is a sequence, we create three prototype viewpoints, in which the sequences are asserted to contain zero, one, and two elements. If there is another argument (one that is not known to be a sequence), we will not assert anything about it.

2. If the function being synthesized has two arguments that are both sequences there are a total of nine different prototype viewpoints that are established. These represent the cross product of zero, one, and two elements being tried out on each.

The tests are known by the names: `null-test`, `singleton-test`, and `doublet-test`. Figure 6 shows the function that actually initializes these prototype viewpoints.

Fig. 6. code to initiate test

```
(defunc initiate-test (test-name object)
  (selectq test-name
    (null-test
     (assert (equal →object →nill)))
    (singleton-test
     ;Create a new object that will become the single object of the sequence
     (let ((el (new-object  instance-prefix 'singleton-element)))
       (assert (length →object 1))
       (assert (sequence-element →object →el 1))))
    (doublet-test
     (let ((el1 (new-object  instance-prefix 'first-doublet-element))
           (el2 (new-object  instance-prefix 'second-doublet-element)))
       (assert (length →object 2))
       (assert (sequence-element →object →el1 1))
       (assert (sequence-element →object →el2 2))))))
```

We see that in the case of the `null-test` it merely asserts that the object is equal to `nill`. For the `singleton-test` case we create a new-object by executing the function:

`(new-object` *instance-prefix* `'singleton-element)`

that will return a new Ether object with absolutely no attributes. It could be a sequence, number, or elephant for that matter, if Ether had sprites that could reason about animals.† We then assert that the

† We do give `new-object` one argument, called the *instance-prefix*, which is for debugging purposes only. If in the middle of debugging we have occasion to print the object it will print as something like `singleton-element6` so that we will have some idea where it came from.

length of `object` (which is bound to one of the inputs, already known to be a sequence) is 1. We finally assert that the newly created object is in the first position of `object`. In the case of a `doublet-test` we create two new objects and assert that the length of the input, `object`, is 2. We finally assert that these two new objects are in the first and second positions of `object` respectively. This is all there is to it.

We arrange for other parts of the system to have access to these prototype viewpoints through the relation `viewpoint-for-test`. For example, if we wished to access the prototype viewpoint that used the `null-test` for the first argument, and the `singleton-test` for the second argument, we would activate the following sprite:

```
(when {(viewpoint-for-test (null-test singleton-test) =vpt)}
   ... vpt is now bound to this viewpoint in the body of the sprite. ...)
```

What we will typically do is create a new viewpoint that inherits from this prototype and make additional assertions in there, as in:

```
(when {(viewpoint-for-test (null-test singleton-test) =vpt)}
  (let ((v (new-viewpoint  inherits-from vpt)))
     (within-viewpoint v

               Here we make additional assertions in this viewpoint for our experiment.)))
```

We will see our first application of this in section 5.4.

We will frequently have occasion to decide if one object is equal to another. A `defgoal` has been designed for this purpose. Most of the "expertise" of the goal handler is with sequences. It is shown in figure 7. It is instructive to read through this code. The first two sprites,

```
(when {(equal →object1 →object2)}
  (stifle act))

(when {(not-equal →object1 →object2)}
  (stifle act))
```

check for the answer already being known. If it is the case that they are both known to be equal or not equal then there is no point continuing with this computation and the activity is stifled. This way any activities that have invoked this goal will reclaim the processing power assigned to it. The next three sprites are actually dedicated to *disproving* the equality of the two objects. If it can be shown that the lists are of different lengths the two objects are known not to be equal.

Fig. 7. Goal For Equality

```
(defgoal equal (object1 object2) act
  ;If the objects are known to be equal, the activity is no longer needed.
  (when {(equal →object1 →object2)}
    (stifle act))
  ;If the objects are know not to be cqual, the activity is no longer needed.
  (when {(not-equal →object1 →object2)}
    (stifle act))
  ;If the objects are sequences of different lengths, they are not equal.
  (when {(length →object1 =n)
         (length →object2 =m)
         (not-equal →n →m)}
    (assert (not-equal →object1 →object2)))
  ;If something is a member of one and not the other, they are not equal.
  (when {(not-member =el →object1)
         (member →el →object2)}
    (assert (not-equal →object1 →object2)))
  (when {(not-member =el →object2)
         (member →el →object1)}
    (assert (not-equal →object1 →object2)))
  ;If there is an element of one that is not equal to the element in the corresponding position
  ;of the other, then they are not equal.
  (when {(length →object1 =n)}
    (foreach m (list-of-integers from 1  to n)
       (when {(sequence-element →object1 =el1 →m)
              (sequence-element →object2 =el2 →m)
              (not-equal →el1 →el2)}
         (assert (not-equal →object1 →object2)))))
  ;If they are sequences and their elements agree in every position, they are equal.
  (when {(length →object1 =n)
         (∀ m in   →(list-of-integers from 1 to n)
              check   {(sequence-element →object1 =el →m)
                        (sequence-element →object2 →el →m)}))}
    (assert (equal →object1 →object2))))
```

```
(when {(length →object1 =n)
       (length →object2 =m)
       (not-equal →n →m)}
    (assert (not-equal →object1 →object2)))
```

If it can be shown that one has a member that is known not to be a member of the other than they cannot be equal.

```
(when {(not-member =el →object1)
       (member →el →object2)}
  (assert (not-equal →object1 →object2)))

(when {(not-member =el →object2)
       (member →el →object1)}
  (assert (not-equal →object1 →object2)))
```

The last two sprites in figure 7 check each element of the respective sequences for their equality or non-equality. If it can be shown that there are two elements, `el1` and `el2` in corresponding positions that are known not to be equal, then `object1` and `object2` are not equal.

```
(when {(length →object1 =n)}
  (foreach m (list-of-integers from 1  to n)
     (when {(sequence-element →object1 =el1 →m)
            (sequence-element →object2 =el2 →m)
            (not-equal →el1 →el2)}
       (assert (not-equal →object1 →object2)))))
```

If, on the other hand, it can be shown that every element of one of the sequences is equal to the corresponding element in the other sequence, then we can conclude the two sequences are equal.

```
(when {(length →object1 =n)
       (∀ m in    →(list-of-integers from 1 to n)
           check   {(sequence-element →object1 =el →m)
                   (sequence-element →object2 →el →m)}})}
  (assert (equal →object1 →object2)))
```

5.4 Generation and Refutation of Termination Clauses

You will remember from the description of the iterative accumulation-type function that each begins with a `cond` clause that specifies the end of the iteration, that is the point where the `accumulant` is returned. Each of the conditions of the clause consists of a test to check if one or both of the input sequences is equal to `nill`. Our first step in synthesizing a function is to decide what done condition to use. This will be our first demonstration of the interaction of proposers and skeptics in the synthesis of a piece of the function. For some of the functions: `reverse`, `unique-members`, `delete`, and `greatest`, that either have only one input, or only one input that is a sequence, there is only one choice for the termination clause. For the others, however, where inputs are known as `input1` and `input2` there is a choice of three possibilities:

```
(cond
  ((and (equal input1 nil) (equal input2 nil)) accumulant)
  ...)


(cond
  ((equal input1 nil) accumulant)
  ...)


(cond
  ((equal input2 nil) accumulant)
  ...)
```

We will find that, for each of the functions, skeptics will be able to invalidate one or more of the choices without having to consider any further specification of the program. **This is a very important point** and we use bold face to draw your attention to it. What are actually doing is discarding *whole classes* of possible functions with one single skeptic. In a more conventional problem solving system, without concurrent skeptics, many hundreds of programs might have to be generated to completion, each with the same bug.

The code that proposes these termination clauses is shown in figure 8.

Fig. 8. Code For Proposing Termination Clauses

```
(defunc propose-termination-clauses ()
  (cond
    ((and (= (length list-of-inputs) 2)
          (eq (<- (car list-of-inputs) 'typed-object) 'sequence))
     (let ((first (car list-of-inputs))
           (second (cadr list-of-inputs)))
       (assert (termination-clause ((equal →first →nill) →accumulant)))
       (assert (termination-clause ((equal →second →nill) →accumulant)))
       (when {(disproven-termination-clause ((equal →first →nill) →accumulant))
              (disproven-termination-clause ((equal →second →nill) →accumulant))}
         (assert (termination-clause
                   ((and (equal →first →nill) (equal →second →nill)) →accumulant))))))
    ((= (length list-of-inputs) 2)
     (let ((sequence (cadr list-of-inputs)))
       (assert (termination-clause ((equal →sequence →nill) →accumulant)))))
    (t (let ((sequence (car list-of-inputs)))
         (assert (termination-clause ((equal →sequence →nill) →accumulant)))))))
```

The function checks to see if there is only one input that is a sequence, and if so, asserts the one possible termination clause. If there are two inputs that are sequences it asserts for each one:

```
    (assert (termination-clause ((equal →input1 →nill) →accumulant)))
    (assert (termination-clause ((equal →input2 →nill) →accumulant)))
```

At some point, because of the running of the skeptic, it may be determined that one or both of them are `disproven-termination-clause`'s. If both of them are disproven, the sprite:

```
(when {(disproven-termination-clause ((equal →first →nill) →accumulant))
       (disproven-termination-clause ((equal →second →nill) →accumulant))}
  (assert (termination-clause
             ((and (equal →first →nill) (equal →second →nill)) →accumulant))))
```

will fire and the following assertion will be made:[†]

```
(assert
    (termination-clause
      ((and (equal →input1 →nill) (equal →input2 →nill) →accumulant))))
```

When a termination clause has been proposed two new activities are created, one whose function is to build a function based on it, and the other whose function is to show that it is not a possible termination clause. These are the skeptics and we will treat them first.

We must first address the question of how it is possible to demonstrate that a particular termination clause could not possibly be correct. Let us assume we wish to refute a termination clause of the form:

```
                  ((equal input1 nill) accumulant)
```

Let us assume that the iteration has just begun. Since we have not yet been through even one loop of the iteration, the `accumulant` must be equal to `nill`.[‡] Since we have just begun the iteration, if we can find input examples for which `input1` is `nill`, but the `accumulant` (which is equal to the output since this is the last clause of the iteration) is something other than `nill` we will have disproven its validity as a termination clause. Code for doing this is shown in figure 9. As we can see, it selects a viewpoint (bound to `vpt`) in which the input being tested for is `nill` (i.e. it is the `null-test`) and the other input contains one element (the `singleton-test`). We assert that the `output` is null. If we discover any contradictions in this viewpoint we will assert that the termination clause has been disproven. Now let us see how this will work in practice. There are four examples that take two

[†] I am not certain whether this is logically correct, that is, if the final termination clause requires the conjunction, whether the two individual conjuncts can always be proven invalid. It is certainly true for the examples presented here and others I have looked at. This problem could be avoided by giving the activity pursuing functions based on the conjunctive termination condition a small amount of processing power to begin with and increase it if and when the non-conjunctive termination conditions have been proven invalid.

[‡] The type of the accumulant is always the same as the type of the output. In all but one of the examples, `greatest` the output is a sequence.

Fig. 9. Code For Disproving Termination Clause

```
(defunc test-null-termination-clause (input clause)
  (let ((test-configuration (if (eq input (car list-of-inputs))
                                '(null-test singleton-test)
                                '(singleton-test null-test))))
    (within-viewpoint control-viewpoint
      (when {(viewpoint-for-test →test-configuration =vpt)}
        (let ((scratch-vpt (new-viewpoint  inherits-form vpt)))
          (within-viewpoint scratch-vpt
            (activate-knowledge
            (assert (equal →output →nill))
            (when {(contradiction)}
              (within-viewpoint control-viewpoint
                (assert (disproven-termination-clause →clause)))))))))))
```

sequences as arguments. We will examine each in turn.

5.4.1 Set-Difference

The correct termination condition for the `set-difference` function is `(equal input1 nil)`. We will show how the other possibility is disproven. The code in figure 9 is invoked to check the possibility of `(equal input2 nil)`. This will cause the creation of a viewpoint in which the following assertions will appear:

```
 ;These are inherited from the prototype viewpoint
(assert (equal →input2 →nill))
(assert (length →input1 1))
(assert (sequence-element →input1 →singleton-element1 1))

 ;The following additional assertion is made:
(assert (equal →output →nill))
```

If we now look at the sprites that define `setdifference` we find the following:

```
(when {(member =x →input1)}
  (when {(member →x →input2)}
    (assert (not-member →x →output)))
  (when {(not-member →x →input2)}
    (assert (member →x →output))))
```

This sprite will trigger with `x` bound to `singleton-element1`. The body of the sprite will then be evaluated with in this environment. In paricular the following sprite will be activated:

```
(when {(not-member →x →input2)}
    (assert (member →x →output)))
```

Since `input2` is known to have no elements, this sprite will immediately fire, and `(member singleton-element1 output)` will be asserted. A `(CONTRADICTION)` will then be asserted because it is known that `output` has no elements. Then the sprite watching for contradictions in this viewpoint shown in figure 9 will trigger causing the assertion:

```
    (disproven-termination-clause ((equal →input2 →nill) →accumulant))
```

to be generated. As we will see later, this assertion will cause the activity pursuing functions based on this termination clause to cease functioning, and the processing power redirected to the correct termination clause.

5.4.2 Union and Merge

Both of the functions Union and Merge have as their termination clause:

```
        ((and (equal input1 nil) (equal input2 nil)) accumulant)
```

For both of them, the individual termination clauses `((null input1) accumulant)` and `((null input2) accumulant)` are disproven in a similar way, and we will treat them together. Suppose we have the task of disproving `((null input2) accumulant)`. As happened above, we create a new viewpoint in which we find the following assertions:

```
                    (equal →input2 →nill)
                     (length →input1 1)
         (sequence-element →input1 →singleton-element1 1)
                    (equal →output →nill)
```

In both the definitions for `merge` and `union` we find the following sprite:

```
(when {(member =x →input1)}
       (assert (member →x →output)))
```

This sprite will trigger with `x` bound to `singleton-element` causing the following to be asserted:

```
             (assert (member →singleton-element1 →output))
```

A `(CONTRADICTION)` will be asserted. The clause under consideration will be marked a "`disproven-termination-clause`" as happened above.

Both of these functions are symmetric with respect to the arguments `input1` and `input2`. Demonstration that the termination clause `((null input1) accumulant)` will not work is handled similarly. When we have asserted both:

```
    (disproven-termination-clause ((equal →input1 →nill) →accumulant))
    (disproven-termination-clause ((equal →input2 →nill) →accumulant))
```

work attempting to create functions based on either of those termination clauses will cease. Additionally the sprite in figure 8 will trigger and we will assert:

```
(termination-clause (and (equal →input1 →nill) (equal →input2 →nill)))
```

This will initiate work attempting to thresh this out into a function as we will soon see.

5.4.3 Intersection

We listed the termination condition for this function to be `(equal input1 nil)`. In fact, `intersection` is completely symmetric with respect to its arguments; thus either termination clause will work, and neither can be disproven. What happens when we actually push this function through the system, is that *both* functions with respect to this symmetry are pursued in parallel. This function points out one defect in our system -- by translating the logical description of our program behavior into sprites we have lost the level of description at which it is possible to deduce that they are in fact symmetrical, and this knowledge used to `stifle` one of the two branches as being unnecessary. What this costs us in speed is only a factor of 2 here, but in a much more realistic program synthesis system these factors of 2 have a way of leading to an exponential decrease in efficiency with increasing complexity of the specifications.

5.5 Proposing Simple Constructors

Before we can continue with our discussion of the proposition, testing, and elaboration of iterative programs we must first discuss how it is possible for the synthesis system to propose simple constructions. For example, if we know that one of the inputs to a function, `input1` is bound to the list `'(a b c d)` and we know the output of the function, `output`, is the list `'(b c d)`, an obvious *proposal* for a simple function to produce the output is the function `(cdr input)`. How is this proposed? That is the subject of this section.

There is a function called `propose-possible-constructors` of one argument. When called, it activates a number of sprites that each look for different ways of constructing the function. We will examine the workings of these sprites one by one with examples. There is one predicate (assertion) that we must explain first, the `available` predicate. When we reason about how to construct an object, there are certain objects that are *available* to us for doing the construction. The inputs to our function, as well as certain constants, can be used by us to construct new objects, but certain objects, say the output, cannot be used. There are objects we wish to be able to reason about but are not objects we can use for construction; only ones that can be used have been asserted to be `available`.

The complete function is shown in figure 10; we will explain each of the sprites in its body in turn.

Fig. 10. Function To Propose Possible Constructors

```
(defunc propose-possible-constructors (object)
  ;This function takes one argument, OBJECT, and asserts possible primitive functions that could construct it.
  ; ----------
  ;If the object you are trying to construct is equal to an available object, then that object is a constructor for it.
  (find-equal-objects object)
  (when {(equal →object =x)
         (available →x)}
    (assert (possible-constructor →object →x)))
  ; ----------
  ;If you are trying to construct the first element of a known list, a constructor for it is the CAR of that list.
  (when {(sequence-element =list →object 1)
         (available →list)}
    (assert (possible-constructor →object (car →list))))
  ; ----------
  ;If the object you are trying to construct has a CAR of something that is constructible, and its CDR is
  ;equal to some available sequence, then you can construct it by cons'ing the first onto the second
  (when {(sequence-element →object =car 1)}
    (propose-possible-constructors car)
    (when {(possible-constructor →car =make-car)
           (length →object =n)
           (length =cdr →(- n 1))
           (available →cdr)
           (∀ m in →(list-of-integers from 1 to (- n 1))
                check {(sequence-element →object =el →(+ m 1))
                       (sequence-element →cdr →el →m)})}
      (assert (possible-constructor →object (cons →make-car →cdr)))))
  ; -----------
  ;If the object you are trying to construct is equal to all but the first element of another, then CDR is a
  ;possible constructor.
  (when {(length →object =n)
         (length =list →(+ n 1))
         (available →list)
         (∀ m in →(list-of-integers from 1 to n)
              check {(sequence-element →object =el →m)
                     (sequence-element →list →el →(+ m 1))})}
    (assert (possible-constructor →object (cdr →list)))))
```

5.5.1 Objects Equal To The One Your Are Trying To Construct

If we wish to return an object of a certain form, the simplest thing to do is find objects we already know about that have that form! This is what the first section of `propose-possible-constructors` tries to do. The code looks as follows:

```
(find-equal-objects object)
(when {(equal →object =x)
       (available →x)}
  (assert (possible-constructor →object →x)))
```

We first call the function `find-equal-objects` (described in section 5.5.5). This will activate sprites trying to find objects that are in fact equal (in the Lisp sense) to the one we are trying to synthesize, but not yet known or *asserted* to be `equal`. Whenever an object is found that is known to be equal to the argument, and that object is known to be available, we can assert that a possible constructor is that object by itself.

5.5.2 Synthesizing Car

If the object we are trying to construct is is the first element of some list, and that list is available, a possible constructor for the object is the `car` of the list. The code to accomplish this is:

```
(when {(sequence-element =list →object 1)
       (available →list)}
  (assert (possible-constructor →object (car →list))))
```

The first clause of the sprite looks for *any* item currently known about that has the target object in its first position. This item is then bound to the variable `list`. If `list` is `available`, `(car list)` is a possible constructor.

5.5.3 Synthesizing Cons

Suppose you want to construct a list. A way of doing this is to see if there is a way of constructing the first element of the list. If there is, see if there is a list which is equal to the CDR of the list you are trying to construct. If such a list exists, a possible constructor for the object we are trying to construct is the CONS of these two. This is shown below:

```
(when {(sequence-element →object =car 1)}
  (propose-possible-constructors car)
  (when {(possible-constructor →car =make-car)
         (length →object =n)
         (length =cdr →(- n 1))
         (available →cdr)
         (∀ m in →(list-of-integers from 1 to (- n 1))
              check   {(sequence-element →object =el →(+ m 1))
                       (sequence-element →cdr →el →m)}})}
    (assert (possible-constructor →object (cons →make-car →cdr)))))
```

The code first extracts the first element of the target function which is bound to `car`. It then tries to propose possible constructors for this object. If a possible constructor is located it becomes bound to the symbol `make-car`. We then look for all lists that have a length of 1 less than the length of the object we want to construct and are available. If such a list is found, it is bound to the variable `cdr`. The final clause essentially checks each element of the object we want to create and sees if that element occurs in a corresponding position of `cdr`. If this is true for *every* position of the list then we can use CONS to construct the target object.

5.5.4 Synthesizing Cdr

If there is an available object that is equal to the target object with something tacked onto the front, then we can construct the target object by taking the CDR of this other object. The following code checks for this:

```
(when {(length →object =n)
       (length =list →(+ n 1))
       (available →list)
       (∀ m in   →(list-of-integers from 1 to n)
          check  {(sequence-element →object =el →m)
                 (sequence-element →list →el →(+ m 1))}})}
  (assert (possible-constructor →object (cdr →list))))
```

The code checks for any lists known to be of length 1 greater than the target object. If it finds one (and it is available), we check to see if every element (bound to `el`) that is in the mth position of the target object is in the m+1st position of the list. If this is true for all objects, we can assert that taking CDR of the list is a possible constructor for it.

We have hardly exghausted the possibilities for primitive constructor functions and don't pretend to have a complete theory of such constructions. Nevertheless, the function for finding possible constructions is sufficient for all the examples we have looked at and has a good deal more generality.

5.5.5 How To Find Equal Objects

If we want to find objects equal to the a given object in the current viewpoint we call the following function:

```
(defunc find-equal-objects (object)
  (when {(value →object =n)
         (value =object2 →n)}
    (assert (equal →object →object2)))
  (when {(length →object =n)
         (length =object2 →n)
         (not-eq →object →object2)
         (∀ m in →(list-of-integers  from 1  to n)
              check   {(sequence-element →object =el →m)
                       (sequence-element →object2 →el →m)}})}
    (assert (equal →object →object2))))
```

Executing it has the effect of activating two sprites. The first sprite checks to see if the *value* of the object is known. The `value` of an object is only defined if the object is known to be a number and it is known which number it is. We then look for other objects known to have this same value. If any are found, we assert the two objects to be `equal`.

The second sprite triggers if an object is known to be a list of a known length. We check for other objects of the same length and make sure they are not the same exact object. (This is what the `not-eq` predicate does. We do this only for efficiency reasons; if they are the same exact object then they are already known to be equal.) We then let `m` range over all the integers from `1` to the length of the list. If something (bound to `el`) is in the same position in both lists, the two lists are asserted to be equal.

5.6 Pursuing The Function Definition

What actually happens when we first propose a termination clause is that the code in figure 11 is executed.

Fig. 11. After Finding A Termination Clause

```
(let ((activity-for-terminator (new-activity prefix 'term-clause-activity)))
  ;If we ever show this termination clause is invalid, stifle the activity.
  (within-viewpoint control-viewpoint
    (when {(disproven-termination-clause →clause)}
      (stifle activity-for-terminator)))
  (within-activity activity-for-terminator
    ;Here we create two sub-activities, one to pursue the termination clause and one to attempt to refute it.
    (let ((proponent-act (new-activity prefix 'terminator-proponent))
          (opponent-act (new-activity prefix 'terminator-opponent)))
      (support-in-ratios
          parent     activity-for-terminator
          activities (list opponent-act proponent-act)
          factors    '(2 1))
      (within-activity proponent-act
        (propose-possible-recursions clause))
      (within-activity opponent-act
        (test-plausibility-for-termination-clause clause)))))
```

As we can see, the code creates a new activity called `activity-for-terminator` which will pursue both the further development of the clause and its refutation. A sprite is then activated,

```
(when {(disproven-termination-clause →clause)}
  (stifle activity-for-terminator))
```

If we ever learn, via the techniques of section 5.4, that the termination clause cannot possibly work, the activity is stifled and its processing power is returned to its superior activity (which in this case is `the-root-activity`) for redistribution to other, more promising activities. After creating this activity, we create two new subactivities called the `opponent-act` and `proponent-act`. The first of these pursues the work described in section 5.4 attempting to disprove the validity of the clause. The second carries out the task of trying to flesh out the termination clause into a function. This is described now.

We now ask ourselves the following question: "What characterizes the simplest possible extension of the skeletal function we now have that *might* lead to a complete function?" This skeletal function is:

```
(defun function1 (input nil)
  (cond
    ((equal input nil) accumulant)
    ...))
```

The answer is simple: assume we have but one alternative clause, and then construct a plausible *accumulating* function. That is, we first look for functions of the form:

```
(defun function1 (input nil)
  (cond
    ((equal input nil) accumulant)
    (t (function1 (cdr input) accumulating-function))))
```

There is actually only one function that fits this format, `reverse`. We will show how `reverse` is synthesized, and then show how the other functions synthesized are further elaborations of this basic approach.

The best places to look for simple proposals for accumulating functions are traces of the function when run on simple examples. The example in which we have an empty input is a bit too simple (we already have a clause for that one). The next place to look is where we have a function that has one input. Our sprites that define the Reverse function tell us what form the output must take when the input is a singleton. They tell us that the output must be a list consisting of the same element. Thus, in the prototype viewpoint for the `singleton-test`, we will have derived this fact. Because we are iterating down the list `input`, the `singleton-test` reflects the stage of the iteration *immediately preceding the final null test* of the first clause of the function. This being the case, the accumulating function that we wish to propose should be able to construct the output out of the available objects in the viewpoint. Code for deriving possible accumulating functions is shown in figure 12. The function creates a new viewpoint that inherits from the `singleton-test` viewpoint. Because there is only one iteration required to take us from where we begin, to the final output, we assert that the `accumulant` is null (and also that it is available). We then try to construct objects that are equal to output and propose ways of constructing those objects out of ones that are available. If a constructor is found, we assert that it is

Fig. 12. Code To Propose Possible Accumulators

```
(defunc propose-accumulators ().
  (within-viewpoint control-viewpoint
    (when {(viewpoint-for-test (singleton-test) =v)}
      (let ((scratch-vpt  (new-viewpoint  parent v  prefix 'find-accum)))
        (within-viewpoint scratch-vpt
          (activate-knowledge)
          (assert (equal →accumulant →nill))
          (assert (available →accumulant))
          (when {(equal →output =is-output)}
            (propose-possible-constructors is-output)
            (when {(possible-constructor →is-output =make-output)}
              (assert (accumulating-function →make-output))))))))
```

---

an `accumulating-function`. In the case of the `Reverse` function, there are three possible accumulating functions that are produced. They are:

```
                    input
            (cons (car input) nil)
        (cons (car input) accumulant)
```

Now that we have proposed accumulating functions, we now (actually) have three proposition for complete functions. They are:

```
(defun Reverse1 (input accumulant)
  (cond
    ((equal input nil) accumulant)
    (t input)))


(defun Reverse1 (input accumulant)
  (cond
    ((equal input nil) accumulant)
    (t (cons (car input) nil))))


(defun Reverse1 (input accumulant)
  (cond
    ((equal input nil) accumulant)
    (t (cons (car input) accumulant))))
```

The last of these is the correct function definition. The first two are not. How they can be eliminated is the subject of section 5.8.

5.7 Conditional Introduction

The simple program schema used above will not work for anything but the `reverse` function. All the others have paired conditions. These paired conditions must be proposed in some manner. The technique we use is quite similar to the technique of *conditional introduction* used by Manna and Waldinger [41]. There is one assertional type we have not yet mentioned that is important in the introduction of conditionals. Whenever we ask a question about the equality of two objects, i.e. whenever we activate a sprite of the form:

```
(when {(equal →object1 →object2)}
  ...)
```

and it is not then known whether the objects are equal or not, we assert the following:†

```
                    (possibly-equal →object1 →object2)
```

There is one other relation that is treated in this manner, `less`. Whenever we create a sprite of the form:

```
(when {(less →object1 →object2)}
  ...)
```

and the relation between the two objects is not definitely known, we will assert:

```
                    (possibly-less →object1 →object2)
```

The sprites that will ask the questions about the equality or relative order of two objects that lead to the conditionals in the code are, of course, the ones in the specifications for the functions to be synthesized. For example, one of the sprites used in the specifications for the `intersection` function was:

```
(when {(member =x →input1)
       (member →x →input2)}
  (assert (member →x →output)))
```

Now suppose that we invoked this sprite in one of our prototypical viewpoints as defined in section 5.3.2.

† There are several ancillary remarks we can make with respect to `possibly-equal` assertions:
1. As will be made clear in chapter 7, the "internal" behavior of sprites and assertions is something that is programmable by the user. This feature is useful in the construction of the program synthesis system, but does not mean that equality sprites in all Ether-base systems must exhibit this behavior.
2. Also, as will be explained in that chapter, the cost of a `possibly-equal` assertion is almost nothing.
3. It may seem at first glance that the presence of `possibly-equal` assertions violates our principle of monotonicity. That is, if something is possibly equal then if we later learn that it is definitely equal or not equal we will have done something inconsistent. We will see by the way these assertions are used that a proper interpretation for a `possibly-equal` assertion is that "a sprite wants to know whether these two objects are equal." The truth of that statement does not diminish when it is learned that the two objects are either equal or not equal.

We will invoke it in the viewpoint in which each of `input1` and `input2` contains a single element. In this viewpoint we have not said anything more about the relationship between these two member elements (which we call `el1` and `el2` for purposes of discussion). The sprite above will fire with `x` bound to `el1`, when we then reach the sprite pattern:

```
(member →x →input2)
```

it will try to determine whether or not `el1` is equal to `el2` (the object it knows is a member of `input2`).† This information is not, of course, known and a `possibly-equal` assertion mentioning the two elements will be made. This will cause the system, as we will see, to introduce the conditional in the definition of the `intersection` function:

```
(defun intersection1 (input1 input2 accumulant)
  (cond
    ...
    ((member (car input1) input2)
     ...)
    ((not-member (car input1) input2)
     ...)))
```

The code that initiates all conditional introduction is shown below:

```
(foreach
  input
  list-of-inputs
  (when {(sequence-element →input =first 1)}
    (when {(possibly-equal →first =object)
           (available →object)}
      (propose-car-equal-tests input object test-viewpoint))
    (when {(possibly-less →first =object)
           (available →object)}
      (propose-less-tests input object test-viewpoint))))
```

This code is executed in one of the prototypical viewpoints in which the input(s), if they are both sequences, contain only one element. If one of the inputs is a number or an atom we use the prototypical viewpoint in which nothing is known about the properties of that particular element. The actions of the functions `consider-car-equal-tests` and `consider-less-tests` will be discussed in section 5.7.1. Section 5.7.2 considers the proposing of accumulating functions after the conditionals have been picked.

† That the possible triggering of a `member` sprite causes the activation of an `equal` sprite is quite important. This is an aspect of our "semantically meaningful" implementation of sprites and assertions that is the subject of chapter 7.

5.7.1 Proposing The Conditionals

The code for `propose-car-equal-tests` is shown in figure 13.

Fig. 13. Test Proposing Function for Equal First Element

```
(defunc consider-car-equal-tests (input object test-viewpoint)
  (within-viewpoint test-viewpoint
    (propose-possible-constructors object)
    (when {(sequence-element →input =first-element 1.)
           (possible-constructor →object =make-object)}
      (assert (possible-conditional (member →make-object →input)))
      (assert (possible-conditional (equal (car →input) →make-object)))
      (when {(equal →first-element →object)}
        (assert (disproven-conditional (member →make-object →input)))
        (assert (disproven-conditional (equal (car →input) →make-object))))
      (when {(equal →first-element →object)}
        (assert (disproven-conditional (member →make-object →input)))
        (assert (disproven-conditional (equal (car →input) →make-object))))
      (let ((vpt (new-viewpoint  prefix 'suggested-opponent-vpt))
            (el1 (new-object  instance-prefix 'opponent-vpt-object))
            (el2 (new-object  instance-prefix 'opponent-vpt-object)))
        (within-viewpoint vpt
          (assert (sequence-element →input →el1 1))
          (assert (sequence-element →input →el2 2))
          (assert (not-equal →el1 →object))
          (assert (equal →el2 →object)))
        (assert (suggested-opponent-for-conditional
                   (equal (car →input) →make-object) →vpt))))))
```

Remembering that the proposing of these conditionals happens within a viewpoint where each input (if a sequence) contains a single element. If we generate a `possibly-equal` assertion that refers to the first element of the input sequence, there are two possible kinds of tests that would make valid conditionals. The first kind of test is of the form:

```
(equal (car input) object)
```

The second possible test is of the form:

```
(member object input)
```

The code in figure 13 will propose both of these. First we have to find possible constructors for the object that the first element of the input is known to equal. This is done by:

```
(propose-possible-constructors object)
```

For each possible constructor of the `object` we make two assertions that there is a possible conditional:

```
 (assert (possible-conditional (member →make-object →input)))
(assert (possible-conditional (equal (car →input) →make-object)))
```

These assertions will trigger sprites that will create activities for the purpose of proposing the accumulation functions for each conditional branch as described in section 5.7.2. If, however, we eventually learn that the two objects that were asserted to be `possibly-equal` in fact are either `equal` or `not-equal`, the proposed conditionals are not possible (because one of the branches will never be followed). In this case we assert that the conditional is a `disproven-conditional`. There are sprites watching for these assertions that stifle the respective activities pursuing the conditionals.

Once conditionals are picked, as will be explained in section 5.7.2, we have complete the proposing of a possible function by picking the appropriate accumulating function. The proposed function is then subject to the refutation process. There is some aid we can give this refutation process by suggesting viewpoints that represent test cases likely to cause problems *if* the conditional picked was incorrect. The remainder of the code in figure 5.7.1 does precisely that. Suppose the `equal` test had been picked, what would be a good test case for possible completions? An alternative way to ask this question is: "What would be cases for which the `member` test will work but the equal test will not?" The simplest example of this is a case in which the first element is known to be not equal to th object in the test and the second element is known to be equal. As shown in figure 5.7.1, we create a new viewpoint (bound to `vpt`) and two objects (bound to `el1` and `el2`). We then place in that viewpoint the following assertions:

```
(assert (sequence-element →input →el1 1))
(assert (sequence-element →input →el2 2))
    (assert (not-equal →el1 →object))
      (assert (equal →el2 →object))
```

We then assert that this viewpoint is a useful one for refuting proposals in which the `equal` conditional was used by executing:

```
(assert (suggested-opponent-for-conditional
            (equal (car →input) →make-object) →vpt)))))
```

This can then be used in the final refutation process to test proposed functions.

5.7.2 Proposing the Accumulating Functions Within Conditionals

The picking of accumulating functions is done similarly to the way it was done in section 5.6. The example used in that section, `reverse`, did not contain any conditionals (other than the termination condition). If there are conditionals (as all other functions have) we must generate accumulating functions for each branch of the conditional. In the viewpoint in which we propose the accumulating functions we must *assume* the condition of the branch to be true and in that viewpoint propose the accumulating functions. We propose the accumulators by looking at the prototypical viewpoint that contains one element for each of the inputs.

We will use as an example the `intersection` function whose definition was given in section 5.4.3. The correct test for this function is:

```
(cond ...
  ((member (car input1) input2) accumulating-function1)
  ((not-member (car input1) input2) accumulating-function2))
```

Let us consider how the accumulating functions for each of the branches is proposed.

The code that does the proposition creates a subviewpoint of the prototypical viewpoint in which we assume the first of the conditions is true. In this viewpoint we activate the following sprite:

```
(when {(sequence-element →iterated-input =el 1)}
  (assert (member →el →member-list)))
```

where `iterated-input` is bound to `input1` and `member-list` is bound to `input2`. This will cause the first element of `input1` (call it "`el1`") to be known as a member of `input2`. Since `input2` is known to have a length of 1 (and we will call its single element "`el2`"), it will deduce that `el1` and `el2` are `equal`. The following sprite in the specifications of `intersection` then gets triggered:

```
(when {(member =x →input1)
       (member →x →input2)}
  (assert (member →x →output)))
```

Indicating that this element (that is in both `input1` and `input2`) is also in `output`. In this viewpoint we are now able to propose the accumulating function using the techniques of section 5.6.

We will now consider the other branch of the conditional. In the appropriate viewpoint we activate a sprite that represents the condition:

```
                    (not-member (car input1) input2)
```

The sprite is:

```
(when {(sequence-element →iterated-input =el 1)}
  (assert (not-member →el →member-list)))
```

with `iterated-input` bound to `input1` and `member-list` bound to `input2`. This will cause the two items that are known to be members of the two lists to become known to be not equal to one another. The other sprite that was used in the specifications of `intersection` was:

```
(when {(member =x →output)}
  (assert (member →x →input1))
  (assert (member →x →input2)))
```

Nothing more can be deduced of importance in this viewpoint. The `output` is in fact null, but we have not yet shown how this can be deduced. Since it is often the case that the `output` in these simple viewpoints *is* null, we have a special mechanism that usually allows us to deduce this fact if true. We execute the following code:

```
(let ((null-test-viewpoint
      (new-viewpoint parent test-viewpoint  prefix 'null-test-viewpoint))
      (obj (new-object  instance-prefix 'output-member-skolem)))
  (within-viewpoint null-test-viewpoint
    (assert (member →obj →output))
    (when {(contradiction)}
      (within-viewpoint test-viewpoint
       (assert (equal →output →nill))))))
```

What we do is create a new viewpoint, called `null-test-viewpoint`, that is a subviewpoint of the viewpoint in which we assumed that the given element was not a member of the list (bound to `test-viewpoint`). Within this viewpoint we make one additional assumption. *We assume the list has at least one element.* We create an object (bound to `obj`) of which we say absolutely nothing except that it is a member of the `output`. If we are able to deduce a contradiction in this viewpoint then we know that the `output` is null.† This is then asserted in the `test-viewpoint`. Now we will see how the contradiction is deduced. When we assert that the `output` has a member, we trigger the sprite:

```
(when {(member =x →output)}
  (assert (member →x →input1))
  (assert (member →x →input2)))
```

That asserts this same object in a member of both `input1` and `input2`. Both of these sequences are known to be of length 1 and have elements (which we call `el1` and `el2`). We will thus deduce that `obj` is equal to `el1` and that it is equal to `el2`. By transitivity we will deduce that `el1` is equal to `el2`. We have, however, previously deduce that `el1` and `el2` are not equal to one another. Therefore there is a contradiction and we learn that `output` is equal to `nill` in `test-viewpoint`. In this viewpoint we now have enough information to propose accumulating functions as was shown in section 5.6.

We will go through one more example of this process of proposing accumulating functions for conditional branches, the `delete` function. The two conditional branches of the `delete` function are:

```
(cond
  ...
  ((equal atom (car inlist))
   ...)
  ((not-equal atom (car inlist))
   ...))
```

† This is the same logical principle that is used in resolution theorem proving. However, our understanding of the semantics involved allows us to do the deduction much more effectively.

Since one of the inputs is an atom, and one a sequence, the prototype viewpoint we use for proposing lets the sequence (i.e. `inlist`) contain a single element (call it `el`) of which we have asserted nothing.

We create a subviewpoint of this prototypical viewpoint that represents the possibility of the clause:

```
(equal atom (car inlist))
```

being true. This is done by the sprite:

```
(when {(sequence-element →iterated-input =el 1)}
  (assert (equal →el →equal-element)))
```

where `iterated-input` is bound to `inlist` and `equal-element` is bound to `atom`. The sprites that define the `delete` function are shown on page page 63. If `inlist` contains a single element (as is the case here) and `atom` is equal to this element, then we should be able to deduce that the output is the null sequence. The way this is done, analogously with the previous example, is we create a subviewpoint of the current viewpoint in which we *assume* that the output has an element and watch for contradictions. As before, we will execute within this special viewpoint:

```
(assert (member →obj →output))
```

This will trigger one of the sprites used in the specifications for `delete`,

```
(when {(member =element →output)}
  (assert (member →element →inlist))
  (assert (not-equal →element →atom)))
```

Because `inlist` is known to have only one element in this viewpoint, which is equal to `atom`, the two assertions that get made will cause a contradiction and it will be asserted in the higher viewpoint that `output` is equal to `nill`.

Now we consider the other branch of the conditional:

```
(not-equal atom (car inlist))
```

In the viewpoint in which we propose accumulating functions, we execute the following sprite:

```
(when {(sequence-element →iterated-input =el 1)}
  (assert (not-equal →el →equal-element)))
```

with `iterated-input` bound to `inlist` and `equal-element` bound to `atom`. The following sprite in the specification of `delete` will then trigger,

```
(when {(member =element →inlist)
       (not-equal →element →atom)}
  (assert (member →element →output)))
```

establishing the necessary result for the proposition of accumulating functions.

5.8 Skeptics By Progressive Testing

By the time we have passed through the successive stages of the system thus far described:

1. Proposing and refuting termination conditions

2. Conditional introduction

3. Proposing accumulating functions

We have arrived at proposals for *complete functions.* There may be a number of proposals that survive previous refutation procedures. On the test cases they range from 3 (for `reverse`) to a few hundred for `union`. The process of discriminating between them is conceptually quite simple. We merely run them on test cases and throw out the ones that do not yield the expected results. In retrospect, the way I would have designed the system would have been to require the user to supply the system with a set of *criterial test cases.* If the program ran on each of these test cases successfully we will consider it a success. Programmers are quite good at picking sets of examples to use. Most of the "proposals" that make it this far are still sufficiently silly that they can be refuted quite easily.

I had instead elected to construct a system that tested the functions using progressively more complicated *prototypical test cases.* The idea would be to effectively execute the function on these prototypes. We would go through progressive loops of the function, each one of which would generate a new viewpoint that represented the state of the variables between loops. If the value of the output disagreed with that computed by the specifications, the function would be refuted and no more testing would be done on it. We would start first with the simplest viewpoints (those whose sequences had 0 or 1 elements in them) and then progress to more complicated viewpoints. Whenever a `possibly-equal` or `possibly-less` assertion was encountered (due to a conditional in the code itself) a bifurcation of viewpoints would have to occur; one path would consider the case as if the conditional were true, and one would consider it as if the conditional were false. The technical problems in coding this were so complex† that I was not able to complete coding it. I also feel, in retrospect, that efficiency considerations would have made this needlessly costly. This is particularly true because of the bifurcation that must occur each time we run into a conditional. Those proposed programs that are not

† And greatly complicated by a bug in the Lisp machine compiler that caused my programs to die abruptly.

eliminated on the very simplest cases will be tested at considerable computational expense. The only help to this process are the suggested test cases generated in section 5.7.1. For certain classes of the proposed functions they would give test cases that would quickly eliminate incorrect ones. Perhaps this methodology can be generalized to a theory of test case generation based on the specifications. I think future work on systems like these will probably, when testing whole functions, be better off using a select set of concrete examples.

5.9 Related Approaches To Program Synthesis

Program synthesis is a generic terms for techniques that simplify the intellectual tasks of creating a program by having the computer perform some of these tasks. A program synthesis system requires the programmer to specify the behavior of the desired program in some way. The form of the specification ranges from natural language to more formally definable specification languages. The common feature of all of these specification languages is that certain information that would have to be specified in any conventional language need not be given; the program synthesis system will be able to figure it out. This distinguishes program synthesis from other techniques for making computers easier to program such as debugging tools, integrated programming environments, structured programming, and modular languages. The computer, programming regimen, or programming language might make it easier to perform the intellectual tasks needed to design a program, yet cannot be considered to replace them.

Program synthesis projects vary widely in the classes of decisions that they wish to automate. This literature is voluminous and will not be reviewed here. There are, however, two classes of such systems that relate in interesting ways to the one developed in this work. These might be called the *deductive* and *inductive* approaches.†

The deductive approach starts with a *specification* in a formal language, usually first-order predicate calculus or a simple variant. From this specification it produces a program. Each starts with some sort of axiomatization for the target language, or inference engine based on logic, and constructs a program concurrently with a *proof* that the program produced satisfies the specifications. An early attempt at this was Green's QA3 system [16] that used resolution with unification to construct the program from an

† I find an interesting parallel between the two classical epistemological theories discussed in section 2.1 and the two approaches to program synthesis that I have called *inductive* and *deductive*. The inductive approach bears a resemblance to empiricism and the deductive approach to rationalism.

axiomatization of Lisp. Manna and Waldinger [41, 42] have a system that also generates simple Lisp programs from specifications. Their system understands allows specification in terms of sets and can deduce car-cdr type recursion from the set specifications. They have the ability to introduce other simple conditionals and auxiliary functions in some of their examples. Hansson and Tarnlund [18] introduce a system that axiomatizes more interesting data structures than simple sets and ends up with some interesting Prolog programs as a consequence. Like our own, none of these systems can work on more than a handful of examples.

Our approach begins with a similar specification, but, unlike the deductive approach, does not end up with a proof that the program is correct. At first sight this might seem to be a limitation of our approach. It could also, however, be an advantage. By not requiring the system to generate a proof *in addition* to the program we have lessened it burden and possibly increased its scope of applicability. None of these approaches has developed sufficient breadth that this issue can be decided. One argument in favor of our approach is that is certainly corresponds to the way people program. They have ideas, propose programs, and then test the programs on examples. Programmers will accept the program as being correct when they have tested it on sufficiently many examples that they are confident. Programmers rarely, in practice, "prove" their programs to be correct. This is our interpretation of falsificationist philosophy applied to the problem of engineering design (i.e. programming) which we believe is fundamentally correct.

There is a sense in which the techniques developed in this chapter can be gainfully used to augment the methodologies employed by the above systems. The problem solving structure of each of the above systems is naive. The descriptions concentrate mainly on the *deductions* involved in generating the target program and not on how they were picked. Each system generates numerous intermediate goals, many of which are invalid. There are some cases where the use of skeptics run in parallel with the pursuit of the goals could quickly eliminate what would otherwise be a costly branch in the (sequential) search.

The *inductive* approach does not begin with logical specifications, but with a set of example computations. Examples of these approaches include Hardy [19], Shaw and Swartout [57], and Summers [63]. Each of these takes a set of Lisp input-output pairs and produces a Lisp program as the result. Each of these systems is also capable of generating only a handful of programs. The programs, however, seem more contrived than either ours or ones produced by the deductive systems. Our programs (as well as the ones developed by the deductive school) are ones we would expect programmers to want to generate. The ones used by the inductive school aren't. An example from Hardy synthesizes a program from the following I/O pairs:

```
(A B C D E F) ==> (A B B C D D E F F)
```

The examples from the other works cited are of a similar sort. The reason these systems are weaker is, we believe, that they do not have any higher level specification of the desired behavior that they can reason about. The intuition that they are building on, that examples are important in the synthesis of programs, is a good one. In our system examples (or "generalized examples") play a role in both proposers and skeptics. However, the ability to reason about specifications gives us considerable power that systems just working from concrete examples cannot make use of.

While systems that can do unaided program synthesis are perhaps premature, a reasonable compromise might be a *programmer's apprentice*. The idea was first discussed by Hewitt and Smith [25]. An extensive implementation of such a system has been pursued by Rich [52], Waters [67], and Shrobe [58]. Many of the ideas presented here, both the concept of *proposers* and *skeptics* can potentially be of use in such a system, although I think people probably will probably make much better proposers in the near term. If the user specifies the *intent* of the code being generated. These intentions can be turned into sprites. When programs or partial programs are proposed, the specifications for these can also be expressed using sprites. The apprentice, in background mode, can then reason about the intentions and programs and discover bugs that can then be reported to the user.

Chapter VI How We Get Parallelism On Top Of Lisp

Previous chapters have dealt with programs in the Ether language without any concern for how these programs were implemented. This chapter goes into great detail on techniques of implementation. Ether is implemented on top of Lisp Machine Lisp, an upward compatible extension of MacLisp that differs from MacLisp (or the original Lisp 1.5, for that matter) little in essential detail. The nature of Ether is really quite different from Lisp in many of these details. It is thus of interest to study how the one language is grafted onto the other. I use the term "grafted onto" rather than "implemented in" because straight Lisp code is very definitely present in the "Ether code" examples we have seen. This has certain advantages as long as the merge can done smoothly, i.e. without violating Ether semantics. The principle advantage of doing this is that we can make use of the much more efficient and already implemented Lisp primitives. The Ether primitives `when` and `assert` are actual Lisp functions that get evaluated by the interpreter when they are run across.† There are two places in which the underlying Lisp implementation clearly violates Ether semantics and extra care is necessary; they are:

1. Lisp, is a fundamentally sequential language. There is a very definite order with which things get done. It makes no sense to talk about creating several activities and executing pieces of *Lisp* code in them. In fact, if this were done, one activity would be executed in its entirety before the others got a chance.‡

2. Lisp is a dynamically scoped language. Ether requires lexical scoping. If a variable is bound in a certain environment and a sprite is created in that environment (and contains the variable), we would like the variable to have the same binding when the sprite is *executed.* We have to go to considerable pains in the implementation to ensure the appropriate variables get bound to their values in the Lisp environment when the body of the sprite is executed.

We will address these two problems after explaining more of the details of the implementation. In fact, we will discover that with minor caveats about how Lisp code should be mixed with "straight" Ether code, the two problems seem to vanish. We can think about the code as being truly parallel and lexically

† More correctly, they are macros that expand into different code that gets executed. The nature of these macros is the subject of chapter 7.
‡ Here we are ignoring the Lisp machine *process* construct. It actually does have the capability to execute several Lisp programs in parallel. However, the time to switch from one process to another is non-negligible. The technique works fine when there are only a couple of processes runnable at any one time. It does not perform efficiently when dealing with more than a few processes.

binding almost all the time without running into bugs.

The discussion of the implementation is divided into two chapters that reflect different conceptual levels of the implementation. The lowest level, the subject of this chapter, discusses how parallelism is actually obtained through the message-passing sublanguage of Ether. This is where we explain how parallelism, lexical binding, activities, processing power, etc. are implemented. There are features here of interest not only to those concerned with problem solving languages, but also those interested in more general-purpose parallel language architectures.

Following this in chapter 7 we explain how assertions, sprites, and viewpoints are implemented in this parallel message-passing sublanguage. There are several novel features here not found in previous pattern-directed invocation languages. One aspect of these languages we have done away with is the *database*, or place in the implementation where all assertions and active sprites are stored. In fact, the information is stored in a very distributed manner and storage and retrieval is based on the *semantic* content rather than the *syntactic* form. This has many advantages which we will discuss.

6.1 Message Passing Languages

Message-passing languages come in two varieties. The more well-known of the two is typified by Smalltalk [29]; MIT readers may be more familiar with the Flavor system on the Lisp machine which has similar characteristics from the point of view of this discussion. The way programming is viewed in these languages is an inversion of the normal view. Normally we think of programs as recipes for *control* in which the language primitives manipulate data stored inside the machine. In message-passing languages we don't think of programs as manipulating data, rather the program is built out of *objects* that contain both data and procedures to manipulate the data. Rather than writing a monolithic program that acts on data, we *send an object a message* where the "data" is a parameter of the object and the message is a request to perform the operation. The reason this make a difference is because different objects can have different methods for responding to the same message and thus the same piece of program text can result in different (though appropriate) behavior when acting on a different set of objects. For example, if we wished to add 3 to some other number, we would send that number a `[+ 3]` message. How it performs this operation depends on the kind of number it is. If it was a conventional number the obvious would happen. We could also create, say, a complex number type that responded to this message in a different way. Many languages have a complex datatype† though we can imagine less standard kinds of "numbers" that we may wish to implement that a conventional language would be unlikely to provide. We might, for example, want to implement a number that serves the function of an *infinitesimal* (call it "ϵ") in non-standard analysis. We would define the "+" operation to, when asked to add ϵ with an integer (say "3") will return a new object (call it "$3+\epsilon$"). The object $3+\epsilon$ has procedures with it that specify what to do when it receives messages of various kinds. For example, its message handler for messages of the form "[> n]", meaning "are you greater than n" might look like:

```
if (n ≤ 3)  then true  else false
```

giving the correct response. Message passing has turned out to be a useful paradigm for implementing modular programs.

Although Smalltalk and similar languages provide tools that improve program modularity, the types of *control structures* that are possible are essentially no different than those available in Lisp.‡ The idea of message passing, however, leads us to a very different model of computation, one that is inherently parallel. This is known as the *actor* model of computation.

Once we understand the program as objects passing messages, and nothing more, we can get many more interesting control structures with no additional conceptual complexity. The key realization is that there is no longer a *process state* that enforces a sequence on program execution. Objects are sent messages; different objects being sent different messages concurrently process them concurrently. Hewitt [26] develops this concept by showing how standard control structures (those, say, involving recursion and a therefore a stack) can be redescribed as actor computations in which the stack has been replaced by actors that represent *continuations* [39]. Coroutines are just as easily had; two actors can be programmed to send each other messages back and forth. Parallelism falls naturally out of the actor model. What is referred to as *parallel forking* in other languages is will result in the actor model if one actor, after receiving a message, sends out more than one message. The notion of *process* is no longer well-defined; at any one time any number of messages can be in transit.

The principle actor language is known as Act1 [23, 24]. Act1 was developed primarily as a research tool

† Although few languages give you the tools to construct a complex datatype if it is not already provided by the system. This is the point.
‡ The language Simula [8], from which many ideas of Smalltalk derived, made use of a coroutine facility that cannot be done using a conventional stack-oriented control structure.

to explore the actor model, and in particular, techniques for dealing with mutable objects in a highly parallel programming environment. Act1 takes a radical approach to the problem of building a programming language based on actors -- *all* computation, down to a very microscopic level, is done by message passing. On a Lisp machine, not specifically designed for actors, there is an inherent overhead for running programs where all computation is done via message passing. In the current Ether we have taken a more pragmatic approach; message passing is used to a level of granularity necessary to ensure true parallel computation, but function calling of the conventional sort is made frequent use of. Another actor-based language, known as Atolia [7], allows actors to do computation internally by means other than message passing.

In order for there to be *effective* parallelism in Ether, which is implemented on top of Lisp, we have to ensure that no Lisp function has control of the interpreter for too long a time. As we will see in section 7.4, every time we create a sprite† or make an assertion we cause a break in the normal Lisp evaluation. The command that causes the actual work of the sprite or the assertion (which as we will also see in section 7.4 is the sending of a message) is saved for later execution. There is one other construct in the Ether language that causes a break in the normal Lisp evaluation. This is the `within-activity` command. When the Lisp evaluator comes across the following in an Ether program:

```
(within-activity act
  -- body --)
```

the Lisp commands contained in `-- body --` are not evaluated right away but are *queued* for evaluation under the auspicies of the activity `act`. A perusal of the code samples present in the text should convince the reader that Lisp evaluation cannot go very far without running into a `when`, `assert`, or `within-activity` construct. As an example we have taken the definition of `parallel-fork` for the cryptarithmetic problem solver shown in figure 2 and replaced all occurences of those three constructs with asterisks. This appears in figure 14. The forms that actually appears in the positions of the asterisks are simply commands to queue the appropriate code for later evaluation; the queuing operation requires very little computation. The code consists of a conditional with a simple predicate. The function `foreach` iterates over the list `alternatives` and for each one binds a few variables, executes `add-current-explorers` which merely adds an entry to a list, and queues three forms for later computation. This whole evaluation locks out the processor for so little time that effective

† Remember, as noted in section 3.5, that a sprite with several patterns is really a shorthand representation for several nested sprites.

Fig. 14. Shell of Lisp Code

```
(defunc parallel-fork (letter alternatives parent-viewpoint)
  (if (null alternatives)
      ;If there are no viable alternatives, the there is no consistent assignment possible.
      *****
      ;Otherwise, fork on each alternative
      (foreach
        digit
        alternatives
        (let ((v (new-viewpoint   parent parent-viewpoint))
              (a (new-activity    parent start-act)))
          (add-current-explorers v a)
          (within-viewpoint v
            *****
            *****)
          *****))))
```

parallelism is maintained. The same is true of the other code samples presented throughout this text.

6.2 Implementation of Activities

Some mechanism should be found to control the parallelism in actor languages if we are to write effectively controllable search programs. As we will soon see, the *activity* notion we have been making use of throughout this document is actually a mechanism for controlling the parallelism of message passing languages. The notion is perfectly general and can be integrated with any actor-based programming language.

One other mechanism has been explored in the literature for controlling the parallelism in actor programs, the *future* construct of Baker [1]. Futures are a very elegant mechanism for controlling programs *without side-effects*, but have serious deficiencies when programs with side-effects are considered.† Ether, although it is monotonic when viewed as a program that deals with sprites and assertions, compiles into a message-passing implementation that is very highly non-applicative. The activity mechanism was developed as a way a of controlling parallel programs that involve mutable objects. The decision that an activity is no longer essential to the whole computation involves an understanding of the semantics of the computation. The knowledge required to demonstrate a computation is no longer useful to the overall problem solving effort must be derived by the individual

† The deficiency involves the mechanism by which it is decided that an activity is no longer necessary and can be stopped. In an applicative language, if no continuation is currently waiting for the *result* of a computation, we are assured that the computation is no longer necessary and the activity halted. In non-applicative programs information can be passed between objects by other means than returning a value to the caller. Thus having no waiting continuations does not mean the activity will not pass information via a side-effect. If there is no waiting continuation for the result of an future, the "activity" it creates is stifled.

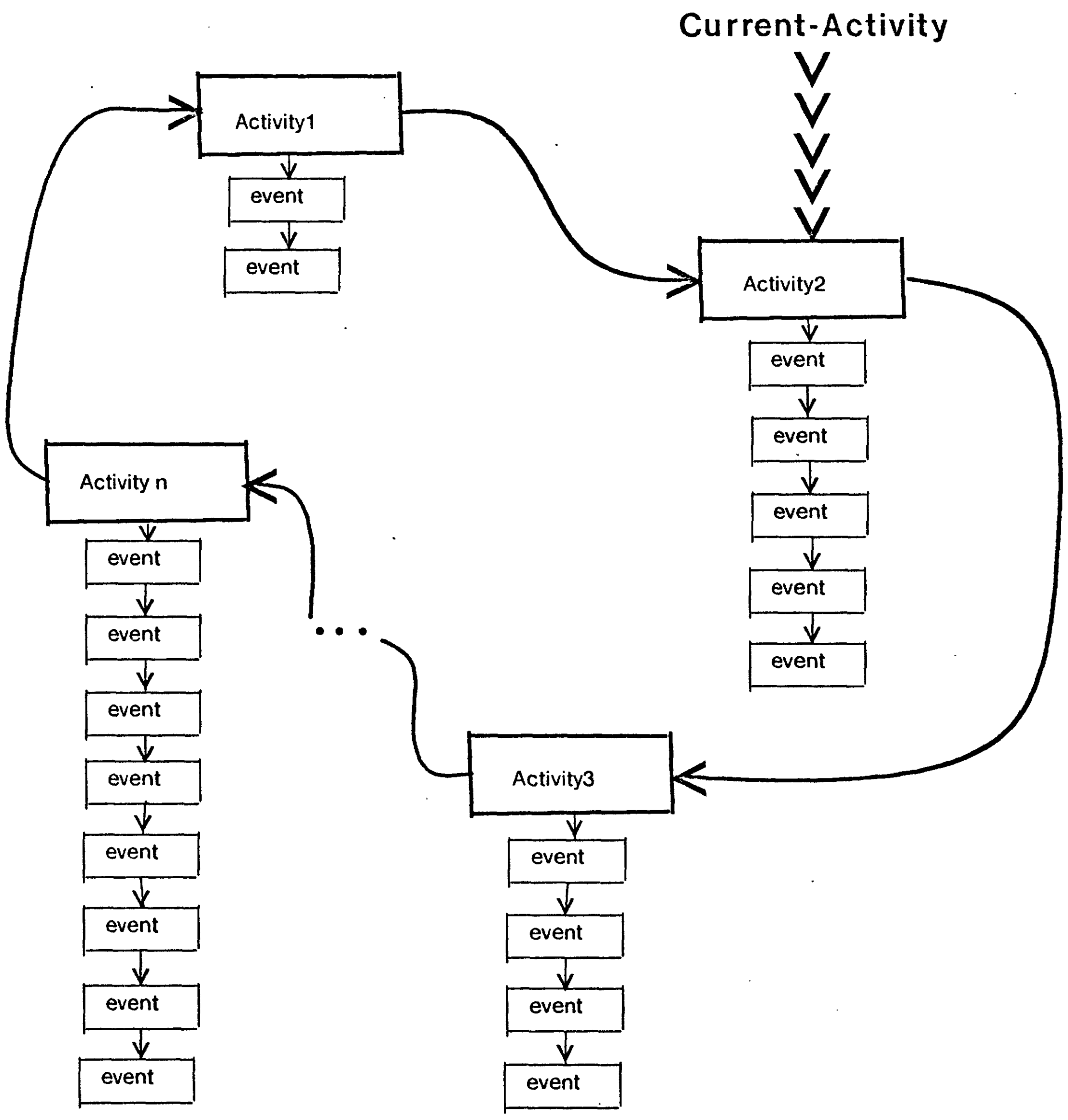
problem solver. Thus primitives to *control* the computation must be available to the problem solver. These are the processing power manipulation and `stifle` primitives we have been making use of.

To understand how the parallelism of Ether is implemented we must examine it momentarily with a very powerful magnifying glass. Ether is, of course, implemented on a single processor machine and if we look deep enough there is a well-defined *state* of the computation at any point in time. A schematic of what this state looks like is shown in figure 15. The whole computation is comprised of a number of *events*, each one of which is executed uninterruptably by the Lisp interpreter. The Ether interpreter is really a very simple function known as the *sanctum sanctorum.*† This function consists of a simple loop. On each iteration of the loop an event record is removed from the queue attached to the activity marked *current activity*. The event record contains code which is then executed. As we explained above, the time required to execute this code is short (in almost all cases not more than a few milliseconds). If any `when`'s, `assert`'s, or `within-activity`'s were present in the original code, commands would be in the code actually executed as part of the event that would put new events on the queue of some activity. In the case of `when`'s and `assert`'s the new event would be queued on this same activity's queue; in the case of a `within-activity`, the activity on which the event would be queued is the activity that was given as first argument to `within-activity`. Each time we execute an event we keep a tally of the total amount of time used to execute it. When a preset time limit is exceeded, the pointer marked *current activity* is advanced to the next activity in the ring and events are executed from its queue until its limit is exceeded. The choice of these limits is determined by the *processing power* assigned to the activity; how this number is computed is the subject of section 6.5. The execution of this loop -- through the ring of activities and queued events on each activity -- continues until no activity in the ring has any events on its queue; the interpreter then goes into a *wait state* awaiting further commands from the user.

The reader should not be confused by the fact that the ring of activities is flat. Each activity, save the root activity, has at least one parent activity and many have children activities; the activity graph has been flattened at this deepest level of the interpreter for reasons of efficiency. During the execution of an event, commands (such as `stifle` or `new-activity`) may be executed that will cause the structure of the ring of activities to be modified. Care has been taken in the implementation to prevent anomolous states of the ring of activities from occurring; we will explain the operation of these in section

† "Holy of Holies." This was the innermost chamber of the ancient temple in Jerusalem, accessible only to the High Priest and his assistants.

Fig. 15. State of an Ether Computation



6.6.

6.2.1 What An Event Record Looks Like

The definition of an event record is shown in figure 16.

Fig. 16. Definition Of An Event Record

```
(defstruct (event-record :named)
    next-event-record
    activation-body
    activation-closed-variables
    activation-closed-values)
```

Each event record is a structure containing four components. The first component implements the queue of events; it contains a pointer to the next event in the queue. If this slot is nil, then we have reached the end of the queue; the activity has quiesced. The code that is actually executed as part of the event is contained in the slot `activation-body`. It is a piece of Lisp code that is evaluated. This code is evaluated in an environment defined by the last two slots, a list of variables and a list of equal length of values to which these variables are bound. The variables are lambda-bound to these values and the body evaluated in this environment. The triplet `activation-body`, `activation-closed-variables`, and `activation-closed-values` really form a *closure*. They are represented in this more atomic form largely because the current implementation evolved from an earlier PDP-10 MacLisp implementation that did not have an explicit closure primitive.

6.2.2 What An Activity Looks Like

The definition of an activity is shown in figure 17. Slots marked with an asterisk, those relating to point sprites, will be explained in section 7.8.2 after the concept of a point sprite has been introduced. The slot `activity-symbol` contains the *print name* of the activity. If the object ever has to be printed (as often happens when debugging) it will print as the name in this slot. The print name is told to it via the arguments *name* or *prefix*† to the function `new-activity`. The activity ring depicted in figure 15 is

† We use the *name* argument when it is expected that the call to `new-activity` will only get evaluated once. The argument *prefix* is used when it is possible the code will be evaluated more than once. In this case a unique digit is appended onto the symbol passed as an argument. The ability to give activities names that relate to their purpose is invaluable in debugging.

Fig. 17. Definition of an Activity

```
(defstruct (activity :named)
     activity-symbol
     next-activity-in-ring
     parents
     children
     stifled
  *  stifled-point-sprites
     quiescent
  *  quiescent-point-sprites
     front-of-queue
     end-of-queue
     total-time-used
     total-time-should-have-used
     relative-processing-power-self
     subactivities-processing-power
     absolute-processing-power-total)
```

implemented by the slot `next-activity-in-ring`; each activity points to the next one in the ring. The slots `parents` and `children` implement the *subactivity* relation. The parents are all those activities higher than it in the activity tree; the children are those lower in the tree. If the activity has been *stifled* then the slot `stifled` will contain T; if the activity has ever *quiesced*† the slot `quiescent` will contain T. The queue of pending events is implemented with the two slots `front-of-queue` and `end-of-queue`. Events to be executed are pulled off `front-of-queue`. When a new event is to be added to the queue for the activity, the `next-event-record` slot of the event pointed to by `end-of-queue` is set to the new event record and the `end-of-queue` slot is then changed to the new event record.

The last five slots, `total-time-used`, `total-time-should-have-used`, `relative-processing-power-self`, `subactivities-processing-power`, and `absolute-processing-power-total` have to do with the way processing power is notated and implemented. This is the subject of the section 6.5.

† What this means in low level terms is that the activity has processed at least one event, but currently has no events to process. This ensures that there will be no race conditions. For the kinds of activities for which it makes sense to talk about quiescence, the only activity that can add a new event to the queue is the activity itself.

6.3 Lexical Environment Management

As mentioned in the introduction to this chapter, Lisp is a dynamically scoped language while Ether is not. We have already discussed in section 6.2.1 that each event record carries along with it an environment that gets instantiated before the code contained in the event record is executed. This, however, begs the question of just how the appropriate variable-value pairs got placed in the environment. That is the subject of this section.

Part of the answer to this question is that the implementation maintains two dynamically-scoped special variables that at all times *represent* the current lexical environment. These variables are called `current-closed-variables` and `current-closed-values`. They are, at all times, of equal length and elements in corresponding positions represent bindings of variables to values in the lexical environment. The presence of these variables is completely invisible to code written by the user. Of course, at all times the bindings represented by the current values of these two variables are in force in Lisp's dynamic environment. The code that queues activation records picks these two lists out of the dynamic environment and puts pointers to them in the appropriate slots of the event record. When the event record is executed, the two variables are bound to the values in these slots and the variable-value bindings are instantiated in the normal Lisp environment.

A slightly modified form of the Lisp binding mechanism `let` is used in Ether. In the code that was actually run, a function by a different name: `slet` is used. The function `slet`, in addition to binding the variables in its argument in the normal Lisp environment, also adds the bindings to `current-closed-variables` and `current-closed-values`. The `let` function isn't the only function that creates Lisp bindings. We have not written "Ether versions" of the others, such as `defun`, `lambda`, and `do`. If we used one of these binding mechanisms without taking some extra care we would be in a lot of trouble. Suppose, for example, we define the function `foo` as follows:

```
(defun foo (x a)
  (within-activity a
    (print x)))
```

`Foo` can is passed two arguments, the first can be anything, and the second must be an activity. What we would *expect* to happen is that the first argument that was passed to `foo` will be printed (assuming the second argument is an activity with processing power). This is not what would actually happen, however. The reason is that the call to `within-activity` causes its body (which is the form `(print x)`) to be placed in a new event record that is placed on the queue associated with the activity bound to a. The code that does the queuing places the currently known lexical environment (represented by those two variables) in the event record. However, the variable x was *not* bound by slet or any other Ether function and thus will not be part of the known lexical environment. The end result of this is that when (print x)) is evaluated, the variable x will either be unbound or have the wrong binding. The actual code we would have to write to get the proper behavior out of this is:

```
(defun foo (x a)
  (slet ((x x))
    (within-activity a
      (print x))))
```

We have to bind x to itself, a seemingly useless operation. This aspect of Ether will be further commented on in section 8.4.3. In the presentation of the code in the examples throughout this work we have replaced slet with let and eliminated all bindings of variables to themselves. This is the only place where the code has been "doctored" for presentation.

Sprites also introduce new bindings into the lexical environment. Just how this happens will be discussed in chapter 7.

6.4 Sending Messages

In fact, most events during the normal running of an Ether program consist of *messages* being sent to *objects*, rather than just the small chunk of code plus environment exemplified in the previous section. Almost all when and assert constructs get turned into message transmissions. The exact natures of the objects and the messages they can receive will be explained in chapter 7, however the basic mechanism will be explained in this section. The object-oriented sublanguage of Ether is built on top of the *flavor* system of the Lisp machine. We do not, actually, make use of any of the distinctive features of flavors, those that distinguish it from Smalltalk [29] or Simula [8]; they were merely available. The Ether send primitive takes three arguments as depicted:

(send *object message-type rest-of-message activity*)

The first argument, *object*, must by a conventional flavor-type object capable of receiving messages. The second argument, *message-type*, is a symbol. For each kind of message type, and each class to which the object can belong, there must be a particular method to handle the message. These message handlers can take arguments and these arguments are package in the third argument to send, *rest-of-message*. The last argument is the activity through which the event representing the message being processed by the object is is associated.

The `send` primitive causes a new event record to be added to the end of the queue of the activity argument. The body of this event record consists of the necessary Lisp commands to have the message with arguments processed by message handler associated with the object.

6.5 Implementation of Processing Power

In our example systems we have presented programs that make use of processing power. In this section we will review how the writer of an Ether program sees processing power and then discuss how it is actually implemented. We will only discuss the handling of processing power for *fixed* collections of activities. In section 6.6 we describe the implementation of commands (e.g. `stifle`) that *modify* the activity graph. The modification of processing power in these situations is a bit more complicated and requires the following as a prerequisite to its understanding.

Our design decisions concerning the implementation of processing power are based on one important premise: *Changes in processing power allocations to activities are a very infrequent event in comparison with the processing of events on activities' queues.*

The implementation of processing power presented in this section has many desirable characteristics. Among them:

1. The amount of overhead for ensuring that the processing power allocations are abided by is *very* small during normal execution. It amounts to one multiply plus one add every time the current activity is advanced, and a clock read, one add, and one comparison every time an event is executed.

2. The implementation ensures that over *any* period of time the ratio of actual clock time two activities get will be in proportion to their processing power allocations with an error that is at most the time of one event plus a small constant.†

3. Property 2 remains true even for activities with arbitrarily small amounts of processing power.

4. Property 2 remains true even for activities that have quiesced for a while (i.e. had nothing on their queues) but then received new event records.

† The constant is equal to the time it takes to do one complete cycle of the ring of activities. It is specifiable by the implementer. The current value used is 5 seconds.

5. The time required to readjust processing power allocations for an activity is proportional to the size of its own subactivity tree. This characteristic is desirable because modifications of resource allocations high up in the tree should happen much less frequently than farther down in the tree.

6.5.1 Relative And Absolute Processing Power

There are actually two different kinds of processing power that the system knows about. One of them, *relative processing power*, is the kind the user actually manipulates. The other, *absolute processing power*, is computed by the system for its own internal use. Initially, before any computation occurs, there is only one activity present, `the-root-activity`. When new activities are created within this activity, they become children of the root activity and a decision is made as to how much of the processing power assigned to `the-root-activity` is transferred to the new activities. Similarly, if new activities are created within the auspicies of any other activity, processing power must be transferred to them for any computation to happen within them.

Relative processing power expresses the way each activity divides the processing power assigned to it amongst itself and its children. At all times the total of the relative processing power kept for itself, and the amounts given to its children activities add up to 1. For example, if activity A1 had two children activities A2 and A3 and had relative processing power allocations of:

self-power: .33333, *A2 power:* .33333, *A3 power:* .33333

then each of the activities, A1, A2, and A3 would run at precisely the same rate. If the allocations were instead:

self-power: .2, *A2 power:* .4, *A3 power:* .4

each of the children activities, A2 and A3, would run at the same rate which would be twice the rate of the parent activity. If the allocations were instead:

self-power: 0, *A2 power:* .75, *A3 power:* .25

the parent activity would not get to execute at all, and the subactivity A2 would run three times as fast as the subactivity A3. It is not uncommon for a parent activity to have a self-power of 0. This configuration is desirable when A1 represents some particular goal for which there might be two subgoals. After establishing its subgoals there may be nothing more for it to do, so its processing power can all be given to its subactivities (which represent different approaches to accomplishing the goal).

A convenient function is supplied to allow the user to specify changes in relative processing power allocations. The function is called `support-in-ratios`. It takes five arguments, three required and two optional. A call to the function that would result in the final example mentioned above would be:

```
(support-in-ratios
  parent   A1
  children (list A2 A3)
  ratios   '(3 1))
```

would result in A2 being assigned three times as much processing power as A3, leaving A2 and A3 with relative processing powers of .75 and .25 respectively. An additional argument, *self-factor* can be specified that expresses the factor that should be maintained for use by the parent activity; *self-factor* defaults to 0. To obtain the previous example (in which A1, A2, and A3 had relative processing power allocations of .2, .4, and .4 respectively) we could have executed:

```
(support-in-ratios
  parent   A1
  children (list A2 A3)
  ratios   '(2 2)
  self-factor 1)
```

One other optional argument can be supplied: *default-factor*. This expresses the amount of processing power reserved for those activities not explicitly mentioned on the *children* list. The default for this argument is also 0.

Relative processing power indicates the proportions by which each activity divides processing power among itself and its subactivities. It does not indicate how much processing power an activity gets *relative to the entire system*, a very important number when we wish to compute a time quantum for scheduling each activity. This is the meaning of *absolute processing power*. Absolute processing power is computed from the relative processing power allocations. It is a fraction that represents the proportion of the total system execution time that it receives. The absolute processing power allocation received by an activity can be computed by multiplying all the relative processing power allocations along the chain starting with the root activity. At any one time the total of the absolute processing power allocations for all activities in the system add to 1. This is ensured by the implementation and cannot in any way be violated by the user program.

6.5.2 The Scheduling Algorithm

There is a constant called the *cycle time*, maintained internally by the system, that represents the total amount of time it takes for the scheduler to cycle through the ring of activities once. The behavior of a program is insensitive to the value of this constant within a large range. The cycle time should be small with respect to the total runtime of the program; this will ensure effective parallelism. If the cycle time is very small, it will occur additional overhead in the scheduling algorithm. The cycle time for all runs discussed in this work was 5 seconds.

The relevant fields that define the processing power allocations of an activity are to be found in figure 17. The field `relative-processing-power-self` contains the fraction of relative processing power that the activity maintains for its own use. The field `subactivities-processing-power` is a list of fractions whose length is equal to the list of children of the activity. The sum of all of these fractions is, of course, 1. The field `absolute-processing-power-total` represents the fraction of total system processing power allocated to *this activity and its children.* Thus, to compute the quantity of absolute processing power reserved for the activity's own use we multiply this number by the fraction `relative-processing-power-self`.

Whenever relative processing power allocations are changed, the absolute processing power allocations are at the same time modified to reflect this change. Thus, at all times they are consistent.

The scheduling algorithm is the following:

1. The *current-activity* pointer is advanced to the next activity in the ring.

2. The value in the field `total-time-should-have-used` (a number representing time in microseconds) is replaced by the following:

```
total-time-should-have-used + (cycle-time * relative-processing-power-self)
```

3. If `total-time-should-have-used < total-time-used`, go to step 1.

4. If there are no events on the event queue, go to step 1.

5. Else, execute an event from the event queue, keeping track of the amount of time it took to execute. Add this time to the slot `total-time-used`. Go to step 3.

The actual algorithm is actually a bit more complicated than the above (but not much). It also ensures that when the entire system quiesces (has nothing to do) this fact will be detected and the system will enter a "wait state", awaiting further activity from the user.

The above algorithm has several features worth noting:

1. The amount of overhead per event execution is quite small. In a microcoded implementation, the total overhead for event execution would be similar to the overhead to do a function call.

2. If an activity has a very small amount of absolute processing power, sufficiently little that it cannot support even one event for each cycle through the activity ring, the amount of time in the slot `total-time-should-have-used` will slowly accumulate. It may take several cycles for this value to exceed the value in `total-time-used`. Thus such activities may only execute an event an average of once every n cycles where n can be of any size. Yet the overall effect is to keep the amounts of time used by all activities in acordance with the processing power allocations to within the grain size of the events.

3. If an activity has no events to execute, the time allocated to it will slowly accumulate so that when it finally does have something to do, it will "make up for lost time." Again the over all effect is to keep time usage by activities in accordance with the processing power allocations.

6.5.3 Changing Processing Power Allocations

Whenever `support-in-ratios` is called, the slots `subactivities-processing-power` and `relative-processing-power-self` are adjusted so that they are in correct proportions to the factors given as arguments and are normalized so that they add up to 1. After these *relative* processing power allocations are changed, the *absolute* processing power allocations for each of the children activities must be modified.† An extremely simple recursive function accomplishes this for us. The function is shown in figure 18. The function takes two arguments, an *activity* whose processing power is to be adjusted, and an *amount* that it is to be adjusted by. The amount is a difference, so it is positive if the amount is to be increased and negative if the amount is to be decreased. The function first adjusts its own absolute processing power by executing:

† The relative processing power allocations of no other activities have to be modified. It is a purely local change.

Fig. 18. Code to Modify Absolute Processing Power Allocations

```
(defunc adjust-absolute-processing-power (activity amount)
  ;Recursively makes the processing power adjustment mentioned.  Positive means it gets more power.
  (struct+ (absolute-processing-power-total activity) amount)
  (let ((total-ppr-children
          (sumlist (subactivities-processing-power activity))))
    (foreach
      (subactivity relative-ppr)
        (children activity)
        (subactivities-processing-power activity)
      (adjust-absolute-processing-power
        subactivity
        (* amount (// relative-ppr total-ppr-children))))))
```

```
(struct+ (absolute-processing-power-total activity) amount)
```

It then computes the proportion of the absolute processing power change that is to be distributed to subactivities (bound to the variable `total-ppr-children`) and recursively distributes this change to the subactivities. When this procedure completes execution, the total absolute processing power of all activities will sum to 1.

6.6 Modifying The Activity Structure

There are two primitives that actually modify the structure of the ring of activities. These functions, and their implementation, are described in this section.

6.6.1 Creating New Activities

New activities are created by executing the function `new-activity`. Here we explain precisely what happens when this function is executed. The function creates a new activity structure (as depicted in figure 17) with the `parents` slot being filled by an explicit *parent* argument to the function or, by default, whichever activity happens to be currently executing. The `children` slot of the current activity is augmented by the addition of this newly created activity. In the current implementation, processing power allocations are modified at that time so that all children have equal amount of processing power. The creation of an activity is typically followed by a form that rearranges processing power in accordance with its needs.† After the activity is created processing power allocations for all affected activities are

† Future versions of Ether should have more intelligent ways of redistributing processing power within `new-activity` based on certain sterotypical patterns of doing so.

recomputed.

The new activity is inserted in the ring of activities *behind* the activity that is currently executing. This is necessary to avoid an unfair scheduling situation that would otherwise be possible. No matter how little processing power an activity has been given, it will be able to execute at least one event. Suppose there were a program that (due to, say, an infinite chain of goals) it were to create another activity during its first event. If this activity were to do the same thing *ad infinitum*, the system would never get on to execute the next activity in the ring. By placing newly created activities *behind* the current activity in the ring, this deadlock situation is avoided.

6.6.2 Stifling Activities

When an activity is stifled, all subactivities of it are recursively stifled. (The one exception to this statement involves subactivities introduced by the goal mechanism described in section 6.7.1). The `stifled` field of the activity object in figure 17 is set to `T` and the activity is spliced out of the ring of activities, i.e. the `next-activity-in-ring` field of the previous activity in the ring is set to the `next-activity-in-ring` of the activity being stifled. Thus events in the activity will no longer be available for execution. When an activity is stifled, relative processing power allocations of the remaining unstifled children activites are increased so that they remain in the same relative proportions and sum to 1. Absolute processing power allocations are then modified as described in section 6.5.3.

6.7 Other Mechanisms

There are two other mechanisms that relate to the execution of activities, the implementation of which needs to be explained.

6.7.1 The Goal Mechanism

The specially supported goal mechanism was introduced in section 3.3 and reasons why it is convenient are given in section 3.6. Here we discuss the specifics of the implementation.

The system maintains a table of all goals that have been established indexed first under the type of goal (e.g. `equal`) and then under the arguments given to it. In this table is the *internal activity* that is associated with the goal. When the user executes the code:

```
(goal (foo args) act)
```

the system checks to see if an entry in the table already exists for the specific set of arguments *args*. If so, it adds the activity bound to `act` to the list of parent activities of the internal activity, then the appropriate code to readjust absolute processing power allocations is called. Thus, any processing power that was given to `act` will be transferred to the internal activity under which all work on the goal happens.

If there is not currently an entry in the table for this particular goal, the following is done:

1. An activity (to become the internal activity) is created. With `act` as its parent activity.

2. The `defgoal` method for `foo` methods is then processed. If the method were defined as follows:

```
(defgoal foo (arg1 ... argn) act2
   -- body -- )
```

Then `act2` is bound to the newly created internal activity, the variables `arg1 ... argn` are bound to the arguments of the call to `goal`, and the body of the method is queued for processing under the internal activity.

3. An entry is created in the table for the call for the goal listing the internal activity created.

Thus only one activity (the internally created activity) will ever be working on the goal. At any time the activities that were used in the establishment of the goal can have their processing power changed and this change will get reflected in the internal activity. At all times this activity has processing power equal to the sum of the calling activities. The `stifle` primitive treats these internal activities differently in two respects:

1. When one of the user-created activities is stifled, the activity is removed from the list of parents of the internal activity. If every user-created activity associated with the goal is stifled, the internal activity is *not* stifled. The processing power assigned to it is automatically lowered to zero because it has no parents, yet it stays potentially executable. Thus, if some other part of the system later becomes interested in the particular goal, it can execute the `goal` command with a new activity that has processing power. The activity would then continue where it had left off.

2. If the internal activity is stifled, *every one of its parent activities is automatically stifled.* The internal activity can only be stifled from *within* the `defgoal`. The only reason that it might be stifled is if

either it is demonstrated that the goal is unattainable or that it has already be attained. In this case the activities are no longer necessary. By stifling them, the processing power that had been allocated to them is automatically redistributed among the unstifled activities.

6.7.2 Continuously-Execute

In at least one place, we have made use of the primitive `continuously-execute`. In the cryptarithmetic problem solver we created a *manager activity* that ran continuously in the background, monitoring relative progress of activities, and modifying processing power appropriately. We executed the form:

```
(continuously-execute (allocation-strategy))
```

This is implemented by placing on the queue of the respective activity essentially the following code:

```
(progn
  (allocation-strategy)
  (continuously-execute (allocation-strategy)))
```

Thus the function `allocation-strategy` will get run, and afterwards the same form will be placed on the end of the queue, enabling the function to get run again and again. For the reasons discussed in section 6.5, regardless of how long the function `allocation-strategy` takes to run each time, the percentage of resources allocated to the activity it is in will asymptoctically approach its processing power allocation and have a maximum error equal to the length of time the function takes to run plus a small implementation-specified constant.

Chapter VII The Implementation of Assertions and Sprites

We have made much use of code involving *assertions* and *sprites* in the examples but have not yet discussed what *really* happens when we execute the `assert` or the `when` construct. This chapter discusses these issues in depth. The implementation technique we have found is quite novel; it differs in substantive ways from other languages that contain constructs analogous to sprites and assertions. These differences have important implications for both the efficiency, power, and distributability† of these languages. As we will see, the implementation "inverts" many of the concepts that we have been using. The user of sprites and assertions "thinks" of the hierarchy of constructs in the language as the graph in figure 19a suggests. We normally think of creating an activity and activating sprites in it. We think of assertions as being placed in viewpoints and sprites as watching for assertions in viewpoints. Furthermore, assertions and sprite patterns contain objects. The implementer of sprites and assertions, as we will see, views the hierarchy of concepts in a manner suggested by figure 19b.

7.1 A Review of the Properties of Sprites and Assertions

Here we briefly review the properties that sprites and assertions must exhibit and then go on to describe the implementation.

<u>Monotonicity</u> Once an assertion has been made it cannot be erased. Any sprite that is capable of matching the assertion that is created at any future time (as long as it is in an activity with processing power) will be triggered.

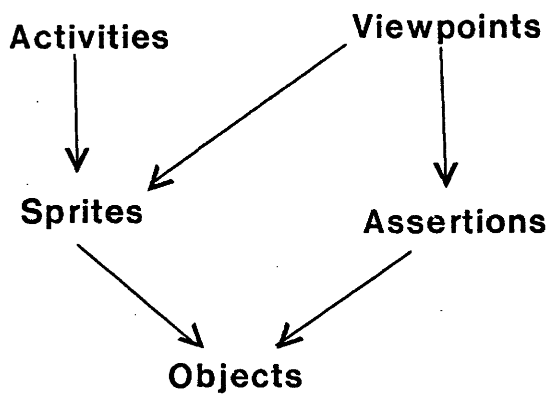
<u>Commutativity</u> When there is a sprite created, and an assertion that matches the sprite, the order of creation of the assertion and the sprite is immaterial.

<u>Viewpoints</u> Every assertion is done in the context of some viewpoint. The assertion is accessible to sprites in that viewpoint and in all viewpoints that inherit from that viewpoint. In the current system, all the parents of a viewpoint must be declared at the time of its creation; although, as we will see, trivial additions will make it possible to add new parent viewpoints at a later time.

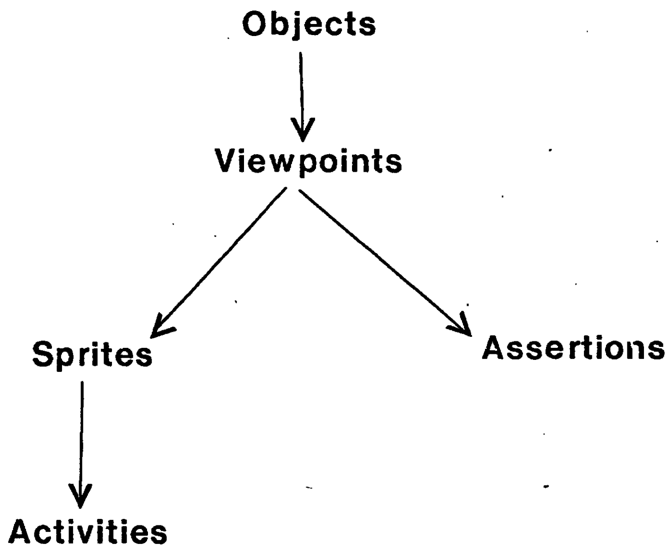
† That is, the ability to get the code to run on multiple processors. Most implementations of such languages make use of a *database* that can act as a bottleneck.

Fig. 19. Hierarchy of Concepts

(A) How the user of sprites and assertions sees the world:



(B) How the implementer of sprites and assertions sees the world:



Activities All sprites are created within some activity and the work required to actually effect the execution of the sprite must be turned into events executed by that activity.

7.2 Virtual Collections of Assertions

Previous languages of this form, ones with assertional capabilities and data-driven procedures (reviewed in section 3.6) treat the assertions and sprite patterns as *uninterpreted lexical forms.* The *semantic content* of the assertions is not in any way understood by the mechanisms that store and retrieve the information. We will shortly argue that by understanding something about the meaning of the assertions, ways of encoding them can be found that are much more satisfactory from several points of view. The concepts involved were first proposed in an earlier paper by myself [33].

There are various clever encoding schemes that make it possible to check for the presence of an assertion (or sprite pattern) in the database in time which is less than linear with the size of the database. All languages of this form have some sort of *discrimination net* that allows the retrieval mechanism to dispatch off of pieces of the assertion or sprite pattern and use a kind of "divide and conquer" approach to avoid searching most of the database most of the time. The most intricate scheme described in the literature is the one used by QA4 and is well documented in [54].

The bulk of this chapter describes the implementation of assertions and sprites for the program synthesis system. The description for the cryptarithmetic system, though similar, is much simpler and will be deferred to section 7.7.

7.2.1 The Basic Idea

The skeletal idea of virtual collections of assertions is quite simple. The primitives `assert` and `when` are not functions defined to interact with any kind of database. Instead, they are *macros*† that expand into code that is guaranteed (by the designer of this code) to have the *effect* of the assertions or sprites. For different *classes* of assertions, e.g. `equal` assertions or `member` assertions, we will replace the `assert` or `when` with very different pieces of code. Thus, the assertions do not *actually* exist as lexical items, nevertheless users of the `when` and `assert` constructs can write code as if they do exist. In this

† *Macro* is a generic term for code that replaces other code.

sense, the replacement code represents a *virtual collection of assertions.* Depending on the domain of the problem solver being designed, there are any number of ways a virtual collection of assertions can be implemented. Both the program synthesis and cryptarithmetic programs use very similar techniques that will be the subject of most of this chapter. Section 7.9.7 describes a very different kind of virtual collection of assertions to give some additional breadth to the concept.

7.2.2 How The Replacements Are Made

There is really *another* level of pattern-directed invocation that runs in Ether, but runs at compile time rather than evaluation time. When the compiler runs across an `assert` statement it checks a list of possible *assertion replacement procedures* that are indexed by the class of assertion they are to replace to see if one matches. If it finds one that matches, the body of the replacement procedure is evaluated and it will return code that replaces the `assert` statement. An example assertion replacement procedure is shown in figure 20.

Fig. 20. Length Assertion Replacement Procedure

```
(def-assert-vca (length =obj =number)
     '(establish-object-point-assert
        object        ,obj
        message-type  'assert-length
        property      ,number))
```

The assertion replacement procedure itself has a pattern, in this case `(length =obj =number)`. This pattern says that the replacement procedure is capable of replacing any assertion that has the symbol `length` in the first positions, and any other two objects in the second and third positions. If an `assert` statement matches these characteristics, then this procedure will be invoked with the variables `obj` and `number` bound to the second and third parts of the assertion respectively. The procedure will return code that is to be evaluated to get the virtual effect of the `assert`.

As an example, suppose we had some code that contained the command:

```
(assert (length →x →n))
```

At compile time, the replacement procedure shown in figure 20 would be invoked and would replace the `assert` expression with the expression:

```
(establish-object-point-assert
   object        x
   message-type  'assert-length
   property      n)
```

The reader is not expected to understand what this expression means at this point, only that it has replaced the Ether code the user wrote. It actually is a macro that yet expands into something else.

There is a completely analogous mechanism that replaces sprites with the code that implements them. A sprite replacement procedure for `length` assertions is shown in figure 21.

Fig. 21. Length Sprite Replacement Procedure

```
(def-when-vca (length =obj =number)
     (if (ether-variable obj)
         '(establish-object-point-sprite-every-instance
              object-type    'object
              message-type   'when-length
              property        ,number
              body           ,*body*
              variable       ,obj
              activity       current-activity)
         '(establish-object-point-sprite
              object          ,obj
              message-type   'when-length
              property        ,number
              body           ,*body*
              activity       current-activity)))
```

Whenever the compiler comes across a sprite that has a pattern with the symbol `length` in the first position and any two other objects in the other positions, this sprite replacement procedure is invoked. The replacement procedure is invoked with `obj` and `number` bound to the second and third items in the sprite pattern. There is an additional variable, `*body*`, that is bound to the body of the sprite; this is the code that should be executed if the sprite is triggered. You will notice that the particular sprite replacement procedure has a conditional that checks a predicate on `obj`, the second object in the pattern. The predicate, `ether-variable`, returns T if its argument is an Ether pattern variable (i.e. one prefixed by "=").

For example, if we were to compile a sprite of the form:

```
(when {(length →list →n)}
  (function-to-execute))
```

the `ether-variable` test would fail and we would return the expanded code from the else part of the `if`. The expanded code would look like:

```
(establish-object-point-sprite
    object         list
    message-type   'when-length
    property       →n
    body           (function-to-execute)
    activity       current-activity)
```

If, on the other hand, the sprite to be compiled looked like:

```
(when {(length =list →n)}
  (function-to-execute))
```

the other branch of the `if` would have been taken and the replacing code would become:

```
(establish-object-point-sprite-every-instance
    object-type    'object
    message-type   'when-length
    property       →n
    body           (function-to-execute)
    variable       '=list
    activity       current-activity)
```

In the current version of the system *all* `assert` and `when` statements are replaced at compile time in the manner just described. Actually, the functions they expand into:

```
                establish-object-point-sprite
        establish-object-point-sprite-every-instance
                establish-object-point-assert
```

are themselves macros that expand into something else.

7.3 Objects and Object Oriented Programming

The code that implements the effects of sprites and assertions depends very heavily on the metaphor of *object oriented programming.* A brief discussion of the concept was contained in section 6.1. The *objects* that are parts of our assertions in Ether programs, such as `input1` and `accumulant` for the program synthesis system and `column1` for the cryptarithmetic system are *objects* in the object-oriented sense -- they can accept and respond to messages. We will see that the code produced by the assertion and sprite replacement procedures mentioned in the previous section actually turn into commands to send messages to these objects. Each one of these message transmissions makes use of the message sending primitive discussed in section 6.4, and thus each message transmission is queued for processing via the activity mechanism.

7.3.1 Defining an Ether Object

In dealing with problem solving systems written in Ether, we must define the classes of assertions with which they can deal. As we will see, there is a close relationship between the classes of assertions that can be made and matched by sprites and the class of messages that the objects of our system can accept. To keep the amount of code to a minimum, we have established conventions for how the various message names and instance variables of the objects are named. Much of the code that defines the objects and message handlers is generated automatically by the system. The Ether message passing code is built on top of the *flavor* system of the Lisp machine [68], although it does not make use of the flavor concept in any particular way.† This was merely the most expedient way of getting objects that can take messages on the Lisp machine.

Ether objects are defined by executing the function `defobject`. `Defobject` takes three arguments:

1. The type of the object.

2. A list of those properties that may be *viewpoint dependent.*

3. A list of those properties that are *viewpoint independent* -- they are the same in all viewpoints.

For the program synthesis system, we have only one kind of object, which we call (appropriately enough) an *object.* For the cryptarithmetic system we will discover that there are three different kinds of objects. Although we can refer to several different kinds of objects in the *Lisp* sense (lists, numbers, and atoms) we can have Ether objects that we may not know in some particular viewpoint to be any of these. For example, we can have an element that we know is the first element of some list but we know absolutely nothing else about it. We could assert in one viewpoint that it is a number and in another viewpoint that it is a sequence. For this reason we have only one *Ether object* type, and can tell individual objects that they are of a specific *programming object* type. The definition of a object for the program synthesis system looks as follows:

† Although, there is perhaps some advantage to doing so. In Ether, we have not gotten any advantage out of ways to structure classes of objects since the original concept was developed by Simula. This will be discussed in section 8.4.6.

```
(defobject object
  (type equal not-equal coreferential not-coreferential length
       member not-member sequence-element less greater)
  (typed-object constant-object))
```

There are two viewpoint-independent properties, `typed-object` and `constant-object`. For certain objects, we know at *object creation* time what type they are. In our descriptions of the inputs and outputs of the functions we are synthesizing we may know they are of type *sequence*, *atom*, or *number*. In these cases, after we create the object, we send it a `typed-object` message telling it that it is this particular type.

Similarly, for certain objects, we may know not only that it is of a particular type, but also the exact object that it is. There are certain distinguished objects the system knows about. The object we have been calling `nill` is one such object. It is the sequence with no elements. For this object, the `constant-object` field is set to the symbol `null-sequence`. There is one other class of constant objects that the system knows about, those which are known to be specific numbers. For example the object representing the integer 2 in our system has a `typed-object` field of `number` and a `constant-object` field of 2.

7.3.2 What Happens When You Define an Object

For each Ether object you create, the system does a number of things. For the purpose of the following discussion, assume we have just executed the code:

```
(defobject thing (prop1 prop2) (invariant-prop1 invariant-prop2))
```

The following things happen:

1. A new *class* is defined called `thing` with instance variables `invariant-prop1`, and `invariant-prop2`, and `viewpointed-object-table`. The objects we actually manipulate with our assertions and sprites are objects belonging to this class.

2. A function is defined called `new-thing` that returns an object of type `thing`. It takes various optional arguments as shown in the examples in chapters 5 and 4.

3. The class `thing` is defined to take the following messages: `assert-prop1`, `assert-prop2`, `when-prop1`, and `when-prop2`. When an assertion is made about a property of an object, as for example, we had executed the code:

```
(assert (length →obj →n))
```

a message transmission of the `assert-length` happens to the object bound to `obj`. Why this happens will become clear as we go on. When a sprite is create that asks about one of the properties of an object, a `when-` type message is sent, sometimes to one object, and sometimes to many objects.

4. Message handlers are created for all `assert-` and `when-` type messages for the class `thing`. These message handlers actually "redirect" the information or request for information to the place(s) it/they are actually handled.

5. A new class is defined called `thing-with-viewpoint` that has instance variables: `prop1-property`, `prop2-property`, `this-object`, `pending-sprites`. The instance variables `prop1-property` and `prop2-property` store information about properties *prop1* and *prop2*. For example, if the object were a sequence, the `length-property` would be bound to an indication of the *length* of the object, if it is known. Each object of type `thing-with-viewpoint` is related to a particular object of type `thing`. In fact, for each object of type `thing`, and for each viewpoint for which we know something (i.e. have `assert`ed at least one thing) about this object, we have a *specific* object of type `thing-with-viewpoint` that contains those facts we know about this object *with respect to* this viewpoint. Objects of type `thing` have an instance variable called `viewpointed-object-table` that indexes these viewpoint-specific objects with their corresponding viewpoints.

6. The class `thing-with-viewpoint` is defined to, like the class `thing`, take messages of the form `assert-prop1`, `assert-prop2`, `when-prop1`, and `when-prop2`. The message handlers for object `thing` are written by the designer of the problem solver; they represent special purpose ways of storing and retrieving the information that are based on the semantics of the assertional types. There is another class of message handlers the user must write. They are of the form `merge-prop1` and `merge-prop2`. These are instructions to the system on how to do viewpoint inheritance. When one viewpoint has more than a single parent viewpoint, a method must be supplied that tells the system how to find the corresponding property for the child viewpoint that is consistent with all parents and contains no constraints that are not inherited from some parent.

7. A variable called `thing-pending-sprites` which contains, in effect, code that implements sprites that ask questions about objects that may not exist yet.

7.3.3 The Concept of A Point Sprite

When we write an Ether program containing sprites, we think of the sprites as sitting on the sidelines "watching" for assertions to trigger on. These sprites are part of some *activity* and the rate at which they can run (or the "eagerness" with which they can jump at assertions) is controlled by the activity. Of course, computers of the sort we are used to can't directly interpret code that works this way. In some manner we have to massage the sprite into a procedure that will get invoked in some orderly way when the information it is "watching for" becomes available. In most other assertion-oriented languages the sprite patterns and associated procedures are incorporated into a lexical discrimination net. In Ether there is no uniform mechanism; the procedures that implement sprites are accessed in any of a number of places, depending on the kind of information the sprite is looking for. There is, however, a uniform method of representing the procedure that implements the sprite; it is known as a *point sprite.* A point sprite is a structure, that has five slots:

Point-sprite-key represents the *parameter* of the point sprite. For example, if the pattern of the sprite that led to this point sprite was:

```
{(length →obj →n)}
```

the point sprite would be placed in a location where it would get checked whenever we have asserted anything about the length of the object bound to `obj`. The *key* of this point sprite would be the number object bound to `n`. If it matched the known length, the point sprite would then execute.

Point-sprite-body contains the code that was contained in the body of the original sprite.

Point-sprite-closed-variables and **point-sprite-closed-values** together represent the lexical environment in which the original sprite was defined. When the body of the point sprite is executed, it is executed in this environment.

Point-sprite-activity is the activity in which the original sprite was activated. If and when the body of this point sprite is ever executed, it will be done in this activity.

Since executing a point sprite is a very common thing to do, a convenient function is provided for doing it. If we execute the code:

```
(point-sprite-eval point-sprite)
```

An event record is created as described in section 6.2.1 with the body and environment set appropriately.

The event record is placed at the end of the queue for the activity mentioned in the point sprite.

Occasionally a point sprite wants to *augment* the environment in which the body is evaluated. Optional keyword arguments can be supplied to `point-sprite-eval` to do this. For example:

```
(point-sprite-eval point-sprite
     variable 'x
     value    element)
```

Would evaluate the point sprite (bound to `point-sprite` in the environment of the point sprite augmented with a binding of the variable `x` to the value of `element`.

7.4 A Very Simple Virtual Collection

Perhaps the simplest virtual collection in both example systems is the one that handles `(CONTRADICTION)` assertions. Every viewpoint is a structure that has a field called `viewpoint-contradiction-marker` that has the value `T` if the viewpoint is known to be contradictory. The when replacement procedure for this class of assertions is shown in figure 22.

Fig. 22. When Replacement Procedure for Contradictions

```
(def-when-vca (contradiction)
  '(let ((point-sprite
            (make-point-sprite
              point-sprite-closed-vars current-closed-variables
              point-sprite-closed-vals current-closed-values
              point-sprite-body         ,*body*
              point-sprite-activity     current-activity)))
     (if (viewpoint-contradiction-marker *viewpoint*)
         (point-sprite-eval point-sprite)
         (structpush point-sprite (viewpoint-contradiction-point-sprites *viewpoint*)))))
```

Since there are no arguments to these contradiction assertions, the code that does the replacement is quite simple. The first thing the sprite replacement procedure does is create a point sprite with the environment set to the lexical environment at the point of call (i.e. the environment defined by the variables `current-closed-variables` and `current-closed-values`). The body of the point sprite is simply the body of the original sprite, and the activity is the activity currently running.

After creating the point sprite, the code checks to see if the viewpoint is already known to be contradictory. If so, the point sprite is evaluated. This implements the intended effect of the sprite; if the viewpoint is contradictory we would like the body evaluated. Remember that *evaluating the body of* *a point sprite* is not simply a call to the Lisp evaluator. As explained in section 7.3.3 it involves a queuing of the code to be evaluated onto the event queue of the activity mentioned in the point sprite.

If the viewpoint is not (currently) known to be contradictory we must save the point sprite in case at some future time it is learned to be contradictory. This is accomplished by executing the code:

```
(structpush point-sprite (viewpoint-contradiction-point-sprites *viewpoint*))
```

which causes the point sprite to be added to a list of point sprites interested in knowing if this particular viewpoint has become contradictory.

The companion procedure, the assertion replacement procedure for contradictions, is shown in figure 23.

Fig. 23. Assert Replacement Procedure for Contradictions

```
(def-assert-vca (contradiction)
    '(if (not (eq (viewpoint-contradiction-marker *viewpoint*) t))
         (progn
           ;Otherwise, set the contradiction marker to T.
           (setf (viewpoint-contradiction-marker *viewpoint*) t)
           ;Evaluate the accumulated point sprites.
           (foreach
             point-sprite
             (viewpoint-contradiction-point-sprites *viewpoint*)
             (point-sprite-eval point-sprite)))))
```

The replacement code first checks to see if the `viewpoint-contradiction-marker` for the viewpoint is `T`, an indication that the viewpoint is already known to be contradictory. In this case, there is nothing to be done. If the viewpoint was *not* already known to be contradictory, we set the `viewpoint-contradiction-marker` to indicate that it is. We then iterate through each of the accumulated point sprites and `point-sprite-eval` them.

The reader should carefully study the implementation of `(CONTRADICTION)` assertions to satisfy himself that they correctly model the *effect* of sprites and assertions that satisfy the properties of monotonicity, and commutativity. In particular, note:

1. If a `(CONTRADICTION)` is asserted, and a sprite activated looking for this assertion, the body will be evaluated regardless of the actual order of the assertion and sprite activation.

2. If a `(CONTRADICTION)` is asserted, the bodies of every sprite watching for this to happen will be executed exactly once, and will be executed in the activity the sprite was originally activated in.

Although this example is very simple, and does not involve significant use of objects and message passing, it does indicate the basic principles used in ensuring monotonicity and commutativity are maintained. The implementation of `assert` checks a list of point sprites at a predetermined location for ones that should fire. The implementation of `when` both checks to see if the sprite should fire now, and adds a point sprite to the appropriate list of point sprites in case information learned later will enable the sprite.

7.5 The Length Virtual Collection

In this section we will discuss the implementation of a more interesting virtual collection, one involving ether objects and message passing, as discussed in section 7.3.1. As implementers of the sprites and assertional replacement procedures, we have made a decision about the best repository for information about the lengths of objects. `Length` is a relation of two arguments, a sequence and a number. If both of these arguments are *objects* that can store information about themselves we have two choices for how to store this information:

1. A sequence can have a property known as its `length` in which its length is stored.

2. A number can have a property known as `sequences-of-this-length` where a list of all such sequences are stored.

For a number of reasons (1) is better than (2). One obvious reason is that a sequence can have only one length, but there may be many sequences of length, say, 1. An absurd possibility if (2) is chosen as the representation can be found by considering a number objects for which no constraints are known about its actual value. Then any sequence known to the system could end up on its `sequences-of-this-length` property. There are other reasons why (1) is preferable, but they must await a further description of the mechanism.

7.5.1 The Handling of Length Assertions

The assertion replacement procedure for `length` assertions was shown in figure 20 and is repeated in figure 24 for convenience. We will go step-by-step through what actually happens when this code gets expanded (at compile time) and then what happens when the code is evaluated. Assume that the assertion we are replacing looks like:

Fig. 24. Length Assertion Replacement Procedure Repeated

```
(def-assert-vca (length =obj =number)
     '(establish-object-point-assert
        object        ,obj
        message-type  'assert-length
        property      ,number))
```

```
(assert (length →a-sequence →a-number))
```

This code is replaced by the following:

```
(establish-object-point-assert
  object        a-sequence
  message-type  'assert-length
  property      a-number)
```

`Establish-object-point-assert` is also a macro and *it* expands into

```
(send a-sequence
    'assert-length
    (make-assert-message
        assert-message-basic        a-number
        assert-message-viewpoint    *viewpoint*)
    current-activity)
```

This code serves as the final replacement for the original assertion.

When this code is executed, it causes an event record to be added to the end of the queue for the then current activity. When the event is finally executed, a message will be sent to the object `a-sequence` of type `assert-length` and with a contents that consists of two components: the *basic* part of the message (the property of the object that is being asserted) and the *viewpoint* the assertion is made in (which is bound to the variable `*viewpoint*`).

As described in section 7.3.2, the message handler for this message by the object is constructed by the code that created the Ether object. What this message handler does is *redirect* the message to another object. As was discussed on page 125, there is a class called `object-with-viewpoint`, and there is one instance of this class defined for each object and for each viewpoint for which we know something about that object with respect to the viewpoint. In other words, there is an object that represents what is known about the *Ether* object bound to `a-sequence` in the viewpoint bound to `*viewpoint*`.

When this particular assertion is made the object to which the message is redirected may or may not yet exist. If it does exist, it will be in the `viewpointed-object-table` associated with the object `a-sequence` indexed by the viewpoint. If it does not exist it must be created on the spot and sent the message; we will defer the discussion of how it is created until later. Assume for now it is in the table.

In our `defobject` description of the Ether object for the program synthesis system we gave `length` as one of the viewpoint-dependent properties. Therefore the viewpoint-specific object will have an instance variable called `length-property` that will contain an indication of our belief about the objects length. It will, in fact, contain `nil` if nothing so far has been said about its length, and an Ether object of type *number* that represents the information we know about the object's length.

The message handler for the `assert-length` message for these viewpointed objects is shown in figure 25.[†] The first thing the handler does is try to *merge* the newly asserted length with the already known length. We won't go into the details of numbers, or merging them right now, but the function `number-merge` will return a number object that represents all the constraints contained in both

Fig. 25. Length Assertion Message Handler

```
(defmethod (viewpointed-object assert-length) (n)
  (let ((new-length (number-merge length-property n)))
    (if (not (number-equal new-length length-property))
        (if (null new-length)
            ;If the newly asserted length is not consistent with believed length, the viewpoint is contradictory.
            (assert (contradiction))
            ;Otherwise we have learned new information.  It is recorded and applicable point sprites are executed.
            (progn
              ;Assign the new length property
              (setq length-property new-length)
              ;Remove point sprites from stifled activities.
              (clean-up-point-sprites-list length-point-sprites-list)
              ;Check each point sprite to see if it should get executed.
              (foreach
                point-sprite
                length-point-sprites-list
                (let ((key (point-sprite-key point-sprite)))
                  (if (and (ether-variable key) (ether-numberp length-property))
                      ;If the key is an ether variable, bind the variable and eval the point sprite body.
                      (point-sprite-eval point-sprite
                            variable (ether-variable key)
                            value    length-property))
                      ;Otherwise it is a non-variable.   Check for matchedness.
                      (if (number-merge length-property key)
                          ;If they do merge, eval the body.
                          (point-sprite-eval point-sprite)))))
              ;If this viewpoint handler has subviewpoint handlers, service them.
              (foreach
                subviewpoint-handler
                subviewpoint-handlers
                (let ((*viewpoint* (<- subviewpoint-handler 'viewpoint)))
                  (<- subviewpoint-handler 'assert-length length-property)))))))
```

† `Defmethod` is a standard construct on the Lisp machine for defining message handlers. There is a handler defined for each *class* (known on the Lisp machine by the name "flavor") in this case `viewpointed-object`, and each message type, in this case `assert-length`. The message handler takes an argument list. This one was supplied with a singleton argument list: `(n)`.

number objects. For example, if one argument was a number known to be greater than 1 and the other was a number known to be less than 4, `number-merge` would return a number object that would know it was a number between 1 and 4 (i.e. either 2 or 3). This new number object then gets bound to `new-length`. We evaluate the function

```
(number-equal new-length length-property)
```

to determine whether we have learned any new information about the length of the object. If we have *not* learned any new information, then there is nothing else to be done and we complete the processing of this message. If we have learned new information, we continue. By convention, the function `number-merge` returns `nil` if the the objects do not merge, i.e. there is no integer that satisfies the combined constraints of both its arguments. We check to see if `new-length` is `null`. If so, the viewpoint we are in is inconsistent and we must assert this. We do this by evaluating the code:

```
(assert (contradiction))
```

It is possible to include assertions (and as we will see soon, sprites) in the Lisp code that defines a virtual collection of assertions. If the viewpoint is inconsistent, there are no other things to do. Otherwise, we have a number of things to do. We execute:

```
(setq length-property new-length)
```

to store the new knowledge we have obtained about the object's length. There is an instance variable of this object called `length-point-sprites-list` that contains point sprites created by sprites previously activated that are interested in the length of this particular object. These point sprites could be in any of a number of activities. It is possible that some of these activities are stifled, in which case we no longer want to service them. Point sprites in stifled activities should be removed from the list. We do this by executing:

```
(clean-up-point-sprites-list length-point-sprites-list)
```

We then iterate through the list of point sprites on the list `length-point-sprites-list` and execute all point sprites that are appropriate to execute. We will save the details of this until we have a chance to explain how these point sprites are created. The final piece of code we execute is:

```
(foreach
   subviewpoint-handler
   subviewpoint-handlers
   (let ((*viewpoint* (<- subviewpoint-handler 'viewpoint)))
     (<- subviewpoint-handler 'assert-length length-property)))
```

Each object of this type has an instance variable called `subviewpoint-handlers` that contains a list of all objects that represent facts known about this particular Ether object in viewpoints that inherit from the viewpoint we are currently servicing. Since these viewpoints inherit all information, we must notify them of the newly learned length property. This is accomplished by a Lisp machine-type message send:[†]

```
          (<- subviewpoint-handler 'assert-length length-property)
```

It is worth noting that this one piece of code is all we have to write to allow the viewpoint inheritance mechanism to operate correctly. (There is, of course, alot of code invoked by `Defobject` but this is of no concern to designers of Ether subsystems.)

7.5.2 The Handling of Length Sprite Patterns

The replacement procedure for length sprites is shown in figure 26.

Fig. 26. Replacement Procedure for Length Sprites

```
(def-when-vca (length =obj =number)
     (if (ether-variable obj)
          '(establish-object-point-sprite-every-instance
               object-type    'object
               message-type   'when-length
               property        ,number
               body            ,*body*
               variable        ,obj
               activity       current-activity)
          '(establish-object-point-sprite
               object          ,obj
               message-type   'when-length
               property        ,number
               body            ,*body*
               activity       current-activity)))
```

Somewhat different things happen depending on whether the arguments in the `length` pattern are variables or not. As can be seen from examining figure 26, there is a conditional that that picks between two possible replacements. The condition is true if the first argument position (i.e. the sequence, bound to `obj`) is a variable or not. We consider first the case of it not being a variable because it is the most analogous to the cases we've examined so far. Suppose the sprite we are replacing looks like:

```
(when {(length →a-sequence →n)}
  (contents-of-body))
```

[†] This message send does not go through the Ether activity mechanism and is executed as an "uninterruptible" event. See 8.4.1 for a discussion of this and related issues involving viewpoint inheritance.

The replacement code becomes:

```
(establish-object-point-sprite
  object        a-sequence
  message-type  'when-length
  property      n
  body          (contents-of-body)
  activity      current-activity)
```

The above code is also a macro and *it* expands into:

```
(send a-sequence
      'when-length
      (make-point-sprite-message
        point-sprite-message-basic        n
        point-sprite-message-viewpoint    *viewpoint*
        point-sprite-message-body         '(contents-of-body)
        point-sprite-message-closed-vars  current-closed-variables
        point-sprite-message-closed-vals  current-closed-values
        point-sprite-message-activity     current-activity)
      current-activity)
```

When this code is executed, `a-sequence` and `n` are bound to specific objects. Analogously with the `assert` replacement, this code sends `a-sequence` a message of type `when-length`. The contents of the message includes the basic part, the number, `n`, for which we would like the sprite to trigger if the length is determined to be that number, and the viewpoint. Also included are the body of the sprite that is to be evaluated along with its current environment and the activity of activation. This message is sent to the object `a-sequence`, and analogously with `assert-` type messages it is redirected to the object that represents what is known about `a-sequence` in the viewpoint bound to `*viewpoint*`.

The message handler for `when-length` messages for viewpointed objects is shown in figure 27. The

Fig. 27. Length Sprite Message Handler

```
(defmethod (viewpointed-object when-length) (key body closed-vars closed-vals activity)
  (let ((point-sprite
          (make-point-sprite
             point-sprite-key          key
             point-sprite-body         body
             point-sprite-closed-vars  closed-vars
             point-sprite-closed-vals  closed-vals
             point-sprite-activity     activity)))
    (push point-sprite length-point-sprites-list)
    (cond
      ((and (ether-variable key) (ether-numberp length-property))
           ;If the key is an ether variable, bind the variable and eval the point sprite body.
       (point-sprite-eval point-sprite
                          variable (ether-variable key)
                          value    length-property))
      ((number-equal key length-property)
       ;Otherwise the key is a number.  If it's equal to the believed length, eval the body.
       (point-sprite-eval point-sprite)))))
```

method that redirects the message unpackages the various arguments to the message handler in figure 27. The first thing the handler does is create a point sprite. The *point-sprite-key* field of the point sprite is set to n -- the length we wish to trigger on. The point sprite is added to `length-point-sprites-list` in case future `asserts` might cause it to trigger. We then check to see if the information is already known that might make the sprite trigger; we enter the `cond` expression. The predicate of the first clause of the `cond` first checks to see if the key is an ether-variable (prefixed by the symbol "="). It isn't, so we fall through to the second clause. This clause checks to see whether the key (in this case n is an equivalent number object to the currently believed length. If they are, the "sprite" has been triggered and we execute the point sprite. As before, this execution causes a new event record to be added to the end of the activity of the point sprite with instructions to evaluate the body.

Now consider what happens if the second argument position of the pattern is an Ether variable, as in the sprite:

```
(when {(length →a-sequence =x)}
  (contents-of-body))
```

Glancing back at figure 26, we see the code expands into an essentially similar form; the only difference being the key is now =x instead of n. This code, in turn, expands into the following `send`.

```
(send a-sequence
      'when-length
      (make-point-sprite-message
        point-sprite-message-basic         '=x
        point-sprite-message-viewpoint     *viewpoint*
        point-sprite-message-body          '(contents-of-body)
        point-sprite-message-closed-vars  current-closed-variables
        point-sprite-message-closed-vals  current-closed-values
        point-sprite-message-activity      current-activity)
       current-activity)
```

The message when finally processed will get sent to the same message handler in figure 27. Since the key is an Ether variable this time, we pass the first test in the predicate of the first clause of the `cond`. The second clause checks to see if the `length-property` is a *specific* number; if so the sprite should fire. This time, however, we must assure the variable x gets bound to the believed length of the sequence which is, in this case, the `length-property`. So we execute:

```
(point-sprite-eval point-sprite
       variable (ether-variable key)
       value    length-property))
```

This will cause the body of the point sprite to get executed through the activity mechanism with the environment augmented by the binding of the variable x to the length of `a-sequence`. Looking back at the original sprite,

```
(when {(length →a-sequence =x)}
  (contents-of-body))
```

we see that this is precisely the behavior desired. If we know what the length of a-sequence is, we bind x to that and evaluate the body.

Now that we've covered the nature of point sprites and their execution for length assertions, we should look back at figure 25 which contains the handler for assert-length messages to see how point sprites already in existence are processed. The code that iterates through all known point sprites is:

```
(foreach
  point-sprite
  length-point-sprites-list
  (let ((key (point-sprite-key point-sprite)))
    (if (and (ether-variable key) (ether-numberp length-property))
       (point-sprite-eval point-sprite
              variable (ether-variable key)
              value    length-property))
       (if (number-merge length-property key)
           (point-sprite-eval point-sprite))))
```

We check each point sprite on the length-point-sprites-list and extract its key. If the key is an Ether variable, and the length-property is a number, we augment the environment of the point sprite by binding the key to the length-property and evaluating it. If the key is not an Ether variable, we check to see if the key and the length-property merge. If so, we evaluate the point sprite. Note that the function ether-variable serves a dual function. It is a predicate that is true iff its aregument is an Ether variable; it is also a function that extracts the actual variable name. Thus, (ether-variable '=x) evaluates to x.

We now consider what happens if the length assertion contains an Ether variable in the *first* argument position. That is, consider the compilation of a sprite of the form:

```
(when {(length =x →n)}
  (contents-of-body))
```

This sprite will be triggered by *any* sequence whose length is n. Since we have chosen to represent information about lengths of sequences by storing the information with the sequence rather than the number, this would seem to pose a dilemma. In some way, we must cause a point sprite to appear on *every* sequence. The sprite replacement procedure shown in figure 26 leaves us with the following code:

```
(establish-object-point-sprite-every-instance
   object-type     'object
   message-type    'when-length
   property        n
   body            '(contents-of-body)
   variable        'x
   activity        current-activity)
```

whose function is to get the appropriate point sprite on the `length-point-sprites-list` of every sequence, both those that are already known to exist and those that might be created *at any future time.* The above code expands into the slightly more complicated expression shown in figure 28.

Fig. 28. Implementation of Length Sprites With Sequence Variable

```
(progn
  ;For every object that has so far been created, activate this point sprite.
  (foreach
    object
    object-all-instance-list-name
    (send object
          'when-length
          (make-point-sprite-message
                point-sprite-message-basic         n
                point-sprite-message-viewpoint     *viewpoint*
                point-sprite-message-body          '(contents-of-body)
                point-sprite-message-closed-vars  (cons 'x current-closed-variables)
                point-sprite-message-closed-vals  (cons object current-closed-values)
                point-sprite-message-activity      current-activity)
          current-activity))
  ;Place a copy of the sprite message on the pending list for objects created later.
  (push
    (list 'when-length
          (make-point-sprite-message
                    point-sprite-message-basic         n
                    point-sprite-message-viewpoint     *viewpoint*
                    point-sprite-message-body          '(contents-of-body)
                    point-sprite-message-closed-vars  (cons 'x closed-vars)
                    point-sprite-message-closed-vals  closed-vals
                    point-sprite-message-activity      activity))
    object-pending-sprites-list))
```

When the function `new-object` is run, it does two things that make it possible for the code in figure 28 to function correctly.

(1) It adds the new object to the list `object-all-instance-list-name` so that sprites of this form created in the future will be able to get ahold of the object.

(2) It goes down a list called `object-pending-sprites-list` which contains point sprites that implement sprites that were activated *before* the object was created. Each of these point sprites contains an ether variable for its key and so it ultimately must appear on the `length-point-sprites-list`

for every object.

By referring to figure 28 we can see how this code interacts with the definition of `new-object` to get the desired effect. The first half iterates through each object in the list `object-all-instance-list-name` and sends the `when-length` message to each. Notice that the environment (arguments *point-sprite-message-closed-vars* and *point-sprite-message-closed-vals*) have been augmented with a binding of the Ether variable (the variable `x`) to the particular object. If the point sprite gets executed within any particular object, the variable `x` will get bound to that object. Since the original sprite pattern was:

```
{(length =x →n)}
```

this is the desired behavior.

The code in the second half of figure 28 is responsible for adding the point sprite to the `object-pending-sprites-list`. It actually adds an indication of the message that must be sent to the newly created objects. This consists of a list of the message type (i.e. `when-length`) and the parameters of the message. This contains sufficient information to allow the objects, when receiving the message, to construct the point sprite. Notice that the argument *point-sprite-message-closed-vars* is augmented with the variable `x` but the companion argument *point-sprite-message-closed-vals* has nothing added to it. If and when a new object is created, this object is added to the list of closed values for the point sprite before the message is sent.

The reader should now review the virtual collection implementation of `length` assertions to check that the properties of commutativity and monotonicity are maintained. No matter what order sprites and matching assertions are created, the sprites will get evaluated exactly once and with the proper environment.

7.5.3 The Creation of Viewpointed Objects

Thus far we have been assuming that when a message is sent to an Ether object it will get redirected to an already existing object that represents what is known about the particular object in the viewpoint mentioned in the message. Clearly these viewpoint-specific objects must get created at some point. As mentioned on page 125 there is a `viewpointed-object-table` that contains these viewpointed objects indexed by the viewpoint. When a message is sent to an Ether object, and there is already a viewpointed object corresponding to it in the table, the message is redirected to that object. Otherwise a new object is created and inserted in the table. If the new viewpoint is one that inherits from no other viewpoints, the new object is simply created and inserted in the `viewpointed-object-table`, the original message is redirected to it, and we are done. The more interesting case is where the viewpoint referred to in the original message is one that has one or more parent viewpoints. Somehow we must ensure that all the knowledge possessed by each of the parent viewpoints about this object is reflected in the knowledge stored in the newly created object.

We have a simple, uniform method by which this inheritance is accomplished. For each viewpoint-specific characteristic we have supplied message handlers for `assert` and `when-` type messages. To enable automatic inheritance we must supply one additional message handler for `merge-` type messages. The handler for `merge-length` messages is:

```
(defmethod (viewpointed-object merge-length) (length)
  (<- self 'assert-length length))
```

After the new viewpointed object is created, the system sends one `merge-` message for each property and for each parent viewpoint. The argument to the merge message is the property of the particular object in the parent viewpoint, in this case `length-property`. It is the function of the merge message handler to decide how the information is to be merged. In the case of `length`, the `length-property` represents the believed length and so by sending the new object an `assert-length` message with the believed length we have accomplished our purpose. If there are several parent viewpoints, one `assert-length` message is sent and the new object will merge the various `length-property`'s to obtain the `length-property` for the newly created viewpointed object representing the inheriting viewpoint. If they fail to merge, then a `(CONTRADICTION)` is asserted as desired. After all the properties are merged from the parent viewpoint(s), the original message that mentioned this new viewpoint (and led to the creation of this new object) is delivered. In the event that there are parent viewpoints for the new viewpoint for which no object exists in the `viewpointed-object-table`, the procedure is applied recursively to create these objects.

Our method for handling viewpoint inheritance has some worthwhile characteristics from the point of view of efficiency. At the time of viewpoint creation only a very small amount of work actually has to be done. New viewpoint-specific objects for Ether objects are only created when something new has been said about that particular Ether object in the particular viewpoint. See section 8.4.1 for some further discussion of efficiency issues related to viewpoint inheritance.

7.6 The Member Virtual Collection

We won't go through each virtual collection definition in very much detail, but it is worth going through at least one more to give some feel for the different ways it is possible to handle different classes of assertions. The replacement procedures for both member assertions and sprites is shown in figure 29.

Fig. 29. Replacement Procedures for the Member Virtual Collection

```
(def-assert-vca (member =element =list)
     '(establish-object-point-assert
        object         ,list
        message-type   'assert-member
        property       ,element))

(def-when-vca (member =element =list)
       (if (ether-variable list)
           '(establish-object-point-sprite-every-instance
                object-type    'object
                message-type   'when-member
                property       ,element
                body           ,*body*
                variable       ,list
                activity       current-activity)
           '(establish-object-point-sprite
                object         ,list
                message-type   'when-member
                property       ,element
                body           ,*body*
                activity       current-activity)))
```

The form of the replacement procedures is quite analogous to the ones for length assertions shown in figures 24 amd 26.

In the implementation of member assertions we again have two choices as to which of the two objects should be the repository for the information. Either: (1) Each sequence can known which objects are members of it, or (2) each object can know which sequences contain it. We have chosen the first of these. Our reasoning is that few lists will have many objects that are members of them, but there will be a number of objects that will be members of many lists.

As was the case with the length virtual collection the replacement procedures shown in figure 29, the replacing procedures establish-object-point-assert, and establish-object-point-sprite each expand into assert- and when- type message transmissions respectively. The form establish-object-point-sprite-every-instance also expands into the appropriate message transmissions with the necessary code to ensure that future objects will get delivered the message. The expanded code is so analogous that we will not repeat the explanation here. Viewpoints, message redirection, and viewpointed object creation happen identically with the case of `length` type assertions. The only places we find differences are in the handlers for the three messages `assert-length`, `when-length`, and `merge-length`. We will describe the workings of each of these handlers.

The `Defobject` definition of Ether objects for the program synthesis system causes the class of viewpoint-specific objects to have an instance variable called `member-property`. The way `member-property` is used to "remember" what things are members of what is up to the designers of the `member` virtual collection. We have chosen to let `member-property` be a list where the elements of the list are those things known to be members within the specific viewpoint.

The handler for `when-member` assertions is shown in figure 30.

Fig. 30. Sprite Handler for Member Assertions

```
(defmethodc (viewpointed-object when-member) (key body closed-vars closed-vals activity)
  (slet ((point-sprite
          (make-point-sprite
             point-sprite-key         key
             point-sprite-body        body
             point-sprite-closed-vars closed-vars
             point-sprite-closed-vals closed-vals
             point-sprite-activity    activity)))
    (push point-sprite member-point-sprites-list)
    (cond
      ((ether-variable key)
       ;If the key is an ether variable, for every element that is currently believed to be in this list,
       ;bind the variable and eval the point sprite body with the variable bound to that element.
       (foreach
         element
         member-property
         (point-sprite-eval point-sprite
              variable (ether-variable key)
              value    element)))
      (member-property
       (foreach
         element
         member-property
         (when {(equal →element →key)}
           (point-sprite-eval point-sprite))))))
```

As with the handler for `when-length` messages, the contents of the message includes the `key`, the object for which we desire to know `memberness` (or an Ether variable), and a `body` to evaluate along with its environment and activity. As before we create a point sprite and add it to the list `member-point-sprites-list`. We then check to see if the key is an ether variable. If it is an Ether variable, then we would like to evaluate the body of the sprite (now the *point-sprite-body* of the `point-sprite`) for each object which we know to be a member of the list in an environment in which the variable is bound to that object. Thus we let the variable `element` range over each of the objects known to be members of the list and execute `point-sprite-eval` for each:

```
(foreach
  element
  member-property
  (point-sprite-eval point-sprite
       variable (ether-variable key)
       value    element))
```

which has the effect of binding the ether variable to each element known to be a member and executing the body of the point sprite (through the activity mechanism, of course, to preserve concurrency).

For example, if we had at some point asserted:

```
(assert (member →an-object →a-list))
```

the instance variable `member-property` of `a-list` would contain `an-object` as a member. If we were then to activate a sprite of the form:

```
(when {(member =x →a-list)}
  (random-function x))
```

the variable `x` would be bound to `an-object` by the above code and the form `(random-function x)` evaluated in that environment.

In the event the `key` is an object instead of an Ether variable, (and assuming the member property has at least one element) we execute the following code:

```
(foreach
  element
  member-property
  (when {(equal →element →key)}
    (point-sprite-eval point-sprite)))
```

We iterate through each of the objects that are known to be members, and for each one *we create a new sprite* that will trigger if we learn that the object known to be a member of the list is equal to the object we asked about. If the equality sprite triggers then we evaluate the point sprite.

There are many points worth mentioning here:

(1) It is possible to use sprites inside a virtual collection handler.

(2) The sprite checking for equality will be activated inside the same activity as the original activity of the sprite checking for memberness. Thus, resource control modifications applying to the memberness sprite apply as well to the sprites created to implement it.

(3) The ability to include sprites *inside* the definition of a virtual collection leads to greater commutativity properties while keeping the code simple. For example, if we had made the assertion:

```
(assert (member →an-element →a-list))
```

and activated a sprite

```
(when {(member →another-element →a-list)}
  (random-function a-list))
```

and then at some arbitrary point in the future executed:

```
(assert (equal →an-element →another-element))
```

The sprite would fire executing the code:

```
(random-function a-list)
```

with `a-list` bound to `an-element` in the appropriate activity. The behavior of the system is totally invariant over the time ordering of the two assertions and the sprite activation.

The handler for `assert-member` messages is shown in figure 31. The single argument to the handler, `element`, is the object which is asserted to be a member of this list. The first thing we do is check to see if we already know it to be a member of the list. If it is the `new-members-list` is the same as the old `member-property`; if not the new element is cons'd onto the `member-property`.

If the predicate:

```
(not (eq new-members-list member-property))
```

evaluates to `NIL`, the information contained in this assertion is already known and there is nothing to be done. Otherwise we continue executing the rest of the code in the handler. The first thing we do is run:

```
(setq member-property new-members-list)
```

that establishes the fact that the new `element` is a member of the list so that any sprite generated in the future can access this information. The next item of code deserves some discussion:

```
(when {(not-member →element →this-object)}
  (assert (contradiction)))
```

It is of course the case that an element cannot be both a member of a list and not a member of the list at the same time. Thus if we ever learn it is *not* a member, we assert a `(contradiction)`. Here, again,

Fig. 31. Member Assertion Handler

```
(defmethod (viewpointed-object assert-member) (element)
  ;The member-property is a list of all elements currently believed to be a member of the list.
  (let ((new-members-list (if (memq element member-property)
                              member-property
                              (cons element member-property))))
    (if (not (eq new-members-list member-property))
        (progn
          (setq member-property new-members-list)
          ;It can't be both a member and not a member
          (when {(not-member →element →this-object)}
            (assert (contradiction)))
          ;Delete a point sprites belonging to stifled activities.
          (clean-up-point-sprites-list member-point-sprites-list)
          (foreach
            point-sprite
            member-point-sprites-list
            (let ((key (point-sprite-key point-sprite)))
              (if (ether-variable key)
                  ;If the key is an ether variable, bind the variable and eval the point sprite.
                  (point-sprite-eval point-sprite
                       variable (ether-variable key)
                       value    element)
                  ;Otherwise it is a non-variable.  Check for matchedness.
                  (when {(equal →key →element)}
                    ;If they do merge, eval the body.
                    (point-sprite-eval point-sprite)))))
          ;If this viewpoint handler has subviewpoint handlers, service them.
          (foreach
            subviewpoint-handler
            subviewpoint-handlers
            (let ((*viewpoint* (<- subviewpoint-handler 'viewpoint)))
              (<- subviewpoint-handler 'assert-member element)))))))
```

we make use of a sprite inside a virtual collection handler. This sprite ensures that (among other things) commutativity with respect to `equal` assertions about the costituents of the `member` and `not-member` type assertions will be abided by. For the second position of the `not-member` assertion we use instance variable `this-object` which is bound to the Ether object that is currently being processed.†

The same behavior could have been achieved by activating a sprite in every viewpoint of the form:

```
(when {(member =x =list)
       (not-member →x →list)}
  (assert (CONTRADICTION)))
```

The results would be entirely equivalent except for a minor efficiency advantage with the code inside the virtual collection. This represents a form of "hand compiling."

† A convention in message passing languages is that the variable `self` is bound to the object in which we are processing. In Ether, however, we must distinguish between two kinds of objects. The variable `self` is indeed bound to the object in which we are processing, however that is the *viewpoint-specific* object which is not the kind that is placed inside assertions and sprite patterns. Instead, we must use the non-viewpoint-specific object. This is what `this-object` is bound to.

Referring back to figure 31, the next form encountered is the invocation of the function `clean-up-point-sprites-list` which serves to remove point sprites belonging to stifled activities. We then iterate over all point sprites in the `member-point-sprites-list`. If the key is an Ether variable we bind the variable to `element`, the newly learned member, and execute the body by evaluating:

```
(point-sprite-eval point-sprite
   variable (ether-variable key)
   value    element)
```

If the key is an object, we create the following sprite:

```
(when {(equal →key →element)}
  (point-sprite-eval point-sprite))
```

If we ever learn the `key` of the point sprite is equal to the member of the list, we execute the body of the sprite.

The only remaining code necessary to complete the definition of the `member` virtual collection of assertions is the handler for `merge-member` assertions. It is this very simple piece of code:

```
(defmethod (viewpointed-object merge-member) (member-list)
  (foreach member member-list (<- self 'assert-member member)))
```

When a new viewpointed object is created this code is executed once for each parent viewpoint with the argument, `member-list`, bound to the `member-property` list of the parent. The code simply iterates through all known members of the list in the parent viewpoint and sends itself (the newly created object) and `assert-member` message for each of these objects.

The reader is again encouraged to review the definitions of the handlers for messages in the `member` virtual collection to verify that the properties of monotonicity and commutativity are maintained.

There are a number of other assertional types associated with the the Ether object used for the program synthesis system. It would become laborious to go through the implementation of each one. They are all defined in ways similar to the ones we have discussed in this chapter thus far.

7.7 Implemention of Cryptarithmetic

Most of the discussion thus far in this chapter has been about the implementation of sprites and assertions for the program synthesis system. We discussed some of the types of assertions that were used in the construction of the cryptarithmetic system in section 4.2. We never mentioned the creation of any sprites to implement the constraint propagation. By now it can be explained to the reader just how the constraint propagation was done. In the previous examples of virtual collections, whenever we *always* wanted to take some action when something was asserted, there was no need to create a sprite to watch for that assertion; we could simply include the code inside the handler for the `assert`-type message. The constraint propagation aspect of the cryptarithmetic system is in some sense "hard-wired." The action we take when some information is asserted is always the same regardless of the viewpoint we are in. Thus we are able to encode all of the actions we wish to take when something is asserted in the code for the assertion handler itself.

For the program synthesis system we had one kind of Ether object. For the cryptarithmetic system, there are three.[†] The definition of the `column` object is:

```
(defobject column (constraints)
                  (column-description left-neighbor right-neighbor))
```

There are several viewpoint invariant parameters. Each column knows its `column-description` which is a list of the letters that make up the column. It also knows of two columns, its `left-neighbor` and `right-neighbor`. It must know about these columns to propagate information about carries in and out. It has one viewpoint-sensitive property, `constraints`. Whenever something new is asserted that reflects upon this column, a `constraints` message is sent that contains the object being constrained (either one of the three letters or one of the two carries) and a list of the possible digits these can be.

The definition of the `digit` object is:

```
(defobject digit (cant-be one-of) (digit))
```

Each digit object has only one viewpoint invariant parameter, the digit that it represents. It can accept two kinds of messages. An `assert-one-of` message can be sent with a list of possible letters. The

[†] This, again, is an indication of the greater flexibility of the program synthesis system. Here we were able to create an object that could be a list in one viewpoint and, say, a number in another viewpoint. If this flexibility were not needed then we could have defined several different kinds of objects.

cant-be property is included for convenience.

The last of the three objects is the letter:

```
(defobject letter (cant-be one-of) (containing-columns letter))
```

Like the digit object it also has to know the thing it represents. Letter is bound to the symbol which is its letter. Each letter also knows the columns it occurs in. This is bound to the viewpoint-invariant instance variable, containing-columns. The cant-be and one-of fields are completely analogous to the corresponding fields contained in the digit object.

The techniques for implementing the cryptarithmetic objects are really very similar to those used for the program synthesis system. We will go through just one example. The replacement procedure for one-of type assertions is:

```
(def-assert-vca (one-of =thing =alternatives)
    (establish-object-point-assert
       object        thing
       message-type  'assert-one-of
       property      alternatives))
```

Its form is exactly like the replacement procedures for most of the other assertions we have looked at. Notice that the same procedure will work whether or not the the object type (bound to "thing") is a letter or a digit. We will just show one of the message handlers. The handler of assert-one-of messages for letters is shown in figure 32. We will briefy read through the code. We first check to see if there are any new results in the message. If the argument, choices, is a superset of the set of possible

Fig. 32. Letter Handler For One Of Assertions

```
(defmethod (letter-with-viewpoint assert-one-of) (choices)
  (if* (setdifference one-of-property choices)
       (let ((new-one-of-property (intersect one-of-property choices)))
         (if (null new-one-of-property) (assert (contradiction)))
         (foreach
           column
           containing-columns
           (assert (constraints →column (→this-object →new-one-of-property))))
         (foreach
           digit
           (setdifference one-of-property new-one-of-property)
           (assert (cant-be →this-object →digit)))
         (if (= (length new-one-of-property) 1)
            (assert (one-of →(car new-one-of-property) (→this-object))))
         (setq one-of-property new-one-of-property)
         (foreach
           subviewpoint-handler
           subviewpoint-handlers
           (slet ((*viewpoint* (<- subviewpoint-handler 'viewpoint)))
             (<- subviewpoint-handler 'assert-one-of one-of-property)))))
```

letters we already knew about (bound to `one-of-property`) there is nothing to do. This test is performed by the predicate:

```
(setdifference one-of-property choices)
```

If the test comes out true, we compute the `new-one-of-property` to be the intersection of the two lists. If `new-one-of-property` is empty, the viewpoint is inconsistent, and we assert this:

```
(if (null new-one-of-property) (assert (contradiction)))
```

We then iterate through each of the columns in the viewpoint-independent property `containing-columns`, and for each one we express the new constraints on the column:

```
(assert (constraints →column (→this-object →new-one-of-property)))
```

We then iterate through all digits which we used to believe were possible assignments to this digit (but no longer do) and for each one assert that it `cant-be`.

```
(foreach
  digit
  (setdifference one-of-property new-one-of-property)
  (assert (cant-be →this-object →digit)))
```

If the `new-one-of-property` is of length 1 (meaning that we know a unique digit that it must be in this viewpoint), we assert that the only possible assignment to this digit must be the very letter. We say this with:

```
(assert (one-of →(car new-one-of-property) (→this-object)))
```

The remainder of the handler in figure 32 establishes the new value of the `one-of-property` and handles subviewpoint inheritance in the manner we are already accustomed.

7.8 Miscellaneous Virtual Collections

There are a few assertional types that require somewhat different implementational techniques than the ones we have discussed thus far. We briefly describe them here.

7.8.1 Implementation of Restricted Universal Quantification

In a few places we have used sprite patterns that looked something like:

```
(when {(∀ m in →(list-of-integers from 1 to (- n 1))
          check   {(sequence-element →object =el →(+ m 1))
                     (sequence-element →cdr →el →m)})}
   ...)
```

The most general form is:

```
(when {(∀ var in →set
             check { -- any arbitrary list of sprite patterns -- })}
  --body--)
```

We call this *restricted* quantification† for two reasons. The objects that the variable `var` are allowed to range over must be finite in number and fully known at the time the sprite is activated; these objects are contained in the conventional Lisp list `set`. The sprite will trigger iff the sprite patterns following the *check* keyword trigger for every binding of the variable `var` to each of the elements in the list `set`. Sprites incorporating this pattern do satisfy the property of commutativity; regardless of the order in which the sprite is activated, and assertions satisfying the triggers are made, the sprite will trigger. How this particular virtual collection of assertions is implemented so that this property is maintained is the subject of this section. Note that there is no assertional form corresponding to the sprite pattern type for restricted universal quantification so it does not make sense to talk about a "monotonicity" property.

The virtual collection works as follows. The collection of patterns in the "*check*" field must be true for every binding of `var` to elements of the list `set`. In particular it must be true of the *first* element of `set`. What we wish to do is activate a sprite whose pattern is the "*check*" pattern with `var` bound to the first element of `set`. If this sprite ever triggers, we would like to activate the same sprite, but with `var` bound to the second element, and so on. If we succeed in triggering with `var` bound to each of these elements, then the whole sprite pattern (for which this is a replacement) is true, and we evaluate the body. Code to do this in Lisp machine Lisp is shown in figure 33.

While this is, perhaps, a bit complex to understand, an example will make it much clearer. Suppose we had the following sprite to replace:

```
(when {(∀ x  in   list
             check  {(less →x →number)})}
   (--body-to-evaluate--))
```

† Kowalski, in a talk given at the MIT AI Lab in Spring 1981, mentioned a similar augmentation made to his version of Prolog. He argued (as we do) that the declarative form is much easier to read and understand than the iterative code that implements it.

Fig. 33. Sprite Virtual Collection for Universal Quantification

```
(def-when-vca (∀ =identifier  in =list  check =sprite-clause)
   (let* ((funcname (symbol-append "FOREACH-HANDLER-" (gensym))))
     (make-auxiliary-functions
       '(defun ,funcname (element-list)
          (if (null element-list)
              (progn ,@*body*)
              (let* ((,identifier (car element-list))
                     (rest-elements (cdr element-list)))
                (when ,sprite-clause
                    (,funcname rest-elements))))))
     '(,funcname ,list)))
```

We use the function `make-auxiliary-functions` to create a new function with a unique symbol as its name. This new function embeds the sprite that we activate for each binding of the variable bound by the quantification. For the case above, the created function will look like:

```
(defun foreach-handler-1234 (element-list)
  (if (null element-list)
      (progn (--body-to-evaluate--))
      (let* ((x (car element-list))
             (rest-elements (cdr element-list)))
           (when {(less →x →number)}
             (foreach-handler-1234 rest-elements)))))
```

This function is compiled when the original sprite is compiled; that is the purpose of `make-auxiliary-functions`. The code that actually replaces the sprite looks like:

```
(foreach-handler-1234 list)
```

When `foreach-handler-1234` is called with argument `list`, it first checks to see if the argument is null. If it is, then the sprite pattern is vacuously true, and we evaluate the body. Otherwise we bind the quantified variable (in this case x) to the first element of the input and the variable `rest-elements` to the remainder of the list. We then activate a sprite containing the internal sprite pattern as its pattern. If this sprite triggers, we recursively call `foreach-handler-1234` on the rest of the list (bound to the variable `rest-elements`). We will reach a call with a `rest-elements` of `nil` iff the *check* pattern is true for all bindings of x in the argument list. This is the intended action.

There is a somewhat more general version of restricted universal quantification available. Suppose we had a list of random objects and wanted obtain a list of their lengths. We could created the following sprite:

```
(when {(∀ obj  in    list
                check  {(length →obj =n)}
                binding  list-lengths
                from-values  →n)}
  (random-function list-lengths))
```

As before, we check the collection of sprite patterns in the `check` field for each element of the *in* field bound to the variable `obj`. Additionally we can place an arbitrary expression to evaluate in the *from-values* field. For each element of the list in the *in* field, we evaluate the *from-values* field and add the value to the variable in the *binding* field.

If the sprite above triggers, meaning that we knew the lengths of all sequences that were in `list`, the variable `list-lengths` will be bound to a list of all of the lengths of these when we evaluate:

```
(random-function list-lengths)
```

The implementation of this extended form of universal quantification is essentially similar to the basic form of figure 33. It is presented without commentary in figure 34.

Fig. 34. Universal Quantification With Binding

```
(def-when-vca (∀ =identifier in =list check =sprite-clause binding =accum from-values =return)
   (let* ((funcname (symbol-append "FOREACH-HANDLER-" (gensym))))
     (make-auxiliary-functions
       '(defun ,funcname (element-list returns)
          (if (null element-list)
              (let ((,accum (reverse returns))) ,@*body*)
              (let* ((,identifier (car element-list))
                     (rest-elements (cdr element-list)))
                (when ,sprite-clause
                   (,funcname rest-elements (cons ,return returns))))))
     '(,funcname ,list nil)))
```

7.8.2 Interacting With Activites Using Sprites

In our examples, there were sprite patterns that triggered on conditions that activities could be in. We can, for example, create a sprite that triggers whenever a given activity, its argument, becomes stifled. By referring to the definition of an activity in figure 6.2.2, every activity has a slot `stifled-point-sprites`. As we will see, whenever a sprite is activated waiting for an activity to become stifled, a point sprite is placed on the list bound to `stifled-point-sprites`. The sprite replacement procedure for `stifled` assertions is shown in figure 35. The code that replaces the sprite first creates a point sprite. There is no *point-sprite-key* field because `stifled`-type assertions take no

Fig. 35. Sprite Replacement for Stifled Assertions

```
(def-when-vca (stifled =activity)
   '(slet ((point-sprite
             (make-point-sprite
               point-sprite-body         ',*body*
               point-sprite-closed-vars current-closed-variables
               point-sprite-closed-vals current-closed-values
               point-sprite-activity     current-activity)))
       (if (stifled ',activity)
           ;If the activity is already known to be stifled, eval the point sprite
           (point-sprite-eval point-sprite)
           ;Otherwise, add it to the stifled-point-sprites list.
           (structpush point-sprite (stifled-point-sprites ',activity)))))
```

parameters other than the activity that the point sprite ultimately gets hung off of. If the activity is already known to be stifled, the point sprite is evaluated. Otherwise it is placed on the `stifled-point-sprites` list to be saved in case the activity is stifled at some future time.

The `stifle` function contains the following section of code:

```
(foreach
  point-sprite
  (stifled-point-sprites activity)
  (point-sprite-eval point-sprite))
```

that causes already saved point sprites to be run in the event that the activity becomes stifled.

7.9 Comparison With Lexical Retrieval Schemes

This chapter has been about techniques for implementing a data-driven programming language. The basic concept of this sort of language goes back at least to Selfridge [56] who proposed programming by having *demons* watching *blackboards* and occasionally waking up and writing their own messages on the board. This concept has been developed in a number of directions. Ether belongs to a group of problem solving languages, often referred to as "Planner-like languages" reviewed in section 3.6. Another system that builds on this idea is the Hearsay architecture [12] in which the things written on the blackboard, rather than being derived facts and goals, are hypotheses about possible interpretations of fragments of speech.

In Ether, as we have already mentioned, assertions and sprites (our name for "demons") must satisfy the properties of monotonicity and commutativity. While this is not true of all Planner-like languages, it is arguably better because it allows one to give a declarative interpretation to sprites and assertions. The chief insight expressed in this chapter is that *the blackboard can be virtual.* We view the important properties of this style of programming that it be possible for use to give a declarative interpretation to the code. How this property of the code achieved in the implementation is of no consequence to programmers writing code. To this author's knowledge, all previous implementations of languages of this kind represent the blackboard of assertions and patterns of active demons using purely *syntactic* techniques. The examples in this chapter makes use of the *semantic* interpretation of the assertional forms to allow convenient coding and access. We will argue that implementing an assertional language using *virtual collections of assertions*, where the `assert` commands and sprites have been replaced by storage and retrieval mechanisms suggested by the semantics has a number of advantages. Because assertions must now *mean something* there is additional burden on the programmer to design virtual collections that are appropriate for the problem domain. Implementation is a two-tiered task. First classes of assertions and sprites must be built up and then the higher-level declaratively-interpretable code can be written using them. We believe that the extra work is justified and has the potential to raise this class of languages from the toy language status to languages of practical use.

Although all the virtual collections we have presented compiled into object-oriented style code, this is not a necessity. Other implementation styles might be more appropriate for other problems. Section 7.9.7 briefly describes another kind of virtual collection using unrelated implementation techniques.

The remainder of section 7.9 discusses various aspects of the implementation of assertional-style languages using virtual collections of assertions.

7.9.1 Finding The Optimal Route

No matter how one represents the content of the assertions, the process of accessing the information will occasionally be expensive. When the assertions are stored in a discrimination net, the programmer does not have the ability to decide the order in which the elements of the assertion are checked. Picking a wrong order can lead to an enormous computational waste. You will recall from the description of the implementation of `member` assertions in section 7.6 that we chose to represent memberness by storing on certain objects those things which are known to be members of them rather than storing on objects those objects representing lists that it is known they are on. This choice is important from an efficiency point of view. We are much more likely to as questions like "What elements are members of a given list?" than "What lists have this item as a member?" in the course of reasoning about programs. In other words we would be more likely to create sprites of the form:

```
(when {(member =x →list)}
  ...)
```

than sprites of the form:

```
(when {(member →element =y)}
  ...)
```

The second sprite will, as was described in section 7.6, expand into code that will send `when-member` messages to every object that has been or will ever be created. The first one will only send a message to one object. Clearly we have optimized the implementation of `member` assertions for our particular application.

Discrimination nets have similar properties with respect to variables in certain positions being much more expensive than variables in other positions. Without having control over the means of storing the assertions the much less efficient access path could have been picked as easily as the more efficient one.

7.9.2 Substituting Lower Level Reasoning for Higher

Having knowledge of the semantics of the assertions can sometimes allow results to "fall out" where otherwise extra computation would have to be done. For example, suppose we know that the number X is greater than 3; in other words we have executed:

```
(assert (> →X 3))
```

Also assume a sprite has been created of the form:

```
(when {(> →X 2)}
   --body--)
```

If retrieval is done *lexically* the sprite will not trigger because the atom 2 does not match the atom 3.

In the current implementation, greater-than assertions turn into sends to one of the objects that it is greater than the other. In the case that one is a constant object (e.g. 3) the message is sent to the other one. The object 3 is then put on the list of the `greater-property` of the object X. When the sprite is activated, it again recognizes that 2 is a constant and sends a `when-greater` message to X. The method that handles `when-greater` messages knows that 3 is greater than 2 and thus causes the body to be executed.

There are, no doubt, numerous places where some kind of reasoning can be done much more effectively using low-level implementation techniques and then interfaced to the high-level reasoner through a virtual collection of assertions. Section 7.9.7 presents a rather extreme example of this.

7.9.3 Reduction of Data

There are cases where many assertions can be reduced to one with a savings in both storage and computation by sprites wishing to access that information. Using the example of the previous section we can imagine that the following assertions have been made:

```
(assert (> →X 1))
(assert (> →X 2))
(assert (> →X 3))
```

If we stored these assertions lexically, and then created sprite:

```
(when {(> →X =n)}
  ...)
```

the sprite would be triggered three times, with n bound to 1, 2, and 3 on the three invocations. We know, however, that the knowledge stored in those three assertions can be summarized in remembering simply that X is greater than 3. If we stored only this, the sprite would only get triggered once with only the strongest possible binding for n rather than several superfluous ones.

7.9.4 More Complicated Queries

As we are not tied to items being lexically present in a database we can ask questions about things that were never explicitly asserted, but easily computed on the fly. We need only ensure commutativity and monotonicity. We had one example of such a sprite pattern, the restricted universal quantification discussed in section 7.8.1. This simple observation turns certain questions that have worried researchers in the field into non-problems. Much effort has been wasted worrying about whether certain kinds of reasoning should be done in *antecedent* or *consequent* mode. If we did not have the idea that the universally quantified form did not have to be lexically present in the database for a sprite referencing it to trigger, we would wonder whether we should always generated such things antecedently (i.e. generate them whenever they are true), an obviously silly idea, or invoke some sort of consequent reasoning which is inefficient and linguistically awkward. In the context of the Amord language, deKleer et. al. [72] worry about this very question with conjunctive statements. Bledsoe, in his excellent survey article on the use of a semantic approach in theorem proving [3], mentions a number of techniques that could be easily incorporated into a theorem prover using the virtual collections idea and lead to similar improvements of both efficiency and understandability of code.

7.9.5 Sprites Can Be Used Inside VCAs

The ability to to place sprites inside virtual collection handlers is of enormous significance in our ability to design efficient, understandable problem solvers. In the implementation of `member` assertions we were able to bury the sprite for checking for equality inside the message handler. Thus any techniques whatsoever that we have for managing equality can be buried inside a virtual collection for equality and the `member` virtual collection will function as we would like it to. We have used a very simple technique, but more sophisticated ones (e.g. Bundy [6]) could as easily have been incorporated. To understand the advantages of this, both in terms of simplicity of code and efficiency, it is instructive to consider what we would have to do to get proper behavior with respect to `member` and `equal` assertions using a lexical retrieval mechanism. The simplicity with which it is achieved using our mechanism was discussed on page 143. We wish to ensure that if the following two assertions were made:

```
                    (assert (member →an-element →a-list))
                (assert (equal →an-element →another-element))
```

and we activated the sprite:

```
(when {(member →another-element →a-list)}
  ...)
```

the sprite would trigger, regardless of the relative orderings of the two asserts and the sprite activation.

One way we could get the appropriate behavior is by creating antecedent sprites:

```
(when {(member =obj =list)}
  (when {(equal →obj →other-obj)}
    (assert (member →other-obj →list))))
```

This would generate lots of unwanted assertions that would create what is often called a "combinatorial explosion." The other possibility is to complicate the `member` request by including the `equal` request inside of it. Thus, if we wanted to know if `an-element` was a member of `a-list`, we would have to create the sprite:

```
(when {(member =obj →a-list)
       (equal →obj →an-element)}
  ...)
```

which is unmodular and doesn't give us the ability to have special ways to encode equality. Note also, that without more system-supplied syntactic mechanisms, the above code won't actually work because

```
(equal →obj →an-element)
```

is actually *not* the same request as

```
(equal →an-element →obj)
```

This is why languages like QA4 [54] have invented notions like *bags* (collections without order) so the `equal` pattern need not be duplicated. By giving the user the ability to semantically encode information such syntactic mechanisms are not needed.

7.9.6 Distributability and Parallelism

There has been some recent interest in the design of multiprocessor architectures to execute message passing languages [17, 27]. Implementations built on lexical retrieval schemes require there be a single, monolithic database in which to store the assertions. The compilation of Ether sprites using virtual collections of assertions results in an inherently distributed implementation that could be executed on such an architecture if they become feasible. See also section 8.3.3 that discusses this issue further.

7.9.7 An Alternative Virtual Collection

All of the examples actually implemented that made use of the notion of virtual collections of assertions made use of a compilation into object-oriented style of programming. While this is a powerful technique, other means of storage are possible. We discuss one (unimplemented) virtual collection that would be useful for implementing an Ether algorithm described in [30]. It is not essential to go through the details of the problem or the algorithm. The assertions and sprite patterns being replaced are of the form of predicates on arbitrary subsets of a finite set whose elements are known in advance.

In many problems the number of elements in the set can be as high as 10 or 15, allowing the size of the space of possible assertions that could be made to grow quite large. One of the predicates has the property that if it is true of a set, it is also true of all supersets of that set. The other predicate has a dual characther -- if it is true of some set, is is also true of all subsets of that set. A rather high density of the sets will ultimately get marked by one of the two predicates. (They are mutually exclusive.) In comparison, the number of sprites that will be created is low and always asks about *specific* subsets. This information is useful to us in the design of the virtual collections.

Because the space of possible assertions is the powerset of a set that may be fairly large, and a high density of them will actually be asserted, it might make sense to represent the asserted subsets in a bit table. A trivial algorithm addresses the correct bit: the index into the bit table is a bit string whose length is the total number of elements in the set. For a given set, the index is computed by using a 1 in the designated position for each element that is in the set, and 0 otherwise. When one of these predicates on a set is asserted, the algorithm is run to set the appropriate bit in the table. Depending on which of the two predicates we are considering, we choose one of two simple algorithms that enumerate the bit table addresses for either the supersets or subsets of the initially asserted set.

Similarly, there can be two tables of bits where each bit indicates that a sprite has been activated looking for the particular subset. The number of sprites activated at any one time will be relatively few, so the actual point sprites can be stored in a simple list or other data structure without great concern for optimization.

With this "low level" programming out of the way, code that uses sprites and assertions can be written with ease. It is worthwhile considering how intractable it would be to write such a program using sprites and assertions where the implementer is not able to design a virtual collection of assertions. First, the assertions would be implemented by lexically storage. The amount of storage (per assertion) is relatively high -- no matter how the assertions are represented there is a minimum of n pointers required, where n is the number of elements in the subset. Schemes such as discrimination nets and hash tables that are designed to speed up queries inevitably require more storage than this minimum. Since the density of sets for which assertions will be made is quite high (probably greater than .5 on the average), the difference in storage requirements is of overwhelming significance.

Another implementation requirement if a standard lexical retrieval scheme is used is that the marking of supersets or subsets of marked sets must be marked by *sprites* rather than by more efficient low level operations. Here again, the efficiency loss is so significant that it might well spell the difference between practical and impractical programs.

Chapter VIII Epilogue

In this chapter we collect together several topics that relate to the material of the previous chapters but do not properly fit inside them.

8.1 Nondeterminism and Computation

A very surprising thing happened while running the programs used in the cryptarithmetic problem solver described in chapter 4. Many of the problems tested admitted more than one solution. During different runs of the problem solver on a single problem, different (valid) solutions would occasionally result. This was a surprise because Lisp programs do not normally do such things, that is return different results during different runs. There are exceptions to this, programs that are designed *explicitly* to give different results each time. Such programs contain random number generators, or perhaps sample a value outside of the Lisp environment to be used as a datum by the program (e.g. the real-time clock). The parallel Ether program is *nondeterministic* and this realization has important implications for our conception of what it means for a system, particularly an "intelligent system," to be nondeterministic.

This discussion is in part a critique of two views prevalent in the literature. According to one view, intelligent mechanisms must contain random choice points if they are to produce the nondeterministic behavior that people seem to exhibit. Proponents of the other view argue that intelligent behavior must be deterministic because putting random choice into a computer program cannot possibly make the program "more intelligent." While both points of view have some justification, we will argue that the apparent disagreement arises from a limited understanding of the sources of nondeterminism. We use our Ether programs as a demonstration that intelligent mechanisms may very well exhibit nondeterminism *without* the random choice points that the proponents of the second view (quite rightly) find objectionable. We first develop these two positions by a dialogue between two imaginary individuals.

Mark: Good morning, Barbara.

Barbara: Good morning. Say, you promised you'd explain to me your new musical improvisation program. Now might be a good time.

Mark: Why, yes, I'd like to. Have you ever noticed that if you take pieces of music written in a single style that the various attributes of the pieces, say their pitches and durations, have a characteristic distribution? In the simplest version of my program, I can emulate any composer by processing a corpus of his works and deriving statistical distributions for the pitches and durations. I can then use these tables to "play" in that style by randomly determining a sequence of these notes that conforms to this distribution.

Barbara: Well, certainly *that* isn't music. A random sequence of notes must surely sound like garbage! I would liken it to "monkeys sitting on piano stools." The ingenuity of a composer is exhibited in the careful and thoughtful arranging of the notes, not in some overall distribution of the pitches and durations.

Mark: You're jumping the gun. Indeed it does sound like garbage; but remember, this is just the beginning of my theory. There's more to it.

Barbara: Please continue.

Mark: Actually, what I just explained to you is my "zeroeth-order theory." The "first-order theory" involves creating statistical distributions for all pairs of notes. The synthesis algorithm, when it wants to pick the attributes for note n, uses note n-1 to define the distribution for the selection. The second-order theory looks at a corpus of data collected about occurences of triples and selects the distribution for note n from notes n-1 and n-2. The higher-order theories are similarly constructed.

Barbara: Well, it sounds like you've been spending long nights typing tables of numbers into the machine. I think your time would have been much better spent thinking about the structure of music and how that can be incorporated into an algorithm. I don't understand how randomly picking notes could make something sound remotely like a piece of music worth listening to.

Mark: I'm not just "randomly picking notes." The selections are made according to a carefully calculated statistical distribution.

Barbara: Even still, it doesn't reflect the careful selection of notes based on a high-level understanding of music that my program uses. [Barbara goes on to describe her object-oriented composition program.]

Mark: Ah, but there is a sense in which your program can't be modeling what a composer does. It will always generate the same piece! Human beings are clearly nondeterministic. The same composer will never generate the same piece twice.

Barbara: No! The only reason the composer does not generate the same piece twice is because he remembers generating the first one. It would be boring to generate the same thing twice. Anyway, there is a simple demonstration, or "proof" if you like, that intelligence is deterministic.

Mark: A proof! This is getting interesting.

Barbara: It's really just common sense. Suppose we have a program that randomly chooses things at some point. If we can find a piece of knowledge that suggests one choice is better than the others we can use that knowledge to make the choice deterministically instead. The intelligence exhibited by the program has to increase. If we have no such piece of knowledge then we will do something like always pick the first one on the list. Surely this can't make it less intelligent.

Mark: Well, that sounds all well and good; but I still claim that you aren't adequately accounting for the nondeterministic behavior people obviously exhibit. I think I can demonstrate this to you. Do you play chess?

Barbara: Yes I do, but I'm not that good at it.

Mark: That's fine. [He takes out his chess set and sets up the pieces. He lets Barbara be white.] I'd like to play you a game to demonstrate a point. [Unknown to Barbara, Mark is a chess expert. He chooses, however, to lose the game to Barbara to make his point. The game took about 20 moves.]

Barbara: Checkmate! I believe you were trying to make some point or other?

Mark: To make my point I will have to play you another game. I want you to play the same game you just played, and I will do likewise. Certainly, for a deterministic machine such as yourself, that should be no problem! [They play the game again. Barbara concentrates intensely. On the fourth move Barbara makes a different move than she did the previous game.] You goofed, Barbara!

Barbara: No, I couldn't have. You just threatened my king's pawn and I must have protected it then as I did now.

Mark: Well, you did protect it, but not by P-Q3, you moved this knight up instead.

Barbara: I'm not sure I believe that.

Mark: Well, it just so happened I remembered the game. [He resets the pawn, makes the knight move, and the finishes the game with a running dialog explaining what he thought were Barbara's reasons for making the moves she did.]

Barbara: Well, I admit that ending looks familiar. Well, you're right that I played a different game the second time. But I still claim its not because I'm a nondeterministic machine! I must have learned something or gotten some new idea that made me think the pawn move was better than the knight move, and this caused my "chess program" to do something different.

Mark: That isn't likely. It takes years of research for anyone to learn anything at all about the first few moves of the chess game. Besides, you were *trying* to play the same game as the time before. Face it, somewhere in your "chess program" there is a random choice point that picked one move the first time, and a different move the second time.

Both of the characters in our little dialog have a point. Mark's is based on the simple observation that people *don't* do the same thing every time. Barbara's argument is based on the intuitively satisfactory premise that a deterministic choice being used instead of a nondeterministic choice can't lower the apparent intelligence of an algorithm. (We ignore here certain special uses for random choice such as Monte Carlo simulations and in the theory of games.)

Before continuing with a more abstract discussion of sources of nondeterminism, it is important to understand just why Ether programs are nondeterministic. The running program contains numerous running *activities.* The collection of activities is very fluid; new activities are created and others are stopped all the time. At the lowest levels of the implementation is a scheduler for events and activities described in section 6.2. The main loop of the scheduler selects each current activity in turn and runs computations from its event queue until a predetermined amount of time chosen in proportion to the amount of processing power the activity has. It then continues to the next activity. Nondeterminism creeps in because the number of events that can be processed in a given time quantum by an activity varies. There are a number of reasons why this can happen which we will enumerate:

1. The Lisp machine normally services interrupts from various sources. It is connected to a network and occasional messages are received by the Lisp machine to be processed. The receipt of these messages happens unpredictably. Another source of interrupts that can occur is the pointing device (known as a

"mouse"). If it should be moved (or even just jiggled) it will generate an interrupt that must be handled. The handling of these interrupts, when they occur while an activity is "current," will decrease the number of events that the activity will be able to process.

2. Another source of this variation is the memory configuration of the machine. The Lisp Machine, like most modern computers, has a *virtual* memory configuration. Programmers think about their machine as if it had a very large quantity of homogeneous memory. In reality, there are several different kinds of memory. On the Lisp machine there are essentially two kinds: a relatively small amount of random-access semiconductor memory and a large disk. While the machine is running information can only accessed if it is residing in the random-access memory ("RAM"). If the information does not happen to be in RAM it must be read in from the disk, a very time-consuming operation. If alot of disk operations happen to be done during time the time quanta given an activity by the scheduler, very little actual work will be done. How much time is wasted handling page faults is something that is very hard to model or make repeatable. This reason also interacts with reason (1) above; Handling of interrupts can cause paging and cause a rearrangement of the page set and of Lisp objects on those pages that will affect future paging behavior.

3. Yet a third source of variation is due to the programmer's desire to probe the program in ways that are useful for debugging it. For purposes of debugging it may be necessary to have the program do certain additional operations at certain points. I have implemented a number of different tracing modes in Ether that cause certain information to be printed on the console or different statistics to be gathered. Sometimes changing the precise features we are tracing will cause different answers to result.[†]

The effect of all of this is that different activities may progress at different rates with respect to one another on different runs. One can think of activities in the cryptarithmetic program as "competing" for processing power by attempting to constrain quickly. Two activities may be both heading to valid solutions and if one, because of the reasons mentioned above, is able to get a slight edge, the `manager-activity` will notice this fact and assign it more processing power. This will greatly increase the likelihood of it reaching a solution before its competitor.

How can we relate these observations to the paradox suggested by the dialog? The two characters in our dialog believed that they were taking incommensurable positions. They believed that if the *program* did

[†] This has on occasion greatly frustrated the debugging effort.

not contain random choice points then it must be deterministic. Our example illustrates that nondeterminism is possible *without* the programmer having to place any random choice points into the program. One might argue that the random choice points are *really* there, that if we were to model the interpreter for our program at a fine enough level of detail we would find these random choice points. While this objection is true in principle, in practice such a model would contain so much irrelevant detail (e.g. where Lisp objects are in real memory, what pages are in core, the current confguration of the memory map, etc.) that it would not be, even remotely, a tractable model we could use for understanding the program as a problem solver. Thus nondeterminism becomes not a property of the interpreter itself, but of *our model of it.* Clinger [7] develops this argument extensively in his semantics for (nondeterministic) actor computations. In our problem solving systems we have nondeterminism *not* because we were too lazy to make an intelligent choice when presented with several options, but because there are so many details that we would be forced to think about which are irrelevant to our purpose in writing the program.

8.2 A Comparison With Constraint Networks

The ideas expressed in chapter 7 concerning the implementation of assertions and sprites using virtual collections of assertions emerged from a series of discussions with Luc Steels and Ken Forbus after the publication of my first work on Ether [30]. They had both become proponents of the *constraint network* metaphor for problem solving used by Waltz [66] and more recently pursued by Borning [5] and Sussman and Steele [59]. I had not yet learned that declarative (sprite) code could be written that did not involve lexical retrieval schemes. Steels and Forbus argued that it made more sense to associate facts known about objects with the objects themselves and that all computation should involve *local* interactions between objects. This was necessary so that system performance would not degrade as more knowledge is gained (as happens with lexically-stored assertions). I accepted their criticism as valid, but countered with what I felt to be the greater expressibility and computational generality of sprites. Virtual collections of assertions evolved as a way of keeping the advantages of sprites while exploiting the locality of interaction inherent in the notion of constraint networks.

In this section we make clear the relationship between the two programming metaphors. We emphasize two points:

1. A constraint network can be constructed out of sprites with the following limitation: *All sprites are*

activated before any are run. In other words you cannot have sprites that create other sprites. We believe this puts a significant limitation on the expressiveness of the language.

1. Using sprites (with virtual collections of assertions), code that is expressively equivalent to constraint networks will compile into code that is functionally equivalent. In other words, there is no efficiency penalty for programming with sprites.

We will make these points by building part of a constraint system out of sprites for a domain of interest to workers with constraint networks -- simple linear electronic circuits; in fact, to make our point, we can restrict our discussion to just networks consisting of nodes and resistors. The system has not been implemented, but an implementation would seem to present little difficulty.

The first question we will ask ourselves is: "What is a node?" A node is a set of device terminals at the same electrical potential and that satisfy Kirchoff's Current Law. The terminals all connect to the node, the voltages at each of the terminals are equal, and the sum of all currents coming into a node must be zero. Using the metaphor of constraints this becomes functionally an object which knows about N other objects (the device terminals). If the currents of any N-1 of them are known, then the current of the Nth one can be asserted.

A node with 3 wires coming out of it would be defined by sprites something like those in figure 36.

Fig. 36. Three Terminal Node Sprites

```
(when {(current-out-of T1 =I1)
       (current-out-of T2 =I2)}
  (assert (current-into T3 (+ I1 I2))))

(when {(current-out-of T1 =I1)
       (current-out-of T3 =I3)}
  (assert (current-into T2 (+ I1 I3))))

(when {(current-out-of T2 =I2)
       (current-out-of T3 =I3)}
  (assert (current-into T1 (+ I2 I3))))

(when {(potential-at T1 =E)}
  (assert (potential-at T2 E))
  (assert (potential-at T3 E)))

(when {(potential-at T2 =E)}
  (assert (potential-at T1 E))
  (assert (potential-at T3 E)))

(when {(potential-at T3 =E)}
  (assert (potential-at T1 E))
  (assert (potential-at T2 E)))
```

These represent the relevant "electrical facts" and are similar in form to rules that would be defined in a constraint network.

A resistor is a two-terminal device with three parameters, the potentials at the two terminals and the current flowing through the device. If two of these parameters are known the third can be computed. It is also the case (as for all two-terminal devices) that the current going into one terminal is equal to the current going out the other terminal. Sprites that express the relevant electrical facts about resistors are shown in figure 37.

Fig. 37. Resistor Sprites

```
(when {(current-into T1 =I)}
  (assert (current-out-of T2 I)))

(when {(current-into T1 =I)}
  (assert (current-out-of T2 I)))

(when {(current-into T1 =I)
       (potential-at T1 =V)}
  (assert (potential-at T2 (- V (// I R)))))

(when {(current-into T2 =I)
       (potential-at T2 =V)}
  (assert (potential-at T1 (- V (// I R)))))

(when {(potential-at T1 =V1)
       (potential-at T2 =V2)}
  (assert (current-into T1 (// (- V1 V2) R))))
```

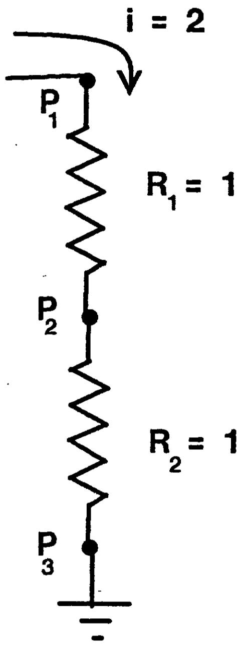
The sprites that define each of the kinds of devices can be collected into procedures named `create-node` or `create-resistor` so they only have to be written down once and then instantiated any number of times for each of the components of the circuit. What we end up with looks very much like the constraint network formalisms of Borning, and Sussman and Steele.

An example will show how sprites such as the ones shown in figures 36 and 37 could be used to solve a simple circuit problem. Suppose we had a circuit such as the one in figure 38 and were interested in knowing the potential at node P_1. How would this be determined?

The current going into upper terminal of resistor R_1 is 2, this will trigger off a sprite from figure 37 of the form

```
(when {(current-into T1 =I)}
  (assert (current-out-of T2 I)))
```

Fig. 38. Series Resistors



That will transfer knowledge of this current to node P_2. P_2 is a two-node (like the three-node in figure 36 but simpler) which will contain a sprite indicating the current into the node from one terminal is equal to the current out of the node on the other terminal. Another sprite like the one mentioned above will transfer knowledge of this current to the terminal of resistor R_2 adjacent to the node marked P_3. At this terminal we now know the current (2) and the potential (0, because it is connected to ground). This will trigger a sprite of the form

```
(when {(current-into T2 =I)
       (potential-at T2 =V)}
  (assert (potential-at T1 (- V (// I R)))))
```

to mark node P_2 with a potential of 2. The analogous sprite for resistor R_1 will then trigger propagating a potential of 4 to node P_1. This was the original question we were interested in knowing and a sprite would wait for the answer to appear and then stifle the activity containing all these sprites.

We now will address the second of our two claims above, that of efficiency. Since we will be implementing the sprites and assertions using virtual collections of assertions, a reasonable strategy will be to make terminals the *ether objects*. You will notice that each of the assertion types mention a terminal and none of them are ether variables, making this a very good choice. What, then, does a sprite of the following form:

```
(when {(current-into T2 =I)
       (potential-at T2 =V)}
  (assert (potential-at T1 (- V (// I R)))))
```

compile into? The first sprite pattern will compile into a command to place a point sprite on the object representing terminal T2. The point sprite watches for information to be learned about the `current-into` the terminal.† If this triggers a new point sprite will be created and attached to the same node to watch for a potential being learned. Upon learning this information we evaluate the code that results from compiling:

```
(assert (potential-at T1 (- V (// I R))))
```

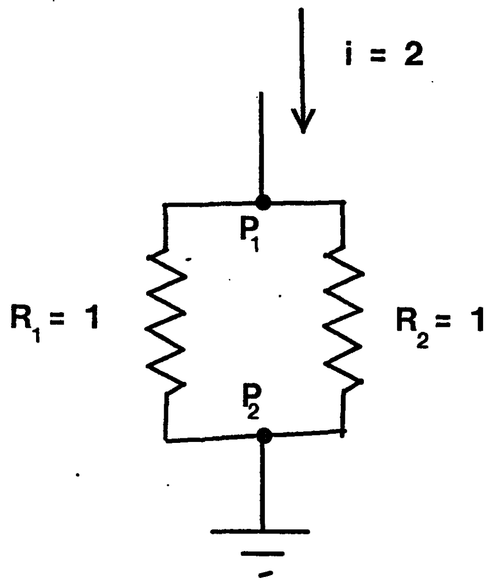
This is a simple message transmission to terminal T1 that its voltage is a given value. At each point in this execution, computation involves only local computation at specific nodes. No search through a database is required anywhere. The only computational overhead required that might not be found in the most efficiently implemented constraint network is that necessary to create the point sprites and implement message passing. This is a very insignificant amount of work. We conclude that *sprites compile into code essentially equivalent to a "hard-wired" constraint network implementation.*

To demonstrate the potential power of using sprites we examine another circuit in figure 39 that (to electrical engineers) is only a little more complicated than the one in figure 38. Here we know two facts. We know the current going into node P_1, and the voltage at node P_2. By examining the sprites that implement the three-node (of which node P_1 is an example) we discover that nothing can be determined about the currents entering the terminals of the resistors. By examining the sprites in figure 37 it will become apparent that nothing new can be learned. For any of these sprites to fire, we require knowledge knowledge of two of the three device parameters (potential at each of the two terminals, and the current through the resistor). We only have knowledge of one of these facts (the potential of the grounded node) for both. Our system quiesces without solving the problem.

An engineer would solve this problem by realizing that the configuration of resistors in figure 39 is one

† We made use of 2 classes of assertions that referred to the currents through a node. They were **current-into** and **current-out-of**. The two assertional types were used for notational and conceptual convenience. In the virtual collection of assertions implementation, only one of them would be stored and questions/assertions about the other would be translated into questions/assertions about the negation of the first. Here, again, the use of virtual collections of assertions allows us to write simple declarative code without any efficiency penalty.

Fig. 39. Parallel Resistors



for which he has special knowledge. They are known as *parallel resistors* and he has a special rule that defines the v-i characteristics of this configuration. The rule states that the configuration acts like a single resistor with a resistance equal to:

$$(R_1 * R_2) / (R_1 + R_2)$$

To make use of this knowledge, we must be able to create a sprite that watches for parallel configurations and then creates new sprites that express this new knowledge.

8.3 What We Need To Make Faster Ethers

We believe that the Ether language shows promise as a practical tool for program development for artificial intelligence, even as a language to be run on a single processor implementation. Much of the discussion in chapter 6 is to convince the reader that the overhead for having a *parallel* language is relatively small. Never the less, there are places where improvements in performance can be gotten.

8.3.1 The Activity Structure

There is nothing inherently inefficient about the notions of activities and events. The kernel of the implementation that cycles through activities and through events on the queues of each activity is small and quite stable. With some specially written microcode the overhead of running events need only be a couple of times the overhead of doing a Lisp function call.

Ether takes a less radical view towards message passing than Act1. In Act1 all computation is done by message passing down to a very fine level.† Whenever a message is sent that is a "request", i.e. corresponds to a function call where the return value is of importance to the caller, ACT1 must CONS an explicit continuation actor to handle the reply. This is the function normally supplied by the Lisp stack without requiring any CONSing to be done. In Ether, there are many function calls that happen when executing each event. Each one of these function calls is implemented as normal Lisp. Thus we don't concur an efficiency loss. This is not to say that we do not lose something in linguistic power by not having explicit continuations. See section 8.4.3. However the current Ether may be a reasonable compromise with Lisp systems that are likely to be available in the near future.

8.3.2 Eliminating Shallow Binding

Each of the events in the events queue is a *closure*, a piece of code to evaluate in an environment. Currently closures in the Lisp machine are implemented by *shallow binding* all variables in the closure. This procedure means that, upon entry into the closure, the entire vector of variables must have the values currently in their value cells save, and the new values for these variables placed in the value cells. Upon exiting the closure the saved values must be restored. In the Lisp machine this procedure takes approximately 40 microseconds per variable.‡ In practice the environments that accompany the events can be quite large (30 or more is not uncommon). The procedure contained in any given closure rarely refers to more than a couple of the variables in the environment, and frequently is quite short. This means that a large fraction of time is spent in overhead related to the environment.

† This is also true of Smalltalk [29]. Yet Smalltalk does not admit parallelism, obviating the need for CONSing continuations or maintaining a queue of events.
‡ Personal communication. Timing tests performed by Gerald Barber.

8.3.3 Improving the Performance with Multiprocessors

Most of this work has concerned itself little with the questions of concepts of parallel languages *irrespective* of the nature of the hardware the program is to be run on. This is an unsettling concept to many people who prefer to view parallel computation as an extension of sequential computation. Following this paradigm, the programmer writes several sequential programs that communicate with each other in some manner. A common means of intercommunication is by means of critical regions in which variables shared by all processes. Programs that make use of shared variables are harder to understand than the sequential programming methodologies they are based on.

The metaphor for parallelism Ether is based on is the actor model of computation. Under the actor model, all computation is the result of *actors passing messages*. When an actor receives a message, the actor may respond by sending one, none, or several messages. Concurrency is the result of an actor sending more than one message when it receives a message. Under the actor model, concurrency is the default and sequentiality the special case where each actor sends at most one message when it receives a message.

Our arguments for parallelism are based primarily on ease of control of search processes. That there is concurrency that is potentially exploitable on multiprocessor systems is a valuable result, though one not essential to this thesis. The presence of true concurrency in the programs follows from the use of virtual collections of assertions as described in chapter 7. Rather than having a global database of assertions that would present a bottleneck to parallel programs that have to constantly access assertions in it, we have a system of objects with *local state* that communicate by sending messages. The potential hardware concurrency is enormous. Hewitt and others [27] are currently investigating techniques for running parallel actor programs on systems of homogeneous processors. Progress made in that area is directly applicable to Ether.

There are some concepts contained in Ether that are new to message passing systems. These involve the notion of an *activity* which is a mechanism for collecting events into convenient groupings so that resource control becomes possible. We envision a multiprocessor implementation of Ether to look much like many single processor implementations that communicate with one another. In other words, each processor would have its own ring of activities and associated event queues. There are, however, two concepts that we have been making use of the implementability of which come into question on multiprocessor machines. We will briefly discuss these two concepts.

1. **Quiescence.** An activity is *quiescent* whenever it has nothing to do. In the single processor implementation, quiescence is determined to have occurred when the event queue of an activity is empty. In a parallel implementation, quiescence becomes a much more difficult property to detect. An activity is quiescent if there is no message which is part of that activity in *any* queue.† This might seem to be a hard occurence to detect, and it is. An algorithm due to Lamport [38] can be adapted to solve this problem, though the overhead would be considerable. There are two possible outs from this dilemma:

(a) There are only certain activities for which we would like to retain the ability to determine whether or not they are quiescent. If, for any given activity, we restricted the number of processors that were allowed to handle messages within it, we would then be able to reasonably (and reliably) detect quiescence. The degree to which we would be willing to do this would depend on the inherent concurrency within each of these activities and how many of them we would like to be running at the same time.

(b) The detection of quiescence should only be used for heuristic purposes. That is, the correctness of a deduction should not depend on whether or not an activity has quiesced. It is reasonable, then, to have a mechanism that can detect quiescence with reasonable probability. If it turned out to be wrong, processing power allocations could be rearranged as they were before.

2. **Processing power.** In the single processor implementation there was a simple scheme for computing the time quantum for each activity from the amount of processing power assigned to the activity. This algorithm does not directly carry over to a distributed implementation because each processor is unlikely to have work under every existing activity to do at all times. Thus activities which tend to be more "spread out" among the processors will get more than their share of processing power, to the detriment of activities localized on a small number of processors. Algorithms can be designed to compensate for this, however. We envision each activity having a "manager." The manager keeps track of the total amount of computation (throughout the whole network) that the activity has done. Reports can be sent periodically by individual processors to the manager with this information. This total is then compared with the amount of processing the activity should have gotten during this period (easily computable as in the sequential case). If the two are the same, the manager does nothing. If the activity has gotten more

† Depending on the specifics of the design of the multiprocessor system, we may have to additionally ensure that no messages are caught in transit. The Lamport algorithm can be adapted for this case too as long as the time a message can be in transit is bounded.

processing done than it should have, the manager sends messages to the processors instructing them to lower the amount of processing power the activity should be considered to have. If the activity has gotten less processing done than it should have, a message informing processors of an increase in processing power is sent.

8.4 Problems With Ether

There are several problems with the current Ether language that have limited its usefulness. We review the more obvious of these.

8.4.1 Inadequacies With Viewpoints

Viewpoints are *a mechanism for creating different and not necessarily compatible models inside the machine.* Other names used in the literature for a similar concept are *context* and *situation.* We avoided the term context because it carries with it the notion of a "current" context. With viewpoints there is no concept of a current one. Context mechanisms do not admit significant parallelism because there is usually a non-trivial amount of work required to switch contexts. The term *situation* was introduced by McCarthy [43] in the context of a logic, the "situational calculus." McCarthy's definition of a situation was limited to differentiating states of the world at different times. Our use of viewpoints has been throughout this work for creating *hypothetical* models, although we do not exclude other uses. We would like to be able to use viewpoints for modeling changes of state (as with situations), personal points of view, belief systems, etc. Barber [2] investigates more varied viewpoint mechanisms.

The current notion of inheritance between viewpoints is severely limited. When a viewpoint inherits from another, it either inherits all the facts or none. We would like more flexibility than this. One of the first AI systems to make use of viewpoint-like mechanisms was the STRIPS [13] system. In STRIPS one viewpoint was related to another by *addlists* and *deletelists.* One viewpoint could be converted to another by adding a set of assertions and deleting some others. The STRIPS mechanism is quite limited. New facts learned by reasoning about facts that are to be deleted, will not themselves be deleted. Thus the STRIPS mechanism is inadequate for our purposes. A much improved mechanism has been explored by Doyle [11] and others. The good idea in his work is that *justifications* should be maintained for each of the facts in the database. If a fact is deleted, those derived facts that require it as a basis are deleted. Doyle couched this mechanism as a *truth maintenance* system. Facts that are no longer believed become invisible; thus only one "viewpoint" is accesible at a time -- an inherently sequential approach. We believe justifications can be used to link facts between several viewpoints. A new viewpoint can be defined that inherits all facts from another except those that depend on a specific fact. We can make use of the logical flexibility of justifications without giving up the ability to process many viewpoints simultaneously.

The current Ether handles viewpoint inheritance by creating a *viewpointed object* for each object and viewpoint where some information is known about the object with respect to that viewpoint. When a new viewpointed object is created, *all* information is copied from the parent viewpoint(s). This copying can sometimes be expensive (although it is conceptually simple to implement). A more efficient mechanism can be envisioned that *transforms* questions about the presence of a fact in a given viewpoint into queries of inheriting viewpoints. Such mechanisms should be investigated.

8.4.2 Lack of Full Continuations

Since Ether is not implemented using full actors (with continuations) there is a certain awkwardness with mixing function-type code with sprite-type code that could be avoided otherwise. For example, suppose (in the program synthesis system) we wished to to determine the length of a list L. We would say in the current system:

```
 ...
(let ((act (new-activity)))
  (goal (length L) act)
  (when {(length →L =num)}
    ...))
```

We could *not* however define a procedure `list-length` that returns the length of the list. This is because the mechanism that locates the list's length (the sprite) does not obey the normal Lisp stack discipline. The code that is the replacement for `when` returns immediately after creating the sprite, losing the path through which the information is to be returned. Assuming we could pass continuations, the code for `list-length` would look like:

```
(defun list-length (L)
  (let ((act (new-activity)))
    (goal (length L) act)
    (when {(length →L =num)}
      (<- list-length-continuation num))))
```

We would then be able to use this function as we would use any other.

We were able to get around this problem by "open coding" all such procedures (as shown above). Nevertheless, the ability to have procedures with full continuation-passing allowed would be an enormous convenience.

8.4.3 Lack of True Lexical Scoping

Ether, though lexically scoped, is built on top of Lisp which is not. This has caused us some difficulty as we discussed in section 6.3. Implementing Ether in a full actor language, or a Lisp that is lexically scoped such as Scheme [60] would eliminate this difficulty. Steele [61] has developed a compiler for Scheme that would make such an implementation quite efficient.

8.4.4 Quasi-quote Syntax

The normal convention in writing Lisp code is that items appearing in the argument positions of evaluated forms are normally evaluated. If we want the literal item as the argument (rather than its value) we precede it by a quote mark. In Ether we have picked the opposite convention. Items that are unmarked are quoted while those that are marked (by the symbol "→") are evaluated. This convention was of some use to us in the design of the program synthesis system. For example, we were able to use expressions of the form:

```
(assert (termination-clause ((equal →sequence →nill) →accumulant)))
```

However, in most cases, every assertion consisted of a single relation symbol followed by several forms, each of which was evaluated. Thus in most cases the syntax was overly tedious. I would recommend that future implementations of Ether or similar languages stick more closely to the Lisp conventions in this regard. As experience increases with languages of this sort, there will tend to be even less uninterpreted syntactic forms than exist already.

The reasons for using the quasi-quote notation are largely historic. The current implementation of Ether evolved from an older implementation that did not have the concept of virtual collections of assertions. Instead, when assertions were made, the *literal* piece of list structure was encoded into the database. Thus the objects comprising the assertions did not have any procedural structure and could be served by literal (quoted) symbols.

In the current Ether, all objects have internal structure and thus must be created and then passed via variable evaluation. Thus all positions inside relations (assertions and sprite patterns) are variables to evaluate. The Lisp convention is far more convenient (as it is with normal Lisp code).

8.4.5 Lack of Unevaluated Functions Inside Relations

The syntax we have been using for assertions and sprites is restricted in a way that logic is not. Logic allows the mixture of *functions* with *relations* while we allow only relations. This can be seen in a certain awkwardness in our syntax. If we wanted a sprite to trigger on the condition that the first element of a list X is a member of the list Y, we would say in our notation:

```
(when {(sequence-element →X =el 1)
       (member →el →Y)}
  ...)
```

A much more concise way to say this would be:

```
(when {(member (car X) Y)}
  ...)
```

Here car is a function that extracts the first element of the list. It would be an enormous convenience to us to be able to have the sprite involving the car function in some manner expand into something the system can understand directly. A general theory of how to handle functions in Ether would be very powerful.

Resolution theorem proving (and Prolog) allow the use of functions inside relations. On the surface, then, we appear to have an impoverished formalism. Resolution, though, treats functional symbols *syntactically* and the kind of power you would hope to get out is lacking. Our system understands the semantics down to a deep level. So, for example, if we knew that X had a length of 1 and that a certain element was a member of it that was also a member of Y, our Ether sprite would be able to trigger. A resolution-based system would probably not be able to perform such a deduction. It would look for a literal (car X) to unify with and not find it.

8.4.6 Use of Class Structuring Mechanisms

The object-oriented formalism we used to represent both of our systems involves no class hierarchy. For example, in the program synthesis system, we had one kind of programming object that could be "configured" to look like a sequence, atom, or number. This one object must contain enough instance variables to represent all possible relational types about all of these. When we create objects we frequently know at *object creation time* which kind of object it is. Thus, in addition to having a general object type, we could have subclasses of this that were specifically for sequences, atoms, or numbers. If we knew already the kind of object it was, the information could perhaps be represented more efficiently by special message handlers for these subclasses. We could further subclassify our objects. For example, we could write special message handlers just for the `nill` object. The response this object would have to `when-member` or `when-sequence-element` messages would be to simply disregard them! The `nill` object cannot possibly have members and so the sprites could never be triggered and there is no need to create a point sprite. Such a class hierarchy would considerably improve the efficiency (and, perhaps readability) of the virtual collection code.

Chapter IX Conclusions

This work makes several contributions towards the art of problem solver construction. We list them.

1. We have demonstrated an approach towards *engineering design*, the design of a system that satisfies specifications. While the specifications were logical in character the process of doing the actual design was quite distinct from any theorem proving. Instead, the specifications were used in two ways. They were used as integral parts of *proposers* that suggest solutions or classes of solutions to the problem. The specifications are run on simple examples to suggest solutions. The specifications are also used to *falsify* solutions or whole classes of solutions that have been proposed. Parts of the program synthesis system are admittedly *ad hoc* but the over all design is suggestive. Many of the ideas could be used to improve the efficiency and possibly the generality of proof-based synthesis techniques or a programmer's apprentice.

2. We have demonstrated the first *resource control* mechanism for a problem solving system that is truly parallel. Previous mechanisms (e.g. tree search strategies, agendas) decide resource management on an event level; the mechanism decides which is the best single thing to do next. We have tested several resource control strategies using parallel resource control and demonstrated that there is no easy mapping for them onto sequential strategies.

3. The example systems have made extensive use of activities, viewpoints, sprites, and assertions. There seem to be no conceptual difficulties with using these constructs in large programs.

4. We have developed an assertion-oriented language where the user has the opportunity to define the behavior of the assertions and sprites to take advantage of the semantics. We have seen that this leads to both greater efficiency and flexibility of the language. The advantage of this style of implementing over the more conventional lexical retrieval schemes (e.g. discrimination nets) seems clear. Furthermore our implementation points to a unification of the concepts of assertion-oriented and object-oriented programming. The examples we have used *compile* assertion-oriented code into object-oriented code. The resulting synthesis is very elegant. We have shown that the concept of "constraints" can be generalized within this framework.

5. We have developed a Lisp implementation of this language that is quite practical. The advantages of parallelism as a conceptual tool do not require us to wait for new hardware architectures before they can be used. Our code mixes "standard Lisp" with "Ether." The mix is surprisingly smooth and has caused

us little difficulty. With minor caveats we can treat our Ether code as being parallel with binding happening correctly. We hope that the language ideas we have developed will be of some guidance to designers of parallel hardware.

Chapter X Bibliography

[1] Baker, Henry G., *Actor Systems for Real-Time Computation*, MIT Laboratory for Computer Science TR-197, March 1978.

[2] Barber, Gerald, *Office Semantics*, MIT PhD thesis, 1982.

[3] Bledsoe, W., *Non-Resolution Theorem Proving*, Artificial Intelligence, 9(1), 1977.

[4] Bobrow, Daniel G., Bertram Raphael, *New Programming Languages for Artificial Intelligence Research*, Computing Surveys, Vol. 6, No. 3, September 1974.

[5] Borning, Alan, *Thinglab -- A Constraint-Oriented Simulation Laboratory*, XEROX PARC report SSL-79-3, July 1979.

[6] Bundy, Alan, *Similarity Classes*, University of Edinburgh, Department of Artificial Intelligence Working Paper 25, January 1978.

[7] Clinger, William, *Foundations of Actor Semantics*, MIT Artificial Intelligence Laboratory TR-633, May 1981.

[8] Dahl, Ole-Johan, Kristen Nygaard, *Simula -- An ALGOL-Based Simulation Language*, Communications of the ACM, September 1966.

[9] Davis, Randall, *Applications of Meta Level Knowledge to the Construction Maintenance and Use of Large Knowledge Bases*, Stanford Artificial Intelligence memo AIM-283, July 1976.

[10] Davis, Randall, *Meta-Rules: Reasoning About Control*, MIT Artificial Intelligence Laboratory memo 576, March 1980.

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

[12] Erman, L. D., F. Hayes-Roth, V. R. Lesser, D. R. Reddy, *The Hearsay-II Speech-Understanding system: Integrating Knowledge to Resolve Uncertainty*, Computing Surveys, June 1980.

[13] Fikes, Richard E., Nils J. Nilsson, *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*, Artificial Intelligence, Vol. 2, no. 3, 1971.

[14] Gaschnig, John, *Performance Measurement and Analysis of Certain Search Algorithms*, Carnegie-Mellon report CMU-CS-79-124, May 1979.

[15] Goldstein, Ira P., *Understanding Simple Picture Programs*, MIT Artificial Intelligence Laboratory TR-294, September 1974.

[16] Green, Cordell, *Theorem-proving by Resolution as a Basis for Question-answering Systems*, Machine Intelligence 4, 1969.

[17] Halstead, Robert H., *Multiple-Processor Implementations of Message-Passing Systems*, MIT Laboratory for Computer Science TR-198, January 1978.

[18] Hansson, Ake, Sten-Ake Tarnlund, *A Natural Programming Calculus*, Fifth International Joint Conference on Artificial Intelligence, 1977.

[19] Hardy, S., *Synthesis of Lisp Functions From Examples*, Fourth International Joint Conference on Artificial Intelligence, 1975.

[20] Hewitt, Carl, *PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot*, First International Joint Conference on Artificial Intelligence, 1969.

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

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

[23] Hewitt, Carl, Giuseppe Attardi, Henry Lieberman, *Specifying and Proving Properties of Guardians for Distributed System*, in *Semantics for Concurrent Computations* G. Kahn, ed., *Lecture Notes in Computer Science* no. 70, Springer-Verlag, 1976.

[24] Hewitt, Carl, Giuseppe Attardi, Henry Lieberman, *Security and Modularity in Message Passing*, First International Conference on Distributed Computing Systems, Hunstville, Alabama, October 1979.

[25] Hewitt, Carl, Brian Smith, *Towards a Programming Apprentice*, IEEE Transactions on Software Engineering, March 1975.

[26] Hewitt, Carl, *Viewing Control Structures as Patterns of Passing Messages*, Artificial Intelligence Journal, vol. 8, no. 1, June 1980.

[27] Hewitt, Carl, *Design of the APIARY for Actor Systems*, Proceedings of the 1980 Lisp Conference, Stanford, CA, August 1980.

[28] Hume, David, *An Inquiry Concerning Human Understanding*, 1748.

[29] Ingalls, Daniel, *The Smalltalk-76 Programming System: Design and Implementation*, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson Arizona, January 1978.

[30] Kornfeld, William, *Using Parallel Processing for Problem Solving*, MIT Artificial Intelligence Laboratory memo 561, December 1979.

[31] Kornfeld, William, *ETHER -- A Parallel Problem Solving System*, Sixth International Joint Conference on Artificial Intelligence, August 1979.

[32] Kornfeld, William, Carl Hewitt, *The Scientific Community Metaphor*, IEEE Systems Man & Cybernetics, vol. SMC-11 no. 1, January 1980.

[33] Kornfeld, William, *A Synthesis of Language Ideas for AI Control Structures*, MIT Artificial Intelligence Laboratory Working Paper 201, August 1980.

[34] Kornfeld, William A., *The Use of Parallelism to Implement a Heuristic Search*, Seventh International Joint Conference on Artificial Intelligence, August 1981.

[35] Kowalski, Robert, *Logic for Problem Solving*, North-Holland Press, 1979.

[36] Lakatos, Imre, *Falsification and the Methodology of Scientific Research Programmes*, in (Musgrave and Lakatos eds.) *Criticism and the Growth of Knowledge*, Cambridge University Press, 1970.

[37] Lakatos, Imre, *Proofs and Refutations*, Cambridge University Press, 1976.

[38] Lamport, Leslie, *Time, Clocks and the Ordering of Events in a Distributed System*, Massachusetts Computer Associates, Inc. memo, March 1976.

[39] Landin, Peter J., *A Correspondence between Algol 60 and Church's Lambda-Notation*, Communications of the ACM, February 1965.

[40] Lenat, D., *An AI Approach to Discovery in Mathematics as Heuristic Search*, Stanford AI Lab Memo AIM-286, 1976.

[41] Manna, Zohar, Richard Waldinger, *Synthesis: Dreams -> Programs*, SRI International Technical Note 156, November 1977.

[42] Manna, Zohar, Richard Waldinger, *A Deductive Approach to Program Synthesis*, ACM Transactions on Programming Languages and Systems, Vol. 2, no. 1, January 1980.

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

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

[45] Minsky, Marvin, *A Framework for Representing Knowledge*, MIT Artificial Intelligence Laboratory memo 306, June 1974.

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

[47] Newell, Allen, Herbert Simon, *GPS, A Program that Simpulates Human Thought*, in *Computers and Thought* Feigenbaum and Feldman, eds., 1963.

[48] Nilsson, Nils J., *Principles of Artificial Intelligence*, Tioga Publishing Company, 1980.

[49] Popper, Karl R., *Conjectures and Refutations*, Basic Books, 1962.

[50] Popper, Karl R., *The Logic of Scientific Discovery*, Harper and Row, 1968.

[51] Quine, W. V. O., *Mathematical Logic*, Harvard University Press, 1965.

[52] Rich, Charles, *Inspection Methods in Programming*, Massachusetts Institute of Technology Artificial Intelligence Laboratory TR-604, December 1980.

[53] Robinson, J. A., *A Machine-oriented Logic Based on the Resolution Principle*, Journal of the Association for Computing Machinery vol. 12 no. 1, 1965.

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

[55] Schank, R., R. Abelson, *Scripts, Plans, Goals and Understanding*, Lawrence Erlbaum Associates, 1977.

[56] Selfridge, O. G., *Pandemonium: A Paradigm for Learning*, in Blake, Uttley, eds. *Proceedings of the Symposium on Mechanisation of Thought Processes*, 1959.

[57] Shaw, D., W. Swartout, C. Green, *Inferring LISP Programs from Examples*, Fourth International Joint Conference on Artificial Intelligence, August 1975.

[58] Shrobe, Howard E., *Dependency Directed Reasoning For Complex Program Understanding*, MIT Artificial Intelligence Laboratory TR-503, April 1979.

[59] Steele, Guy L., Gerald Sussman, *Constraints*, MIT Artificial Intelligence Laboratory memo 502, November 1978.

[60] Steele, Guy L., Gerald Sussman, *The Revised Report on Sceme, a dialect of Lisp*, MIT Artificial Intelligence Laboratory memo 452, January 1978.

[61] Steele, Guy L., *RABBIT: A Compiler for Scheme (A Study in Compiler Optimization*, MIT Artificial Intelligence Laboratory TR-474, May 1978.

[62] Steele, Guy L., *The Definition and Implementation of a Computer Programming Language Based on Constraints*, MIT Artificial Intelligence Laboratory TR-595, August 1980.

[63] Summers, Phillip D., *A Methodology for LISP Program Construction from Examples*, Journal of the ACM, vol. 24, no. 1, January 1977.

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

[65] Sussman, Gerald, *A Computational Model of Skill Acquisition*, MIT Artificial Intelligence Laboratory TR-297, August 1973.

[66] Waltz, David, *Generating Semantic Descriptions from Drawings of Scenes with Shadows*, PhD thesis Massachusetts Institute Of Technology, 1972.

[67] Waters, Richard, *Automatic Analysis of the Logical Structure of Programs*, MIT Artificial Intelligence Laboratory TR-492, December 1978.

[68] Weinreb, Daniel, David Moon, *Lisp Machine Manual (third edition)*, MIT Artificial Intelligence Laboratory Report, March 1981.

[69] Whitehead, Alfred N., Bertrand Russell, *Principia Mathematica*, Cambridge University Press, 1927.

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

[71] de Kleer, J., J. Doyle, C. Rich, G. Steele, G. Sussman, *AMORD A Deductive Procedure System*, MIT Artificial Intelligence Laboratory Memo 435, January 1978.

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