THE DESIGN OF A PARALLEL PROGRAMMING LANGUAGE FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

by

Masahiro Honda

Computer Sciences Technical Report #331
MACC Technical Report #52

August 1978

THE DESIGN OF A PARALLEL PROGRAMMING LANGUAGE FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

BY

MASAHIRO HONDA

A thesis submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1978

THE DESIGN OF A PARALLEL PROGRAMMING LANGUAGE
FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

Masahiro Honda

under the supervision of Professor Larry E. Travis

ABSTRACT

We examine the problems of designing a parallel programming language for artificial intelligence (AI) applications. The problems arise from the conflicting sets of requirements imposed by AI applications and parallel processing. AI applications require flexible access to data as well as flexible control over program flow. Such flexibility is difficult to provide in a parallel processing environment, where data access and program flow must be strictly controlled. We start with the design of a parallel base language, TOMO, to which AI features (such as associative retrieval and data contexts) are later added. TOMO draws heavily, both in concept and in philosophy, from recent research done on system implementation languages, notably Concurrent-PASCAL. Strong emphasis is placed on language enforcement of controls on concurrency. The original monitor concept developed by Hoare and Brinch Hansen is generalized to allow more flexible, yet secure, sharing of data. TOMO programs consist of "modules", which encapsulate procedures with data and serve as restricted referencing environments for processes. A process can only access data of the module it is executing in. A process may transfer from one module to another through remote procedure calls. Such transfers, however, are regulated by module guards, which can suspend processes on queues and are responsible for maintaining cooperation among processes.

TOMO modules are used to provide AI features found in TELOS, a recent AI language based on PASCAL. We show how the module guard can act as a control abstraction mechanism providing complex inter-process control structures, including demons and pattern-directed invocation of processes. Guards can also be used to implement complex forms of message passing, including addressing messages by pattern. The data context mechanism and associative data base facility are provided as predefined modules. In order to avoid problems caused by concurrency, the asociative data base facility consists of many independent and local (to a module) data bases rather than a global, monolithic one.

Although the primary motivation behind the design effort is AI applications, the innovations made in TOMO should be applicable to other applications where multiprocessing is

involved, such as operating systems and data base management systems.

Acknowledgements

Numerous individuals have contributed to this dissertation. Professor Larry Travis has consistently been a willing listener and critic of my ideas. He has provided the guidance without which this work would not have been possible. He has also provided me with the opportunity to take part in the TELOS project, from which this dissertation grew. I would also like to thank him for the financial support he has provided the last two years. I would like to thank Professor Charles Fischer for the invaluable experience working on the UW PASCAL compiler. His careful reading of the dissertation is also gratefully acknowledged. He has consistently been a source of good ideas and good criticisms for bad ideas. Professor Raphael Finkel has gone beyond the call of duty to significantly improve the readability of this dissertation. To him (and to his red pen) go a great deal of thanks. He has also prodded me with the proper questions, which have clarified many ideas. I would like to thank Professor Leonard Uhr for providing inspirations in the artificial intelligence area. His unique and thought provoking ideas have had a significant impact on my thinking in that area.

My TELOS colleagues, Steve Zeigler, Rich LeBlanc, and Frank Horn, have provided a stimulating environment in which to work. The discussions I have had with them on issues of computer science have provided invaluable insights on difficult problems. I would especially like to thank Frank Horn for "pointing the way" when difficulties arose.

On a more personal level, I wish to thank my parents for the many years of support they have given me. They have sacrificed much to get me to this point.

Finally, to my wife Yeung Ha go my deepest appreciation for her companionship during my years in Madison. She has given me many wonderful things during that time, without which life as a computer science graduate student would not have been as pleasant. Her encouragement and understanding during the last few months is especially appreciated. It is to her that I dedicate this dissertation. I would also like to acknowledge her for some invaluable insights on the multiprocessing problems of married life.

Table of Contents

Chapter 1
Introduction and Motivation

The problems of designing a parallel programming language for artificial intelligence (AI) applications are examined in this dissertation. We are motivated by recent hardware advances that promise small, inexpensive, but powerful processors [Noyce 77] and by the recent general interest in parallelism that has resulted from the hardware advances. Several computer science research organizations in the United States are currently investigating the design of parallel machines consisting of a large number of microprocessors [Finkel and Solomon 77, Swan et al. 77, Despain and Patterson 77]. Research is also in progress to develop operating systems [Solomon and Finkel 78, Jones et al. 77] and programming languages [Hoare 77, Brinch Hansen 77, Feldman 76] for multiprocessor machines.

Researchers in the area of artificial intelligence have also begun to take a strong (and renewed) interest in parallelism as a result of the hardware advances. It has long been recognized that the high demands for computing power in AI can potentially be met, at least in part, through concurrency. In applications such as the speech recognition project at Carnegie-Mellon [Fennel and Lesser 75], an execution speed increase of a factor of 10 to

100 can be crucial for real-time response. Parallelism for AI has more recently been motivated by the advent of a "dialogue style" of programming based on a collection of autonomous processes communicating via messages [Hewitt 73, Lenat 75, Atkinson and Hewitt 77, Hewitt 77]. Such a collection is analogous to a society of experts cooperating with each other via explicit message communication to attack a problem. Knowledge is distributed among several "expert" processes rather than centralized in one main process. Recent interest in production systems [Newell 73] has further motivated this style of programming. Programs maintaining this dialogue style can very naturally be given the form of production systems. In order to take full advantage of multiprocessor machines and the recent ideas about multiprocessing for AI, the AI researcher must be provided with a programming language having features for concurrency (sharing, synchronization, communication) as well as features for AI applications (patterns, demons, contexts).

The area of parallel (or concurrent) AI languages has therefore become the object of some interest. Among such languages currently undergoing development are PLASMA [Hewitt 77] and KRL [Bobrow and Winograd 76] (although the major emphasis in KRL is not parallelism). Earlier efforts to provide multiprocessing capabilities in AI langauges exist, such as in SAIL [VanLehn 73], but their constructs for

multiprocessing do not have the benefit of recent research in the areas of parallel processing and language design.

The design goal of a language that provides features for both concurrency and AI applications imposes a unique set of requirements on the designer. AI applications require flexible access to data and control of program flow. Data and routines must be identifiable associatively by patterns as well as explicitly by variables. Some AI applications also require routines not only to be identified by patterns but also to be invoked when particular patterns match existing data. Mechanisms for managing tentative and alternative modifications to data are required to support backtracking and alternative problem solving strategies. The requirements imposed by AI applications, however, must be jointly satisfied with requirements imposed by concurrency. To support concurrency, mechanisms to create, invoke, and synchronize processes are needed. The language must provide facilities for concurrent processes to securely share data. In addition, processes must be able to communicate with each other to achieve cooperation. To satisfy these concurrency requirements, the language must impose strict controls on data access and process execution. Such controls, however, become difficult to implement when combined with the flexible mechanisms for data access and control of flow required by AI applications. In particular,

the combination of AI and concurrency features can lead to a large number of "cross product" interactions that impose language constraints that are both unreasonable and difficult to understand. We must therefore meet the above requirements with a minimal number of language constructs.

We attack the problems caused by the above requirements by drawing on results from two areas of programming language research: serial AI languages and system implementation languages. The area of serial programming languages for AI applications has been investigated extensively [LeBlanc 77, Wilber 76, Bobrow and Raphael 74, LeFaivre 74, VanLehn 73, Davies 73, McDermott and Sussman 72, Hewitt 72]. One of the most recent efforts in this area is TELOS [Travis et al. 77, LeBlanc 77], a PASCAL-based language that provides a small set of abstraction mechanisms for AI features rather than a large number of high-level, specialized constructs. TELOS also benefits from the extensive data-structuring facilities of PASCAL [Jensen and Wirth 76] as well as products of recent programming language research, such as abstract data types [Liskov et al. 77, Shaw et al. 77]. In addition, the TELOS pattern matching facilities allow flexible and powerful specification of match criteria through special procedures called match routines. TELOS is therefore an ideal source of AI features for a parallel AI language. Research in system implementation languages, on the other hand, has

tried to increase reliability of operating systems through language enforcement of controls on concurrency. Strong emphasis has been placed on constructs that permit compile-time checks for consistency as well as proofs of correctness. Recent system implementation languages, such as Concurrent PASCAL [Brinch Hansen 75] and MODULA [Wirth 77], are therefore an ideal source of concurrency features for a parallel AI language.

Two major problem areas arise from the above. First, constructs for concurrency developed for recent languages need to be modified to meet AI processing requirements. Second, AI features found in TELOS must be adapted for a parallel processing environment. We start in Chapter 2 by reviewing use of parallelism in recent AI programs and describing requirements for parallel AI processing based on such programs. Chapter 2 then discusses existing constructs for concurrency and relates them to the AI requirements. We present in Chapter 3 major innovations of TOMO, a parallel base language with constructs for concurrency to meet the parallel AI processing requirements. Chapter 4 presents the detailed specifications of TOMO. The problem of providing AI features in a parallel environment is discussed in Chapter 5, where we show how TOMO can be extended to accommodate the AI features of TELOS. Chapter 6 summarizes the major

results presented in the dissertation and discusses problems for future research.

Chapter 2

Requirements for Parallel AI Processing

2.1 Introduction

In this chapter we define requirements that should be met by constructs for concurrency in a parallel AI language. We arrive at the requirements by reviewing work in AI that has used parallelism. Existing constructs for concurrency are then analysed with respect to AI applications. We then outline the approach taken in the design of the parallel AI language described in the following chapters.

2.2 Use of Parallelism in AI

Several researchers have proposed and built extensive AI programs that have some flavor of parallelism in them. Perhaps the most direct use of parallelism is Fennel and Lesser's contribution to the speech recognition system at Carnegie-Mellon (Hearsay-II) [Fennel and Lesser 75]. Their scheme employs a parallel production system where the pre-conditions and actions of several productions may execute concurrently. Productions read and update a global "blackboard" that represents a state of the problem solution. All productions whose pre-conditions are satisfied by the blackboard are fired concurrently. The action part of

each production is called a "knowledge source". Execution of knowledge sources and pre-conditions are not indivisible and do not exclude each other in time. Integrity of data is maintained by locking blackboard regions delineated according to various criteria and by notifying a knowledge source when data on which it is basing its work becomes invalid when another knowledge source updates the blackboard. Knowledge sources do not know of each other; they communicate entirely through the blackboard as a normal part of the production system operation rather than through messages. The scheme was designed for use with an hypothesize-and-test paradigm where the knowledge sources both suggest and verify hypotheses on the blackboard.

In contrast, Lenat [Lenat 75] has built a program comprised of what he calls "beings", which resemble productions whose pre-conditions can be evaluated concurrently but only one chosen action may execute at one time. Each action, however, may have concurrent parts. Beings are triggered by questions that are asked by other beings and broadcast to all beings; those that find the question relevant to their abilities and responsibilities will attempt to respond. The scheme models a group of experts sitting around a conference table trying to solve a problem. Beings communicate with each other through a highly structured message mechanism. All beings are capable of asking and answering only a stan-

dard set of question types, so there is no need for each being to know about other beings in order to communicate.

Use of concurrency in AI has also been proposed in the area of natural language processing. Kaplan [Kaplan 73] has suggested a multiprocessing approach to parsing natural language sentences. The scheme represents the states of alternative parses in a global "chart". Processes are associated with each phrase, and they are activated by higher-level components of the grammar in a way similar to how sub-networks are called in Woods' augmented transition network [Woods 70]. However, since processes are used instead of sub-networks (which are essentially subroutines), parses can be generated and the results saved concurrently. Kaplan claims that this method has the advantages of both top-down and bottom-up methods.

The potential for multiprocessor machines has also generated interest in applying concurrency to pattern recognition. Pattern recognition programs typically require repeated application of the same algorithms over different data. Already machines with large arrays of simple processors have been built for image processing [Duff 76]. Although the architecture of these machines so far is primitive and the power of each processor limited, further research in this area should produce valuable insight for the design of

more powerful multiprocessor machines, making the case for concurrency even stronger.

Other uses of concurrency proposed for AI include pipelining [Uhr 75], concurrent search [Fahlman 75], divide-and-conquer strategies (decomposition of a problem into independent subproblems), concurrent evaluation of terms of a polynomial evaluation function, and concurrent analysis of features in scene analysis [Uhr 75].

2.3 The Requirements

From the previous section we can see several requirements for a parallel AI language. First, we must be able to specify flexible sharing of large, complex data structures among parallel processes. The blackboard of Hearsay-II is an example of such a structure. Different levels of locking are required to obtain different "grains" of sharing. Certain applications may require locking an entire structure, while other applications may call for locking only subparts of a structure. In order to prevent unnecessary serialization of processes, concurrent read access, as well as concurrent access to independent subparts of a data structure, should be possible. Certain applications may even call for no locking of a shared data structure at all. (In one experiment with Hearsay-II locking was inhibited to

increase parallelism, but since the knowledge sources were designed to handle imperfect data, little degradation of performance was seen [Lesser and Erman 77].) Language constructs that allow the programmer to securely and simply implement a wide variety of disciplines for sharing are therefore required.

Another requirement is the ability to specify coordination of a set of parallel processes. Control of sharing is actually a special case of process coordination. Methods for synchronizing processes to implement pipelines and concurrent search strategies are required. Invocation of processes by pattern (that is, according to conditions existing over data values) is required to implement parallel production systems and other data-directed parallel computation schemes. In general, we require schedulers and resource allocators that can distribute control among a set of processes according to user-specified strategies. The KRL effort [Bobrow and Winograd 76] plans to provide such mechanisms.

Communication among processes is also a major requirement [Lenat 75, Hewitt 73]. Constructs for message passing as well as message broadcasting are required (such as in Lenat's being program). A natural extension of pattern-directed process invocation is pattern-directed mes-

sage passing. AI programs often consist of autonomous modules that do not know of other modules by name, but know of their descriptions. Addressing of messages in a collection of such modules must necessarily be by descriptions (patterns).

Finally, we require that "traditional" AI language features, such as data contexts and an associative data base, be compatible with constructs for sharing, coordination, and message communication. We require that data objects accessed associatively be sharable just as securely and flexibly as normal data objects. In addition, operations on the data context mechanism must be defined so that several processes may concurrently share execution in the same context as well as in different ones.

2.4 Existing Constructs for Concurrency

Having seen the requirements for parallel processing in the AI domain, we now examine existing constructs for concurrency found in recent languages and relate them to AI applications. Various constructs have been proposed to generate concurrency (invoke parallel processes). Perhaps the most primitive construct is Fork, which spawns a new process that executes in parallel with Fork's invoker. Brinch Hansen has pointed out numerous shortcomings of such an

"unstructured" specification of concurrency [Brinch Hansen 73]. The major problem with Fork (and its counterpart Join, which delays a process until another process terminates) is that a reader of a program cannot readily determine which statements execute concurrently with which others. As an alternative, Dijkstra [Dijkstra 68] has suggested the CoBegin-CoEnd construct in which statements in the CoBegin block execute in parallel with each other. Regions of parallel execution are now clearly delineated. In order to avoid uncontrolled race conditions, data modified by one statement of the CoBegin block are not allowed to be read or modified by other statements in the same block. Restrictions on parameter passing and use of global variables are required to enforce this constraint. A major shortcoming of the CoBegin block is that the number and identity of parallel processes generated is fixed at compile time. In AI applications processes are often invoked associatively; neither their number nor their identity is known at compile time. A more flexible means of generating parallelism is therefore required. Another method recently proposed for generating parallelism is the "eager" evaluation of a function's arguments [Friedman and Wise 78] (also known as "passing by futures" [Baker and Hewitt 77]). With this method a function's arguments are evaluated concurrently with the function's execution (and with the other arguments). A new process is created for each argument. This

method, however, requires functions to be free of side-effects and is therefore more suited for applicative languages such as "pure" LISP.

Once processes have been invoked in parallel, methods of synchronization are required to control access to shared data, and for general process coordination. Dijkstra's <u>semaphore</u> [Dijkstra 68] is one such method. Again, Brinch Hansen considers semaphores too unstructured since their use is prone to errors [Brinch Hansen 73]. Hoare [Hoare 73] has suggested as an alternative <u>conditional critical regions</u> whose syntax is as follows:

region V when <Boolean expression> do <statement>

V is a shared variable accessible only from <statement> of critical regions for V ("region V ..."). Execution of <statement> is delayed until <Boolean expression> becomes true. Unfortunately, conditional critical regions are distributed throughout the program to places where V is accessed. To determine whether V is safely shared requires having to look at all <Boolean expression>s of critical regions involving V. Conditional critical regions also cannot explicitly schedule processes competing to access V. Complex process coordination required by AI applications would therefore be difficult to specify.

In order to make sharing of data more flexible and secure, Hoare and Brinch Hansen have jointly proposed a form of abstract data types called monitors [Hoare 74, Brinch Hansen 75]. An abstract data type specifies an encapsulation of routines and data, where the data is only accessible through the routines. The implementation of a data type (e.g., a stack) is hidden from the users of the data type (the processes that call its routines, which are the data type's operations). A monitor restricts the number of processes executing its routines to at most one so that its data may be safely shared. Within each routine processes accessing the shared data can be explicitly scheduled through operations on special "queue" variables (also called "condition" variables). Control of sharing is centralized and hidden in the monitor's routines rather than being distributed throughout the program. Correctness of the sharing is therefore easier to verify. Furthermore, the users of a monitor are not burdened with maintaining the integrity of the shared data.

Unfortunately, monitors may impose too much mutual exclusion. Concurrent read access or concurrent access to independent subparts of a data structure cannot be specified safely. For example, suppose we wish to allow concurrent read access to a variable V. If V is declared as a variable of a monitor, it must always be accessed from a routine of

the monitor. Concurrent access is therefore not possible, although V is protected from a write access occurring while it is being read. In order to allow concurrent read access to V, it must be declared outside the monitor. Access to V is still protected by the monitor if each access follows these steps:

1) Call monitor routine to get permission to access V.
2) Access V.
3) Call monitor routine to notify access has been completed.

Nothing constrains the programmer, however, to adhere to these steps. Users of the monitor are burdened with maintaining the integrity of the shared data. Monitors, therefore, are not sufficient for securely providing flexible forms of sharing required by AI applications.

Another limitation of monitors has been pointed out by Silberschatz et al. [Silberschatz et al. 77]. They show that monitors are not adequate for managing a pool of resources. For example, suppose elements of an array of resources were to be allocated to processes upon request. Such an array would have to be declared within a monitor to protect it from arbitrary access. Access to its elements

would therefore have to be through a routine of the monitor. We are again faced with the situation where the monitor imposes unnecessary serialization of access, since two elements of the array cannot be simultaneously accessed. Furthermore, if the array elements are themselves monitors, we can run into a deadlock. Suppose monitor M1 is an element of an array A defined in monitor M2. If a process executing a routine of M1 is suspended on a queue, mutual exclusion on M1 is released but not on M2 (according to the rules of Concurrent PASCAL [Brinch Hansen 75, Silberschatz et al. 77]). The suspended process can only be awakened by another process executing in M1. In order to execute a routine of M1, a process must be able to execute in M2, which is not possible since mutual exclusion has not been released on M2. Silberschatz et al. propose the use of capabilities with monitors to solve these problems. A capability consists of a pointer to an object and a list of access rights that a process may exercise via the capability on that object. A capability would point to and protect each array element independently from the monitor. The monitor's function is changed from a protector of array elements to a manager of capabilities, having powers to grant capabilities to requesting processes.

As an improvement over monitors, Atkinson and Hewitt have proposed serializers [Atkinson and Hewitt 77], which

relay messages between a shared resource and its users. A user wishing to access a shared resource sends a message to the resource via the resource's serializer. The serializer holds the message until the resource is ready to accept it. To determine whether the resource is ready to accept a message, the serializer may inspect memberships of crowds, which keep track of processes whose messages are currently being processed by the resource. For example, a resource might accept two kinds of messages, Read and Write. The serializer for the resource would define two crowds, one for Read and one for Write. While Write is being processed, the crowd for Write will be non-empty. A Read or another Write message will not be relayed to the resource until the Write crowd becomes empty. Although execution within the serializer must be serial, a resource may process several messages in parallel (such as two Reads). If all messages to a resource are relayed by a serializer, then the resource can be shared securely. Atkinson and Hewitt, however, do not provide methods for constraining messages to be relayed by serializers.

Since a serializer controls sharing on a message-by-message basis, it cannot handle applications requiring an indivisible sequence of messages to be sent to a resource. For example, suppose we have two integer resources R1 and R2 that accept messages Read and Write. We want

to compare R1 and R2 and swap them if R1 is greater. We therefore require a Read possibly followed by a Write on each resource, but we cannot prevent another Write on the resources between the comparison and the swap. R1 and R2 would have to be able to accept additional sorts of messages, such as ReadAndLock, in order to avoid this difficulty. Serializers, therefore, also lack some flexibility that AI applications may require.

Owicki has recently proposed a variation on monitors called shared classes to relax the overly strict mutual exclusion rules of monitors [Owicki 77]. Her scheme allows concurrent execution of routines of shared classes (which are otherwise monitors), and her proof rules for parallel programs [Owicki and Gries 76] are used to show correctness of sharing. She relies on certain restrictions governing where variables for synchronization can be modified, which are not imposed on other variables of the shared class. Such restrictions not only simplify proofs but also enhance readability and writability of parallel programs.

Habermann [Habermann 75] has suggested an alternative method to monitors for controlling access to shared resources. His method requires for each shared abstract data type a specification of a regular expression over the routine names of the abstract data type. The regular expression is

called a path expression, and it defines the permissible sequence of routine calls (operations) that may be performed on the abstract data type. Operations are delayed if necessary to adhere to the sequence. A path expression is therefore a high-level description of the synchronization properties of a shared resource. It is possible to compile a path expression into code necessary to implement the synchronization specified. A compiler can also detect presence of certain deadlocks from the path expressions. Path expressions, however, only specify synchronization of mutually exclusive operations. Specification of concurrent read operations, for example, cannot be specified. Path expressions are, nevertheless, an elegant extension of abstract data types. McGraw and Andrews [McGraw and Andrews 77] have proposed a construct similar to path expressions that specifies permissible combinations of concurrently executable routines rather than permissible sequences.

Although one use of sharing is for communication among processes, messages provide an alternative communication mechanism (as well as a synchronization mechanism). One recent language that uses messages is PLITS [Feldman 76], which provides the primitivs Send and Receive. A process may Send and Receive messages for a transaction. A Receive will delay a process if no messages are present for the transaction. Messages consist of tuples of name-value pairs

called slots, and each process can be restricted to access only a subset of the slots to provide protection. Feldman shows that the PLITS message mechanism can be used to construct various control structures, such as procedure calls and coroutine structures. Hoare [Hoare 77] and Brinch Hansen [Brinch Hansen 77] have also proposed languages in which messages play a key role. They liken message passing to I/O operations and show that messages together with Dijkstra's guarded commands [Dijkstra 75] are quite versatile in providing procedures, coroutines, abstract data types, monitors, processes, and semaphores. Finally, Hewitt has developed a theory of computation based on message passing among what he calls actors [Hewitt 73,77], and has developed a programming language, PLASMA, to embody the theory.

We believe that all these notions on message passing are worthwhile, but also believe that messages by themselves are not sufficiently convenient. Although it has been demonstrated that higher level control structures can be built from messages, some structures, such as procedures, are used often enough to warrant their inclusion in a language. On the other hand, we do not yet know what sorts of control structures over processes are most common, and we therefore may rely on the programmer's use of messages to implement his own interprocess control structures. We also believe that the complexity of message communication in AI applica-

tions (such as addressing by pattern) requires in some cases user specification of message mechanisms to supplement built-in ones.

2.5 Towards a Parallel AI Language

We have seen that many worthwhile, though not perfect, ideas for parallel language constructs exist. We adapt some of these ideas in the design of the parallel AI language described in the following chapters. In particular, we make use of crowds as developed by Atkinson and Hewitt as well as capabilities as suggested by Silberschatz et al. to provide a flexible facility for sharing. This sharing facility is then used as a primitive for pre-defined and user-defined message mechanisms for implementing interprocess communication and interprocess control structures.

On a more general level, we subscribe to the philosophy advocated by Wirth, Hoare, Dijkstra, Brinch Hansen, and others that a language should have sufficient structure so that checks can be made for errors, preferably at compile-time. Security is a goal of any language design, and compile-time checking can often ensure correct use of the features. Not only will such checking increase efficiency due to reduction of run-time checks, but it will also considerably reduce the burden of debugging at run time. Run-time debugging is par-

ticularly difficult for parallel programs, which are inherently non-deterministic. When compile-time checks are not possible, efficient methods for performing compiler-generated run-time checks should be sought. In addition, a language should provide data and control abstraction mechanisms (e.g., abstract data types) that place the burden of maintaining security on the implementor of an abstraction rather than on its users (recall the discussion on monitors). Such mechanisms permit programmer specification of run-time checks that are beyond a compiler's capabilities to generate.

Abstraction mechanisms also allow us to avoid supplying constructs that are too high-level. A major shortcoming of early AI languages has been that their constructs impose a particular method of programming. An early example of this shortcoming is backtracking in PLANNER, which accommodates only depth-first search. Constructs for meeting the requirements for parallel AI processing should be sufficiently low-level so that they do not favor a particular strategy for using parallelism. The constructs, of course, should not be so low-level that details for managing concurrency disrupt the programmer's effort to solve his problem. Formulating constructs at the appropriate level is a general problem in the design of programming languages and is particularly acute in AI languages due to the diversity and

complexity of the applications. One method of attacking the "level" problem is to design an abstraction mechanism that enables the programmer to build his own high-level constructs to control concurrency. As we will see in later chapters, we borrow heavily from recent research done on abstract data types to develop such an abstraction mechanism.

In order to take advantage of the recent research done on programming languages, we start with a strongly-typed procedural langauage, in our case PASCAL [Jensen and Wirth 76], as a base. Use of PASCAL enables us to also take advantage of the work done on the PASCAL-based AI language TELOS [Travis et al. 77, LeBlanc 77]. TELOS takes full advantage of recent research in general purpose as well as AI programming languages and is therefore an ideal source of AI features for a PASCAL-based parallel AI language.

2.6 Conclusion

This chapter has defined the problems that are approached in the following chapters. Past AI programs that used parallelism and existing constructs for parallelism were analyzed to determine the direction to take in achieving the goals set out in Chapter 1. The next chapter takes the first step in that direction by developing a base lan-

guage for parallelism that can accommodate features for AI applications.

Chapter 3

A Base Language: TOMO

3.1 Introduction

This chapter presents the design of TOMO, a base language for parallelism that will accommodate features for AI applications. An informal semantic model of TOMO is first presented, followed by an overview of the language. Several examples of TOMO program segments are given to illustrate TOMO's major innovations. We discuss in particular how TOMO provides mechanisms for coordinating parallel processes, for sharing data objects among such processes, and for communicating among such processes. The actual specifications of TOMO are given in the next chapter, and the problem of merging features for AI applications is examined in Chapter 5.

3.2 An Informal Model

An informal semantic model of a language abstracts the essence of the language while supressing its syntax and low-level semantics. Such a model for TOMO is developed to convey an intuitive understanding of TOMO and to provide a vocabulary that will be used throughout the rest of this dissertation.

A procedural language for sequential computation is modelled by a single stream of control passing from one statement to another. Each statement manipulates data accessed through variables in its referencing environment. TOMO is a parallel language, and therefore there are several conceptually concurrent streams of control, called processes. In order to minimize mutual interference among processes, TOMO constrains the activity of each process at all times within restricted referencing environments that localize its effects. A TOMO process always executes within a referencing environment provided by a module. Each module contains data that only processes executing inside it can manipulate. Such data are called the private data of the module. Processes execute procedures of a module and they may access a module's private data through the private variables of the module. The encapsulation of procedures and data in a module has been used in the past to implement abstract data types such as SIMULA classes [Dahl et al. 70], CLU clusters [Liskov et al. 77], ALPHARD forms [Shaw et al. 77], MODULA [Wirth 77] and EUCLID [Lampson et al. 77] modules, and TELOS capsules [LeBlanc 77]. Modules are used in TOMO to represent restricted environments for execution of processes.

Processes in TOMO may transfer from one module to another through remote procedure calls. Processes always re-

turn to the module they left. Parallelism is generated when a process performs a parallel remote procedure call to invoke procedures of several modules simultaneously. Such a call creates a new process in each of several modules to execute the specified procedures in parallel. As each procedure returns, the process created by the call is terminated. When all parallel calls have returned, execution of the calling process continues. A remote procedure call may have a restricted class of parameters, which will be discussed later.

A parallel remote procedure call counts as a computational step of the algorithm implemented by the procedure containing the call. Each procedure, therefore, implements a serial algorithm whose steps may include those implemented in parallel. The algorithm itself is conceptually independent of the implementation of each of its steps. Such an approach to structuring parallel programs is consistent with the structured concurrent programming concepts advocated by Brinch Hansen [Brinch Hansen 73], Hoare [Hoare 73] and Dijkstra [Dijkstra 68].

Modules may contain other modules. Remote procedure calls are constrained by the resulting hierarchy: a process in a module may make remote calls on procedures of modules it can name from the module's position in the hierarchy.

(Part of the referencing environment of a module are imported names of some other modules.) The name-scoping rules follow the standard ALGOL-60 convention, except variables inherited from outer scopes and shared among inner modules are restricted to read-only pointers to modules; no other kinds of variables may be inherited. We will see in Chapter 4 the reason for this restriction.

The set of processes allowed to make remote calls on procedures of a module is constrained by the guard of that module. A guard for a module regulates processes entering and exiting crowds [Atkinson and Hewitt 77] of the module. From within a crowd, a process has a particular view of the module. A process may access those parts of the module visible from the view it has of the module. (Atkinson and Hewitt do not provide views for members of their crowds.) The guard of a module, therefore, specifies the conditions under which a process may become a member of a crowd having a particular view. Crowd memberships are established by making requests to the guard which handles them strictly serially. The module's guard has the sole power to suspend and awaken processes on queues to control their membership in crowds. Once a process enters a crowd of the module, it is free to access the visible module parts until it voluntarily leaves that crowd.

A module guard has its own set of private variables to keep track of inter-process interactions. Furthermore, the guard may not access the private variables of the module nor call any of the module procedures. All programmer-specified details of inter-process interactions are therefore isolated in the guards of modules. Module are therefore an improvement over Owicki's shared classes [Owicki 77] in which such details are not clearly isolated in a separate entity. Since only guards can control inter-process interaction, TOMO assumes that when two processes can simultaneously view parts of the same module, their mutual interference will be harmless. If interference is considered harmful, the guards must be coded so that one of the processes gets delayed until the other has left its crowd. Guards are the only means in TOMO to control inter-process interactions.

3.3 Modules

We now look more closely at modules, which play a central role in TOMO. We will see later that module guards can implement various complex control regimes such as asynchronous generators and coroutines, as well as message-based interactions, parallel search schemes, and pipelined data-flow networks [Dennis 73]. Often the private data of a module with such a guard can play a key role in these control regimes. In message-based interactions the private data can

provide message buffers, while in parallel searches they can represent the search space. In pipelined data-flow networks the private data can represent the data paths. Modules can also control parallel access to shared data objects in much the same way as monitors in Concurrent PASCAL [Brinch Hansen 75]. The conditions under which a shared object may be accessed are dependent on the implementation of the object and are of no concern to users of the object. Modules and monitors both separate use of an object from the implementation of both access control and operations on the object.

Monitors, however, are really a special form of a module. They do not provide the flexibility for sharing that seems to be required for many parallel processing applications. A monitor is a module with only one crowd whose membership is limited to at most one process at any time. As stated in Chapter 2, neither parallel read access nor parallel read-write access to non-overlapping areas of the monitor's private data are possible. This restriction results from the unnatural combination of routines for access control, which require sequential execution, with routines for accessing the object, which need not be sequential but are constrained to be because of the former. The only way to accomplish parallel access in Concurrent PASCAL is to put the shared object outside the monitor, divorcing the access

control from the object. The object becomes accessible by all processes and unprotected by the monitor unless all processes obtain permission from the monitor before any access, but the monitor cannot enforce all processes to obtain such permission. Users of a shared object protected by a monitor must thus bear part of the burden of maintaining security.

Modules, on the other hand, permit separation of access control routines from accessing routines, enabling parallel access, but only processes that have been granted permission by the access control routines can access the module's data. Parallel access is possible since several crowds may simultaneously have one or more members. Access is protected since each member of a crowd has a limited view of the module. Any attempt to perform operations not currently visible from the view is a detectable error. There is no notion of such an error built into Concurrent PASCAL.

To reduce traffic through the guard and the overhead of run-time checks, a process may maintain membership in a crowd for a sequence of operations. A sequence of operations can also be made indivisible to a process maintaining crowd membership. Again, monitors (or serializers) do not provide such flexibility.

provide message buffers, while in parallel searches they can represent the search space. In pipelined data-flow networks the private data can represent the data paths. Modules can also control parallel access to shared data objects in much the same way as monitors in Concurrent PASCAL [Brinch Hansen 75]. The conditions under which a shared object may be accessed are dependent on the implementation of the object and are of no concern to users of the object. Modules and monitors both separate use of an object from the implementation of both access control and operations on the object.

Monitors, however, are really a special form of a module. They do not provide the flexibility for sharing that seems to be required for many parallel processing applications. A monitor is a module with only one crowd whose membership is limited to at most one process at any time. As stated in Chapter 2, neither parallel read access nor parallel read-write access to non-overlapping areas of the monitor's private data are possible. This restriction results from the unnatural combination of routines for access control, which require sequential execution, with routines for accessing the object, which need not be sequential but are constrained to be because of the former. The only way to accomplish parallel access in Concurrent PASCAL is to put the shared object outside the monitor, divorcing the access

control from the object. The object becomes accessible by all processes and unprotected by the monitor unless all processes obtain permission from the monitor before any access, but the monitor cannot enforce all processes to obtain such permission. Users of a shared object protected by a monitor must thus bear part of the burden of maintaining security.

Modules, on the other hand, permit separation of access control routines from accessing routines, enabling parallel access, but only processes that have been granted permission by the access control routines can access the module's data. Parallel access is possible since several crowds may simultaneously have one or more members. Access is protected since each member of a crowd has a limited view of the module. Any attempt to perform operations not currently visible from the view is a detectable error. There is no notion of such an error built into Concurrent PASCAL.

To reduce traffic through the guard and the overhead of run-time checks, a process may maintain membership in a crowd for a sequence of operations. A sequence of operations can also be made indivisible to a process maintaining crowd membership. Again, monitors (or serializers) do not provide such flexibility.

A monitor, nevertheless, is still a useful specialization of a module since its private data can be guaranteed to be accessible by at most one process at any time. Many applications require modules with this property. Provisions are therefore made for a special kind of module, called a private module, which has an implicit guard that limits membership in an implicit crowd to at most one process at any time. Only one view of such a module is possible. Remote calls on its procedures will always in effect request of the guard membership in the crowd, while returns from such procedures will immediately remove the calling process from the crowd.

Another useful specialization of modules is the open or unguarded module. As the name suggests, an open module does not have a guard. Such modules enable specification of abstract data types without the complexity imposed by guards when they are not needed.

Open modules, as well as non-module data types, fall into a category called unguarded types. Objects of such types must be part of the private data of exactly one guarded module at a time to prevent uncontrolled concurrent access. The protection domain of a guard of a module extends to all unguarded objects freely accessible (without going through another guard) from that module. When a name of an

unguarded object is passed from one module to another as a parameter of a remote procedure call, the sending module must no longer be able to name the object. Such a mode of passing is called passing by revocation since the ability to name the object is revoked in the sending module.

As mentioned in Chapter 2, independent components of a data structure should be concurrently accessible. Although we could make each component a guarded module, we also desire a method of sharing data structures made up of unguarded components. We provide such a method by implementing the entire data structure in a single guarded module that manages capabilities to the unguarded components. A capability consists of a name for an object and a list of access rights that a process holding the capability may exercise. In TOMO crowds are used instead of access-rights lists. A process may enter a crowd of a capability to get a particular view of the object named by that capability. Capabilities in TOMO also contain a third item, the name of the module whose guard controls processes entering and leaving crowds of the capability. Later sections of this chapter more fully illustrate the use of capabilities.

3.4 Language Overview

We now turn to a more detailed overview of TOMO by presenting a series of examples, starting with relatively simple program segments and proceeding to more complex ones. The aim of this section is not to define the language fully, which is done later, but rather to give the reader an appreciation of the major innovative features of TOMO.

We start by considering a version of the readers-writers problem. A buffer is to be shared concurrently by writer and reader processes. In order to guarantee that the reader processes read "good" data (i.e., data that has not been partially updated), a writer process must have exclusive access to the buffer during its write operations. In order to prevent unnecessary serialization of concurrent tasks, however, reader processes must be allowed to read concurrently.

Program 3-1 in the Appendix gives the solution proposed by Hoare using monitors. Hoare's syntax for monitors [Hoare 74] is used. Buffer is declared outside the monitor, so any process that can name Buffer can perform a write operation on it without first calling StartWrite of the monitor. Such undisciplined access can result in readers reading partially updated data. A compiler cannot generate a

check to ensure that StartWrite was called prior to the write operation. Furthermore, security of the Buffer's data relies on the trustworthiness of each reader and writer, the users of the Buffer, rather than on just the trustworthiness of the implementor of the Buffer or the monitor.

One way for the monitor implementor to maintain security is to declare Buffer inside the monitor. Buffer is now read and written by routines of the monitor that mutually exclude each other. Exclusive write access is guaranteed by the monitor rather than by the writer process. Parallel reads, however, are no longer possible.

We now present an alternative solution using TOMO modules, as shown in Program 3-2 in the Appendix. Before explaining this solution, we pause to describe some of the details of the syntax of TOMO. Module definitions in TOMO are type definitions, analogous to PASCAL record definitions. They consist of two parts: the specification part and the implementation part. The specification part, denoted by "modspec ... end", defines the module's abstract specifications. All names declared in the specification part are exported to the immediately surrounding definitional scope. The type part of the specification defines the parameter structures of all exported procedures. Such types are called <u>routine forms</u>. As will be seen later, types exported

as capabilities and pointers may also be declared in the specification's type part.

The specification's type part is followed by its guard part, which defines the crowds of the module and the routine forms of the guard routines. In the present example there are two crowds, Reader and Writer. The view of Reader is the procedure Read while the view of Writer is Read and Write. Each crowd is controlled by routines Enter and Exit having the routine form EnterExitForm declared in the type part above. Enter and Exit are special names given to procedures that must exist for every crowd. Execution of Enter by a process makes the process a member of the crowd associated with Enter while execution of the corresponding Exit removes the process from that crowd. The guard part is followed by the declaration of exported routine names, which associates them with the routine forms corresponding to their parameter structures.

The purpose of separating the specification part from the implementation of the module is to enable different implementations of the same abstract module specifications [Liskov et al. 77, Shaw et al. 77]. Although not shown in the present example, a name may be associated with a module specification and may be used to declare pointers and formal parameters for modules having the same abstract specifica-

tions but possibly different implementations. Separating module specifications from implementation also allows independent compilation of modules. A compiler need only know the specifications of modules referenced by the module being compiled.

The implementation part resembles a PASCAL program except for the presence of the guard part and the syntax of routine declarations. Although not shown in the present example, a full-fledged module can have a const part, a type part, a var part, a guard part, and a routine part. The var part in the present example contains the declaration of Buffer, the entity being shared. The guard part resembles a module in that it may also have const, type, var and routine parts. The declarations in the const, type and var parts are private to the routines of the guard and implement data structures for storing synchronization information. The var part of the surrounding implementation is also available to the guard routines. In the present example two variables of type Queue, QW and QR, are declared. The type Queue is a predefined module type used to delay processes. Queues are similar to Conditions of Concurrent PASCAL [Brinch Hansen 75] and are described more fully in Chapter 4 under "Predefined Entities".

The routine part of the guard defines the routines that control or interrogate crowds. The Enter and Exit procedures are qualified by the name of the crowd they control. The Boolean function <crowd name>.Empty determines whether the crowd <crowd name> has any members. The guard part in the present example is essentially the monitor of Program 3-1.

The guard part is followed by the routine part, which defines the bodies of the routines that manipulate the data denoted by the var part of the module. The syntax is somewhat different from PASCAL routine declarations. A routine body is considered a constant whose type is its routine form. Pointers to a routine form may be declared in the var part and can point to different routine bodies having that routine form. Since operations to modify or copy routine bodies do not exist, variables of routine form types may not be declared.

We now describe the solution presented in Program 3-2. Buffer is a private variable of the module ReaderWriter, and it may only be accessed via the routines Read and Write. These routines may only be called by processes in the crowds of the module. The Enter procedure for crowd Writer permits at most one process to be a member of Writer, and then only if crowd Reader is empty. Any attempt to call Write outside

of Writer will result in an error that can be discovered at run-time. Parallel reads are possible since Enter for Reader will permit multiple processes to be members of Reader when Writer is empty.

Program 3-3 is a solution to a variant readers-writers problem where all writes must have priority over reads, including any currently executing reads. The solution allows multiple writer processes and is an extension of one given by Lamport in [Lamport 77], which only allows one writer. The module has an empty guard. A module with an empty guard is still a guarded module, but its implementor has determined that any of the module's routines may be executed at the same time as any other without sacrificing security. An open module, in contrast, has no guard (not even an empty one) and its implementor has made no such determination. In this example, therefore, Read and Write are considered safely executable simultaneously. If Write is called while a process is executing Read, the read operation will be repeated. A call to Write while another process is still executing Write will result in both processes repeating the write operation (and possibly looping indefinitely if one process always starts the write operation before the other has completed). This solution does not guarantee that the final data value after such a conflict is that of the most recently arriving writer. Writes that interfere with reads

or other writes are detected by comparing a pair of counter variables, WriteEntryCount and WriteExitCount, which are incremented before and after each write operation respectively. After each operation, the value of WriteExitCount before the operation is compared to the value of WriteEntryCount after the operation. If WriteEntryCount is greater, then a writer must have been present at the start of the operation, or a writer has entered before the completion of the operation. The point of this example is to illustrate that TOMO modules may be used to implement rather different disciplines for sharing data.

Having seen two ways in which a simple buffer can be shared, we now examine the more complex problem of sharing data structures. The operations on a data structure fall into two distinct categories: operations on the structure and operations on components of the structure. The latter category, ideally, should not require knowledge of the entire structure. Each component should be nameable individually; operations should be performed directly on the component rather than indirectly through routines of the surrounding module that implements the containing structure. For example, suppose we wish to implement a symbol table that associates symbols with attributes. Given a symbol, we would like to fetch its associated attribute and operate on it. The operations on the attribute, however, should be in-

dependent of the symbol table structure, whether it consists of binary trees, arrays, or linked lists. For serial programs, this independence is easily achieved by pointers that point directly at the attribute.

For parallel programs, however, it is possible for a structure operation to conflict with a component operation. In addition, it may be desirable to control access to several components as a group according to some property of the surrounding structure rather than individually by component (to provide a coarser grain of sharing). Such sharing by groups eliminates the need to make each component a guarded module. Furthermore, sharing should be controllable at a global level, permitting avoidance of deadlocks due to partial allocation of access rights. Both component and structure operations, therefore, should be done from crowds under the control of the same guard.

Structure operations can be done through routines that require membership in one of the module's crowds, and are therefore controllable by the module's guard. Component operations, however, require the ability to name the component and should be done directly on the component or through routines of the module implementing the component. How can the structure module guard control these operations? TOMO allows the structure module guard to control operations on the

component by managing capabilities. References to components may be distributed to processes by the structure module in the form of capabilities. A process must join a crowd of the capability in order to operate on the component, and membership in the capability's crowds can be controlled by the guard of the structure module. The details of capability management are illustrated in Program 3-4.

Program 3-4 implements a shared linked list intended to be used by Program 3-5 to implement a shared symbol table. Each element of the list is a record containing the fields Symbol, Atr, and Link. Atr is a capability to an attribute node. A capability type is defined using the notation "cap-> <type ID>" and is not compatible for assignment to or from a pointer type to <type ID> except within the module defining the capability type. (The "->" notation is used in place of the standard up-arrow to denote pointers.) The variable Head points to the head of the list. The operations defined for the list are: 1) insert an element at the head, create a new attribute node for it, and return a capability to the attribute node (function Insert); 2) given a symbol, find the element having Sym equal to the symbol and return its Atr field (function Find); 3) read from an attribute node; and 4) write onto an attribute node. The latter two operations are not done by routines of the SharedLIist module but are done directly on the attribute

node by processes in the crowds AttrReader and AttrWriter. These crowds are for the capability type AttrCap defined and exported from the SharedList module. We assume here that Attribute is a record type having AttrField as a field. The view from AttrWriter of Attribute is "var AttrField", meaning AttrField may be written or read by AttrWriter's members, while the view from AttrReader is "const AttrField", which only allows read operations on AttrField. Variables of type AttrCap may be declared outside the module and receive values from Find and Insert. In order for a process to read or write an attribute node, however, the process must first become a member of the proper crowd of the capability for the node. Crowd membership for a capability is managed by the guard of a module instance whose type defines the capability type, in this case SharedList. The particular instance of the module type is determined by where the capability got its value. (A reference to the module instance whose guard manages the capability is an implicit part of the value of a capability.) To become a member of a capability crowd, a process need only name the capability and the crowd names. For example, to enter the AttrWriter crowd for the capability AC that got its value from a call to Find, a process executes "AC.AttrWriter.Enter".

We now look at the compatibility of Find and Insert with respect to their concurrent execution. Find and Insert

cannot interfere with the reading and writing of attribute nodes and can therefore be performed concurrently with reading and writing. Furthermore, the value of Head is not updated in Insert until after the new element is linked to the rest of the list, so Insert and Find can be performed concurrently. Find may also be concurrent with itself since it does not modify any part of the list. Find may therefore be performed at any time independent of other accessing processes. Insert, however, may only be performed when no other Insert is in progress. Attribute reading and writing fall under the same constraints as the readers-writers problem. Instead of controlling access to each individual attribute node, we use a coarser grain of sharing. Whenever a write operation is in progress on one attribute node, no read or other write operations may be perfomed on any of the attribute nodes on the list. This coarser grain of sharing requires fewer queue variables and simpler guard routines at the cost of reduced concurrency. In the symbol table example that follows, the grain of sharing will be each list rather than each element of the list, which is reasonable if the lists are short (as they should be for a symbol table).

Once the compatibility of the operations is determined, the implementation of the guard is relatively easy. Each Enter procedure checks if crowds for operations that conflict are occupied. If so, the requesting process is caused

to wait on a queue for the crowd. Each Exit procedure for a crowd wakes up processes that may have been queued because the crowd was occupied. If two or more processes wishing to perform conflicting operations are waiting, Exit chooses one of them according to some scheduling discipline. In this example as well as in Program 3-2, queued readers have priority over queued writers when a writer exits. Queued writers, however, have priority over queued readers when all readers have exited.

Program 3-5 is a shared symbol table made up of an array, BHT (for Bucket Hash Table), of pointers to SharedList module instances. The implementation uses the standard bucket-hash technique [Knuth 73]. Looking at the top of the module definition, we see that the type part of the specification part exports SymAttr as SharedList.AttrCap. The purpose of this declaration is to rename AttrCap exported from SharedList to be SymAttr. Capabilities to attribute nodes can be declared outside SymTab using SymTab.SymAttr, so the user needs no knowledge that SharedList is used. SymTab has an empty guard since it has no unguarded private variables. The functions Enter and Lookup both call Find of SharedList without joining a crowd of SharedList. (Find does not conflict with the other operations of SharedList so it may be called at any time.) Enter also calls Insert, but only from crowd Inserter since two Insert calls may not execute simul-

taneously. Enter and Exit of Inserter are called implicitly at the start and end of the incrowd statement, respectively. The incrowd statement guarantees that the statement following the "do" executes in the crowd specified, removing the otherwise necessary run-time checks to ensure that Insert is accessible. The grain of sharing, as mentioned above, is at the level of each SharedList module instance. The granularity can be made finer by modifying only the SharedList definition.

Moving down two levels of abstraction, we now consider the attribute node. Although the SharedList module assumed the attribute node was a record, the attribute node could be implemented as a linked structure. In a serial language such as PASCAL, Atr would point directly at one of the components of the attribute structure. In a parallel language, one must be careful to insure that any component of the structure be accessed only through a capability of the type AttrCap. Numerous pointers may be pointing to any of the components of the structure. How can we guarantee that none of these pointers will be used by processes to access the components without first joining the proper AttrCap capability crowd?

One way to control access is to make the attribute also a guarded module, and use capabilities to access the compo-

nents of the attribute's structure. As shared data items get smaller, however, the overhead of instantiating guarded modules for each item could get too high. An alternative is to make the attribute node an open module. Atr would then point to a module rather than one of the components of the attribute structure. The attribute structure would be composed of components whose types are local to the attribute component module. Pointers to components of the structure would be in the form of pointers whose types are exported from the attribute module. Such a pointer may access a component only if it is passed as a parameter to a routine of the attribute module. Therefore, the process must be a member of the proper AttrCap capability crowd before the pointer can be used for access. We guarantee that only AttrCap capabilities may reference an attribute module since attribute module instances are created by the the SharedList module's Insert function, which returns capabilities to the new instances. Therefore, access to any component of the attribute structure can be controlled by controlling AttrCap capabilities.

Suppose instead that Insert did not create attribute modules, but was passed a pointer (not a capability) to one as an additional parameter. Since Insert is a routine of a module that defines AttrCap, it has the power to convert the pointer into a capability of type AttrCap. What if, howev-

er, the attribute node had another pointer pointing to it (i.e., an alias) from outside the module? Access through the capability would not be guaranteed to be safe since another process may freely access the attribute through the alias pointer. Because of this problem, pointers to unguarded objects, including unguarded modules, must be passed in a remote procedure call by revocation: all aliases are revoked when the pointer is passed, leaving the formal parameter the sole pointer to the object. Similarly, when a pointer to an unguarded object is passed out as a parameter or a function return value of a remote call, all aliases must be revoked. The two kinds of passing by revocation that exist in TOMO are "revin" and "revout", and are similar in use to value and result parameters found in other languages.

In the above case, revoking the pointer to the attribute module also effectively disallows any access to component nodes of the attribute through pre-existing pointers exported from the attribute module. Exported pointers must be passed to a routine of the module in order to access the objects they point to. Since all pre-existing pointers to the module have been revoked, such routines can no longer be called except by using a capability obtained from Insert or Find. An alternative to revoking pointers would be to copy the entire pointer structure. Such copying is potentially

expensive, and references to the components of the structure are also lost. Such references may be expensive to reestablish. For example, an attribute may contain information in a component that requires a search to find. Repeated searches can be avoided if a reference to the component is kept in an exported pointer. When an attribute node is passed into Insert, however, such a reference would be lost if the attribute were copied. Passing a pointer to the attribute and revoking pre-existing aliases would preserve the reference.

From the above discussion, we see that unguarded data structures (e.g., the attribute node) can be securely shared by first, implementing them as unguarded modules and second, passing pointers to these modules (and to all other unguarded objects) by revocation in remote procedure calls. Since a pointer to an unguarded module implementing a data structure does not define any structural properties of the structure, passing such a pointer by revocation does not destroy such properties. For example, passing by revocation a pointer to a module implementing a circular linked list does not destroy the lists circularity since the pointer does not point directly at one of the list's nodes but instead, points to the module containing a pointer in its var part that points to one of the list's nodes. Because complex data structures require alias pointers to define their struc-

tural properties, pointers to unguarded modules may not be used for this purpose. Such pointers can only be used for access. A TOMO programmer must clearly distinguish between pointers used for access and pointers used for structural definition. As a general rule, structure defining pointers are limited to pointers that point to types local to the module implementing a data structure. Such local types provide the component nodes of the data structure, and they consist of the component value (or an "access" pointer to one) and other structure defining pointers. A more disciplined use of pointer variables is therefore required in TOMO as compared to languages such as PASCAL.

Sharing has been the emphasis of the examples so far. In the next example, Program 3-6, we illustrate the use of modules to coordinate several processes to perform a sort. The module NetSorter sets up a data-flow network [Dennis 73] in which the integers to be sorted flow along paths of a network of operator nodes, which take the data in the paths as operands. A portion of the network is shown below:

```
                  ...
                     \__
                        __Op[i-1]__...
                     __/          __
P[i]____            /            P[i]
        __Op[i]__
P[i+1]__            \__
                        __Op[i+1]__
                     __/          __
P[i+2]__            /            P[i+2]
        __Op[i+2]__
P[i+3]__            \__
                        __Op[i+3]__...
                     __/
                  .../

                  flow --->
```

Operator Op[i] takes two inputs, P[i] and P[i+1], and swaps their values if the value in P[i] is greater than the value in P[i+1]. The operators are divided into even and odd levels, with the even-level ones feeding their outputs to the odd-level ones. The outputs of the odd-level operators are fed back to the even-level ones (N div 2) times, where there are N+1 integers to be sorted.

The above network is implemented in Program 3-6 using modules for both paths and operators. We make each path module (ArrayElemPath) a guarded module whose guard controls access to the path value (Val) by processes executing in the operator modules (TestAndSwap), which are private modules. The guard of each ArrayElemPath instance regulates the order in which processes executing in TestAndSwap instances may enter crowd Operate. The view from crowd Operate is "var Val", which allows Operate's members to read and write Val. The topology of the network is therefore determined by the order in which the guard of each path module allows processes in operator modules to access each path value.

We now look at procedure Sort of module NetSorter to see more of the programming details. The Init procedure of ArrayElemPath copies each of the N+1 integers to be sorted from array A to variable Val of instances of the module ArrayElemPath pointed to from array Paths. The Init procedure of ArrayElemPath is called automatically when an instance of ArrayElemPath is created by New. The additional parameters to New provide the parameters to Init. The array Paths is then passed to the procedure TestSwap of each module instance pointed to from OperatorSet, a set of pointers to modules of type TestAndSwap. TOMO extends the domain of sets to pointers. Paths is passed by "const", a mode of passing that permits only read operations on the formal pa-

rameter. Instances of TestAndSwap are called through a "forall statement", which is one of the ways of performing a parallel remote procedure call. TestSwap is called in parallel on all instances of TestAndSwap that are pointed to by elements of OperatorSet. The forall statement is repeated (N div 2 + 1) times, after which the integers will have been sorted and are copied out of Paths[i]-> to A[i].

Each instance of TestAndSwap in OperatorSet has been initialized with a value for TSIndex at the time the instance was created. TSIndex is used by procedure TestSwap to index the array Paths in the nested incrowd statements in which Paths[TSIndex]->.Val and Paths[TSIndex+1]->.Val are compared and swapped. Path1 and Path2 are used within the incrowd statements to refer to Paths[TSIndex] and Paths[TSIndex+1]. The nesting of the incrowd statements allows an indivisible sequence of operations to be performed between Path1.Val and Path2.Val. Execution of TestSwap by a process causes that process to either enter the Operate crowds of Path1 and Path2, or to be queued, depending on the value of State. State is a private variable of the guard of ArrayElemPath instances. Processes from TestAndSwap instances with even TSIndex values can enter Operate when State is FirstLev. When such even processes exit Operate, State is set to SecondLev to allow processes from TestAndSwap instances having odd TSIndex values to enter Op-

erate. When the odd processes exit, State is reset to FirstLev. State is initialized to FirstLev when an instance of ArrayElemPath is created.

The above example illustrates a general programming technique for implementing data-flow networks in TOMO. Each operator node of a network can be represented by a private module whose routines take as parameters the guarded modules representing the data paths that provide the operands for the operator nodes. The coordination of the "firing" of each operator node is controlled by the guards of the path modules. Each path module's guard is encoded so that the operator nodes access it in the desired sequence, according to the network path being implemented. The data of each path module flows through this sequence of operator nodes. Each operator node may only fire when it can access all of its operands, i.e., when it has become a member of the proper crowd of each path module that contains one of its operands. The network is free of deadlocks caused by circular waits for inputs if the paths define a partial ordering over the operator nodes. Circular waits for inputs can only occur if two operators occur in different order on two different paths. Partial ordering over the operators of networks containing such paths is not possible.

Multiple instances of each path module may exist simultaneously, with each instance accessed by a different operator node on its path. Passing each set of paths to the operator module procedures as a structured type, such as an array or a record, guarantees that each operator always receives operands from the same set. The network can therefore be pipelined with sets of paths having data flowing at different stages of the network. Furthermore, since flow of data along one set of paths does not logically have to wait for flow along another set of paths, deadlocks cannot arise in a pipelined network if deadlocks cannot arise when the network is not pipelined. The NetSorter module just described will enable any number of sorts to be in progress simultaneously in a queued, two-stage pipeline.

3.5 Sharing of TOMO Objects

From the discussion in 3.4, we began to see how pointer-linked structures can be shared by being careful about how such structures are accessed. The general rules are the following:

1) Variables imported from surrounding modules are restricted to pointers to guarded modules. Imported pointers may not be modified.

2) Parameters to a remote procedure call are restricted to sharable types. A sharable type is either: a) a type not containing pointers to unguarded types (open modules or non-module types); b) a capability; c) a module; or d) a pointer to a, b, or c (referred to as a sharable pointer). Shareable pointers to unguarded types must be passed by revocation in a remote procedure call. Functions returning shareable pointers to unguarded types must return them by revocation.

3) Unguarded types must be passed by const or value in a parallel remote procedure call to prevent unguarded concurrent write access. Shareable pointers to unguarded types may not be passed in a

parallel remote procedure call, since passing by revocation in such a call does not guarantee freedom from aliases.

4) Unguarded modules may export pointer types to enable external referencing of components. Exported pointer variables may not be dereferenced outside the exporting module. The objects they point to may only be accessed by passing them back to routines of the module from which they received their value. Exported pointers need not be passed by revocation.

5) Capability types may be defined and exported from guarded modules. Guard routines for their crowds must be defined in the guard part of the module defining the capability type.

The above rules are sufficient to ensure that objects are "safely shared". The term "safely shared" will be defined after some preliminary definitions. An object is said to be accessible from a module instance if and only if it is referenced through a private variable of the module and routines of the module can operate on the object. An object is exclusively accessible from a module instance if and only if it is accessible from only that module instance. An object

is remotely exclusively accessible from a module instance M if and only if it is exclusively accessible from M, or it is remotely exclusively accessible from a module instance that is exclusively accessible from M.

An object is safely shared if and only if one of the following is true:

1) The object is a guarded module.

2) If the object is of an unguarded type, either it is remotely exclusively accessible from a guarded module instance at all times, or it is accessible only through capabilities from outside the module that otherwise has exclusive access to the object.

A safely shared object, therefore, is always protected from concurrent access by a unique guard; no two guards independently control access to a safely shared object. All processes are constrained to join a crowd protected by that unique guard in order to access the safely shared object.

In order to show that the rules listed above are sufficient to guarantee that all accessible objects in TOMO are safely shared, we must show that at least one of the two

conditions just listed holds for each accessible object. We proceed by first showing that all accessible TOMO objects that are neither guarded modules nor have capabilities to them (i.e., unguarded objects) are exclusively accessible from a module at all times. Since only guarded modules are accessible through imported global variables, unguarded objects are not accessible simultaneously from several modules through global variables. A module, therefore, may only access unguarded objects it has created or it has received through parameters of remote procedure calls. Unguarded objects may only be passed to other modules if they are sharable types. Shareable pointers to unguarded types must be passed by revocation in a serial remote procedure call, and may not be passed at all in a parallel remote procedure call. Since all unguarded objects are exclusively accessible when they are created, passing sharable pointers to such objects by revocation preserves their exclusive accessibility. Exclusive accessibility is also preserved if an exported pointer references an unguarded object, since such a pointer must be passed to a routine of the exporting module (which has exclusive access to the object) in order to operate on the referenced object. All unguarded objects are therefore exclusively accessible from a module instance at all times.

From the definition of "exclusively accessible" it follows that if a module A is exclusively accessible from a module B, and B is remotely exclusively accessible from a module C, then A is remotely exclusively accessible from C. We have just shown that all unguarded objects are exclusively accessible from a module at all times. Therefore, unguarded objects are remotely exclusively accesible from a guarded module at all times, assuming that the root of the module hierarchy is guarded.

If the object is not remotely exclusively accessible from a guarded module, it must be either a guarded module or accessed through capabilities. We need to show in the latter case that if the object named by a capability is an unguarded type, the only pointers that point to the object are those from the module defining the capability type. We have already shown that an unguarded object is always exclusively accessible to a module. Guarding such an object with a capability does not allow other modules to have pointers to the object, since assignment of a capability to a pointer outside the capability's defining module is not allowed. Objects of unguarded types accessed through capabilities are therefore accessible by pointers only from the module defining the capability type.

All objects in TOMO, therefore, are safely shared. Every accessible TOMO object is protected by a unique guard. Furthermore, the restrictions imposed to provide exclusive accessibility are not unreasonable. The restrictions insist that all pointer structures that are to be shared be defined using modules, and that access to components of the structure be allowed only through capabilities and exported pointers. Since a module can be used as a data abstraction mechanism, such restrictions do not limit the power of the language any more than limitations imposed by data abstraction mechanisms. The management of pointer structures through such abstraction mechanisms is a highly desirable programming practice and has been shown to be quite feasible [Liskov et al. 77, Shaw et al. 77].

3.6 Control of TOMO Processes

As seen in the data flow example, TOMO provides flexible constructs for generating parallel processes and coordinating their interactions. The forall statement provides a convenient method of specifying parallel execution of the same procedure over members of a set of module instances. TOMO also allows specification of parallel execution of different procedures over a fixed number of module instances of different types, as will be seen in the next chapter. As stated earlier, a new process is created in each module on

which a parallel call is made. Once execution of these processes has started, inter-process interaction may take place using guarded module instances that are common to the interacting processes. Several processes may compete for membership in crowds of a guarded module, and the guard of such a module can schedule when certain processes may enter the crowds. For example, the guard of a path module for a data-flow network sequences processes in operator modules competing to join a crowd for a view of the path value. A set of parallel processes therefore commutes among different module instances, and guards control inter-module transfers of these processes to achieve cooperation. Queues and other variables required to implement the cooperation of processes are isolated in the guards. Therefore, guards are a control abstraction mechanism; the details of how coordination is achieved is hidden from the modules whose processes request coordination. Chapter 5 more fully discusses the use of guards as control abstraction mechanisms.

The collaboration of processes set up by a parallel call is similar to execution of a Concurrent PASCAL program, where processes interact using monitors. A TOMO program consists at run time of a dynamic hierarchy of collaborations, whereas execution of a Concurrent PASCAL program is one static collaboration. The hierarchy is created by a tree of parallel calls that are in progress. In

Concurrent PASCAL, only one parallel call would be in progress.

3.7 Communication in TOMO

Communication in TOMO is achieved by two methods. First, remote procedure calls provide a restricted form of message passing between module instances. Message passing following the procedure call-return form seems common enough to warrant inclusion of such a form as a primitive. Many examples of programs written in message-based languages use the more primitive Send and Receive constructs for nothing more than implementing procedure calls. One process sends a message to another and then immediately waits for a reply. The process receiving the message will service the message, send a reply, and then loop back to wait for another message. Even though two processes are involved, only one is executing at any time. In TOMO such interaction takes place as a remote procedure call where a process in one module jumps to another module to execute a procedure; only one process is involved.

Interactions among parallel processes that do not follow the procedure call-return form, however, are also necessary. Guarded modules can provide a communications channel between two processes executing in different modules. Prim-

itives such as Send and Receive may be implemented as procedures of such guarded modules. Unlike PLITS and other message-based languages, TOMO provides remote procedure calls and Queue variables for implementing message primitives, whereas PLITS and others provide message primitives for implementing procedure calls and process suspensions on queues. This approach was taken in order to allow more programmer control over how messages are passed. We will see in Chapter 5 how guarded modules are used to provide the kinds of message passing required in the AI applications discussed in Chapter 2. For simple message passing, however, TOMO provides a new structuring method called a Channel (a FIFO sequence of objects, similar to PASCAL files) to pass object messages from one module to another. Chapter 4 describes Channel in more detail.

3.8 Conclusion

This chapter has presented an overview of a base language for parallelism centered around the concept of processes executing in modules and interacting by remote procedure calls. The guard of a module may constrain a process' access to the module, and can be quite powerful in achieving cooperation among processes. Complex pointer structures may be shared without aliasing problems by constraining all unguarded objects to be exclusively accessible to a module.

Chapter 4
TOMO Specifications

4.1 Introduction

The detailed specifications of TOMO are presented in this chapter. The presentation assumes the reader's understanding of Chapter 3 in which we provide motivation for the constructs of TOMO. The discussion on exclusive accessibility of unguarded objects is especially important for understanding the design of the constructs. We also assume the reader's thorough familiarity with TOMO's base language PASCAL. We describe those parts of TOMO that are different from or new to PASCAL. The reader may assume that PASCAL constructs not mentioned in the specifications exist unaltered in TOMO.

The language constructs are presented in the following way: For each set of related constructs the syntax is presented in a modified BNF, followed by a description of the semantics of the constructs in English using the terminology developed in Chapter 3. Standard BNF is augmented by "{ <symbols> }", which specifies repetition of <symbols> zero or more times. We describe TOMO in a top-down order, starting with programs and then proceeding through each of a module's parts (a program being an instance of a module).

As with any language definition, the following sections are plagued with forward references that foil any attempt to maintain coherence.

4.2 Programs

Syntax:

```
<program> ::= <program ID> : <module type>
```

Semantics:

A <program> is an instance of a <module type> (see 4.3). It is automatically an inner module to the operating system, which defines the root of the module hierarchy. The operating system may contain other instances of inner modules for library and run-time support.

4.3 Module Types

Syntax:

```
<module type> ::= module <module spec part> ;
                    <const part>
                    <type part>
                    <var part>
                    <guard part>
                    <routine part>
                  end
```

Semantics:

A <module type> is a definitional scope containing declarations for data, crowds, and routines. The <type part> of a <module type> or of a routine may contain other <module type>s, creating a static hierarchy of definitional scopes. A <module type> implicitly imports constants, types, variables, crowds, and routines declared in surrounding <module type>s. We restrict the importation of variables, however, to pointers to guarded module types. Other variables may not be imported since neither they nor objects they reference are protected from concurrent access. Further, the imported pointers may not be modified within the <module type> (they are "constant" pointers), since processes in several

module instances may be sharing the same pointers. The imported variables are bound to outer-scope variables at the time the module is created (see New in 4.15). Constants, types, variables, crowds, and routines may be exported to the surrounding <module type> by specifying them in the <module type>'s <module spec> (see 4.4). Restrictions on exporting are explained in 4.4.

Instances of <module type>s are created dynamically at run time using the predefined New procedure (analogous to PASCAL's New; see 4.15). Unlike PASCAL types, a <module type> may only be used to declare pointers that will point to instances of the <module type>; variables of <module type> may not be declared. Furthermore, copying of entire module instances may not be done by assignment statements, but requires use of the ModuleCopy procedure that is predefined for each <module type> (see 4.15).

The above restrictions are made to avoid difficulties with modules when combined with PASCAL constructs that copy variable values, such as assignment statements and value parameters. Copying a module instance is a complex operation compared with copying instances of PASCAL types. First, in order to maintain exclusive accessibility of unguarded objects reachable from the module instance, all such unguarded objects must be copied. Second, the copy must be made while

no other process is modifying the instance or any object reachable from it. Finally, since the crowds and queues of the copy are empty of processes, the guard of the copy must be initialized to reflect this state. We do not want simple constructs such as assignment statements and value parameters to invoke such complex operations.

The creation of a module instance may also be a complex operation since initialization procedures must be invoked. If module variables could be declared, we again would invoke complex operations by a simple construct: a call to a procedure having a module as a local variable. For these reasons, module instances must always be created and copied explicitly using predefined procedures (see 4.15).

The parts of a <module type> are discussed separately in the following sections.

4.4 Module Specifications

Syntax:

```
<module spec part> ::= <type ID>
                     |  <module spec>
<module spec> ::= <module kind> modspec
                    <const part>
                    <type part>
                    <var part>
                    <guard spec>
                    <routine spec part>
                  end
<module kind> ::= <empty>
                |  open
<routine spec part> ::= routine
                          <routine spec> { <routine spec> }
<routine spec> ::= <routine ID list> : <type ID> ;
<routine ID list> ::= <routine ID> { , <routine ID> }
```

Semantics:

The <module spec part> defines the abstract specifications of a <module type> by listing names exported to other <module type>s. All names declared in a <module spec> are exported to the surrounding <module type>. Each <module

type> is considered to be an implementation of the abstraction specified by its <module spec>. If a <type ID> is defined to be a <module spec>, several <module type>s having the same <module spec> may be declared, enabling several different implementations of a module having the same abstract specifications. Names of <module spec>s may also be used to declare pointers that point to any <module type> having the same <module spec>. Such pointers are particularly useful as formal parameters, allowing routines to accept actual parameters of several <module type>s having the same <module spec>. Variables of type <module spec> may not be declared since a <module spec> does not fully define a type.

The <module kind> specifies a module as open or guarded. An open module does not have a guard protecting it. As we will see in later sections, certain restrictions apply to open modules to guarantee that they are not arbitrarily accessible concurrently.

The <type part> specifies types exported from the module. Exported types may be used to declare variables outside the module. Since the internal structure of an exported type is always hidden outside the exporting module, operations on variables of exported types are restricted to passing them as parameters to routines of the exporting mod-

ule. Exported routine forms and capabilities are exceptions to this restriction and are more fully discussed in 4.6.1 and 4.6.4. The purpose of exporting types is to enable a module to implement operations for a collection of objects as well as for just one object. We have borrowed this idea from EUCLID [Lampson et al. 77] to allow the flexiblity to associate operations with a type (as in CLU [Liskov et al. 77]) as well as with an object (as in SIMULA 67 [Dahl et al. 70]). As discussed in in Chapter 3, we have also applied type exportation to achieve individual protection of data structure components through use of exported pointers and capabilities.

The exportation of variables by the <var part> enables direct access to module variables without calling module routines. There are many cases when such simple access is sufficient. Access to an exported variable, however, must be done from a crowd whose view includes the variable and specifies whether only read access or read-write access is permitted (see 4.8). The types of exported variables are restricted to non-local (imported) types or exported local types. Exported variables of exported types must be passed to routines of the exporting module in order to be accessed, since their internal structure is not known outside the module.

The <guard spec> gives the abstract specifications of the module's guard and is more fully explained in 4.8.1.

The <routine spec part> defines the names and <routine form type>s (see 4.6.1) of the routines callable by processes in one of the module's crowds through <remote calls> (see 4.12). The <routine form type>s must have been declared in the <type part> of the <module spec>. The bodies of the routines will be specified in the <module type>s having the <module spec> as their specification.

4.5 Named Constants

Syntax:

```
<const part> ::= <empty>
               | const
                    <const def> { <const def> }
<const def> ::= <const ID> = <const> ;
```

Semantics:

The <const part> is identical to PASCAL's constant definitions.

4.6 Types

Syntax:

```
<type part> ::= <empty>
              |  type
                   <type def> { <type def> }
<type def> ::= <type ID> = <type> ;
<type> ::= <routine form type>
         |  <module spec>
         |  <module type>
         |  <channel type>
         |  <pointer set type>
         |  <capability type>
         |  <exported type>
         |  <read only pointer type>
         |  <PASCAL type>
```

Semnatics:

As we have seen in 4.3 and 4.4, TOMO augments <PASCAL type>s with <module type>s and <module spec>s. TOMO also has various other extensions to normal <PASCAL type>s that are explained below.

4.6.1 Routine Forms

Syntax:

```
<routine form type>
          ::= procedure <parm spec part>
           |  function <parm spec part> : <return type spec>
<parm spec part> ::= <empty>
                  |  ( <parm spec> { ; <parm spec> } )
<parm spec> ::= <mode> <ID list> : <type ID>
<mode> ::= <empty>
         |  var
         |  const
         |  revin
         |  revout
<ID list> ::= <ID> { , <ID> }
<return type spec> ::= <type ID>
                     |  revout <type ID>
```

Semantics:

The <routine form type> defines the parameters and the return type (if a function) of a routine. It is to be used in <routine def>s (see 4.9), and in the declaration of routine reference variables (pointers to <routine form type>). Such variables may point to any routine having the same

<routine form type> and are particularly useful for passing routines as parameters. Routine references are created by the predefined function Ref that takes a routine name and returns a reference to the named routine (see 4.15). Exporting a <routine form type> from a module allows the module's routines having that form to be called from outside the module. An exported <routine form type> may not be used, however, to declare routine references outside the exporting module. Dangling routine references are avoided by this restriction. Variables of <routine form type> may not be declared since routines may not be copied nor modified.

The <parm spec part> specifies the <mode>, <ID>, and <type> of the formal parameters. Three <mode>s not found in PASCAL are "const", "revin", and "revout". The "const" <mode> restricts the formal parameter to operations that do not modify it: The parameter is considered constant within the routine. If the parameter is a pointer type, the const <mode> does not mean that the object pointed to is constant. One can achieve this additional restriction by making <type ID> of the <parm spec> a read-only pointer.

The revin and revout <mode>s are used for pointers to unguarded shareable types; they are called passing by revocation. Revin is similar to a value parameter, while revout is similar to a result parameter. When a pointer is

passed in one of these <mode>s, all other pointers pointing to the object it points to (aliases) are revoked. The revin <mode> guarantees that at routine invocation only the formal parameter points to the object, while the revout <mode> assigns an alias-free pointer to the actual parameter when the routine returns. No values are passed in with revout and no values are passed out with revin. The revout mode can also be specified for a function's <return type spec> to guarantee alias-free return values. The revocation <mode> is required in a remote call (see 4.12) if the parameter is a pointer to an unguarded, shareable type.

Revin and revout preserve exclusive accessibility of unguarded objects (see Chapter 3). If an object is initially exclusively accessible, passing its pointer by revocation will continue to keep the object exclusively accessible. Since guarded modules are not guaranteed to be uniquely accessible, we do not allow pointers to them to be passed by revocation. If such parameter passing were allowed, we could revoke a pointer to an object concurrently undergoing access from another module. Revoking pointers to non-exclusively accessible objects is therefore not safe. We do not, however, prevent passing a pointer by revocation from a module in which more than one process may execute concurrently. The guard of such a module can be specified

to prevent a process from accessing an object whose pointer is being revoked by another process in the same module.

4.6.2 Channels

Syntax:

<channel type> ::= channel of <type ID>

Semantics:

The <channel type> is similar to the PASCAL file type. It represents a sequence of <type ID> and is intended for passing objects of <type ID> from one module to another. Six routines, Init, More, Put, PutFin, Fin, and Finished, are associated with each <channel type>. These routines are called as if <channel type> were a module type (e.g., C.Init, where C is an instance of <channel type>). Procedure Init puts the channel (an instance of <channel type>) in the "active" state and initializes the sequence to empty. Procedure Put(Obj : <type ID>) places Obj at the end of the sequence. When the channel is in the "active" state, the Boolean function More(var Obj : <type ID>) removes the object at the head of the sequence, assigns it to Obj, and returns true. If the sequence is empty, execution of More is delayed. More returns false with Obj unaltered if the chan-

nel has been put in the "finished" state by procedure PutFin or Fin. PutFin causes the channel to ignore all subsequent calls to Put and to assume the "finished" state after the last object in the sequence has been removed by More. Fin causes immediate transition to the "finished" state. Execution of Put, PutFin, or Fin while the channel is in the "finished state" has no effect. Finally, the Boolean function Finished returns true only if the channel is in the "finished" state. Variables of <channel type> are references to channel instances. Assignment of a channel variable therefore does not create a copy of the channel instance referenced by the variable; the reference is copied, instead.

4.6.3 Pointer Sets

Syntax:

<pointer set type> ::= set of <pointer type ID>

Semantics:

The <pointer set type> extends the domain of PASCAL sets to include pointers. Such a type is needed to collectively name a number of distinct object instances, such as modules. Operations for a pointer set include those for

PASCAL sets. Examples of pointer set usage are in the <forall statement> (see 4.11) and in guards to keep track of views. Insertion of Nil into a pointer set is a null operation, while an attempt to insert an invalid pointer is an error. A pointer is removed from a pointer set if the pointer becomes invalid (as a result of revocation).

4.6.4 Capabilities

Syntax:

<capability type> ::= CAP -> <type ID>

Semantics:

The <capability type> is a type exported from a <module type> and may only be defined in the <type part> of a <module spec> for a guarded, non-private module. Capabilities may not be defined in other kinds of modules since guards are required to protect the objects they name. Capabilities provide protected references to members of a collection of objects, where the protection is managed by the guard of the module implementing the collection rather than by a guard on each member object. A <capability type> consists of three parts: an <u>object-name</u> part, a <u>guard-name</u> part, and a <u>rights</u> part. The guard of the module named by the guard-name part

controls access to the object named by the object-name part. The guard-name part always points to the module instance that defines the capability. Access is controlled by regulating membership in the capability's crowds, which make up the capability's rights part. Capabilities whose object-name part refers to the same object share the same set of crowds (the same rights part). A capability is identical to a pointer when used within the module defining it. Outside the defining module, a capability and a pointer to the same type are not compatible types. Two <capability type>s for the same <type ID> exported from different <module type>s are considered to be distinct types, since the capabilities' guards have different specifications. Use of capablilities is more fully discussed in Chapter 3.

4.6.5 Exported Types

Syntax:

<exported type> ::= <module type ID> . <type ID>

Semantics:

The <exported type> is used to refer to exported types outside the exporting module named by <module type ID>. The <type ID> must have been declared in the <type part> of the

<module type ID>'s <module spec>. Exporting of types was discussed in 4.4.

4.6.6 Read-Only Pointers

Syntax:

<read only pointer type> ::= readonly -> <type ID>

Semantics:

The <read only pointer type> defines a pointer type that may be dereferenced only for read operations. Objects accessed through such a pointer may not be modified. The same objects, however, may be modified through alternate access paths not involving read-only pointers. Since execution of a module routine cannot be guaranteed to be read-only, such pointers may not point to <module type>s. Assignment to a normal pointer from a read-only pointer or from any pointer reached through a read-only pointer is not allowed. Normal pointers, however, may be assigned to read-only pointers. Read-only pointers are particularly useful as formal paramters to routines to assure the invariance of referenced objects within the routine (assuming the routine does not reference such objects through another access path, such as global variables). Read-only pointers are to be

distinguished from pointers that are const parameters (see 4.9).

4.6.7 Shareable Types

As discussed in Chapter 3, types are classified into "shareable" and "unshareable" types. Shareable types have the characteristic that they are either guarded or access to an instance cannot be made from more than one module instance simultaneously without using capabilities. Only a shareable type may be passed as a parameter in a <remote call>, and if the type is a pointer to an unguarded type, the pointer must be passed by revocation (see 4.12). These restrictions are required to guarantee exclusive accessibility of unguarded objects, as discussed in Chapter 3.

The following are the shareable types:

1) Modules.
2) Unguarded-pointer-free types.
3) Capabilities.
4) Pointers to 1 through 3.

An unguarded-pointer-free type is any type which does not contain pointers to unguarded types. Chapter 3 contains a more complete discussion of sharing in TOMO.

4.7 Variables

Syntax:

```
<var part> ::= <empty>
             | var
                 <var decl> { <var decl> }
<var decl> ::= <ID> : <type> ;
```

Semantics:

The <var part> is identical to that found in PASCAL.

4.8 Guards

4.8.1 Guard Specifications

Syntax:

```
<guard spec> ::= <empty>
               | guard
                   <crowd controller spec>
                   { ; <crowd controller spec> }
                 end
<crowd controller spec>
          ::= <crowd ID> : <capability prefix> crowd <view>
                <routine spec part>
              end
            | private
<capability prefix> ::= <empty>
                      | <capability type ID> .
<view> ::= ( <exported item> { , <exported item> } )
<exported item> ::= <routine ID>
                  | <export mode> <variable ID>
<export mode> ::= var
                | const
```

Semantics:

The <guard spec> defines the abstract specifications of the module's guard routines. The <guard spec> is <empty> when <module kind> specifies an open module. The <crowd controller spec> specifies for each crowd a <crowd ID>, the <view> members of <crowd ID> have of the module, and the form types of the routines that manipulate processes entering and exiting the crowd. A <crowd controller spec> for a capability crowd must have the keyword "crowd" prefixed by the capability type name. The <view> specifies the names of routines and variables that are visible to processes that are members of the crowd. The names must be a subset of the names declared in the same <module spec> or names declared as part of a component object named by a capability. Any attempt to use a name outside a crowd from which the name is visible is an error. Names of exported variables must be qualified by "var" or "const" to indicate whether the variables may be modified outside the module. A routine name declared in a <module spec> may appear in the <view> of a crowd of a capability also declared in that <module spec>. In such a case, the capability may be passed as a parameter to the routine by processes in the crowd. Routines and variables declared in <module spec> that are not in any <view> are visible to all processes. A special <crowd controller spec>, "private", specifies the module as private. In a

private module, no other <crowd controller spec> may be specified.

Each crowd must have two distinguished routines called "Enter" and "Exit" that control entry and exit of processes into and out of the crowd. Execution of Enter for a crowd by a process makes the process a member of the crowd, while executing Exit removes the process from the crowd. Enter and Exit may be called explicitly like any other routine, or they may be called implicitly by executing an <incrowd statement> (see 4.10).

4.8.2 Guard Definitions

Syntax:

```
<guard part> ::= <empty>
               |  guard
                    <const part>
                    <type part>
                    <var part>
                    <guard routine part>
                  end
<guard routine part>
          ::= routine
                <guard routine def> { <guard routine def> }
<guard routine def> ::= <crowd ID> . <routine def>
                      |  <routine def>
```

Semantics:

The <guard part> is an implementation of a guard specified by the <guard spec> in the module's <module spec>. Routines for a particular crowd must have their names prefixed with the crowd name they are associated with. Such prefixing is not done for non-crowd-specific routines. Only routine names declared in the <guard spec> are callable from outside the <guard part>. Guard routines may only access

variables declared in the <var part> of the <guard part>, and such variables may not be accessed from outside the <guard part>. Implementation of access control is therefore hidden from the rest of the module. Execution of guard routines mutually exclude one another in time. Processes calling a guard routine are implicitly queued on a FIFO queue to guarantee mutual exclusion. The mutual exclusion is released when the routine returns or when the process calls Wait on a Queue module (see 4.15).

4.9 Routines

Syntax:

```
<routine part> ::= routine
                     <routine def> { <routine def> }
<routine def> ::= <routine ID> : <routine form type part> ;
                    <label part>
                    <const part>
                    <type part>
                    <var part>
                    <routine part>
                    begin <statement list> end ;
<routine form type part> ::= <type ID> <parm ID list>
                           |  <routine form type>
<parm ID list> ::= <empty> | ( <ID> { , <ID> } )
```

Semantics:

The <routine part> defines a set of <routine def>s. Like modules, routines also set up a definitional scope. Each <routine def> must be associated with a <routine form type>, either by a <type ID>, or by an explicit <routine form type>. When a <type ID> is used, the declaration of the <type ID> must be in the same definitional scope as the <routine def>. With such a restriction, passing a routine reference as a parameter does not require passing the routine's environment since the routine reference can only be passed to scopes where its form can be named (that is, the same definitional scope or lower, where its environment will always be accessible). In LISP terminology, "upward FunArgs" are not allowed in TOMO.

The <parm ID list> following <type ID> in the <routine form type part> is required to locally name the routine's parameters. The names may differ from the corresponding names in the <routine form>. The <mode>s and types of the parameters are not specified in the <parm ID list>, but take on those specified in corresponding positions in the <routine form>. The purpose of this requirement is to make the <routine def> more readable by locally specifying parameter names rather than having the specifications appear only in the <routine form>, potentially far removed from the <routine def>. We do not feel, however, that the <mode>s and types of parameters add significantly enough to the readability of the <routine def> to warrant their repetition in the <parm ID list>.

The parts following the <routine form type part> are similar to those found in PASCAL routine definitions.

4.10 Statements

Syntax:

```
<statement> ::= <foreach statement>
              | <remote call>
              | <parallel call statement>
              | <incrowd statement>
              | <PASCAL statement>
```

Semantics:

PASCAL statements have been augmented by statements for iterating through pointer sets, for generating remote procedure calls, and for controlling access to crowds. The following sections will discuss these statements.

4.11 Foreach Statements

Syntax:

<foreach statement> ::= foreach <ptr ID> in <pointer set> do
<statement>

Semantics:

The <foreach statement> sequentially executes <statement> for each member of <pointer set>. Modifications made to <pointer set> while a <foreach statement> is executing do not affect the execution of the <foreach statement>. The order in which members of the <pointer set> are enumerated is undefined.

4.12 Remote Procedure Calls

Syntax:

<remote call>
::= <module ptr name> -> . <routine ID> <parms>

Semantics:

The <remote call> transfers a process from one module instance to another along with optional <parms>. In the case the <remote call> is part of a parallel call statement (see 4.13), a new process is created in the called module. The <parms> are restricted to shareable types (see 4.6.7) nameable outside the <module type> of <module ptr name>. The parameter passing <mode> is restricted according to the type of the parameter and the nature of the call (parallel or serial). Restrictions on parameters to parallel calls are discussed in 4.13. In a serial remote procedure call a pointer to an unguarded type must be passed by revin or revout to preserve exclusive accessiblilty of unguarded objects (See Chapter 3 for the importance of exclusive accessibility of unguarded objects.) Other shareable types may be passed by value, var, or const in a serial remote call. Routine references may not be parameters to remote calls since they are not shareable types.

A remote call on a routine may only be performed by a process that is a member of a crowd from which the routine name is visible. An attempt to remotely call the routine from outside such a crowd is an error.

4.13 Parallel Call Statements

Syntax:

```
<parallel call statement> ::= <forall statement>
                            |  <cobegin statement>
<forall statement>
          ::= forall <module ptr ID> in <pointer set> do
                <remote call>
                <alongwith part>
<alongwith part> ::= <empty>
                  |  alongwith <statement>
<cobegin statement> ::= cobegin
                          <remote call list>
                          <alongwith part>
                        end
<remote call list> ::= <remote call> { ; <remote call> }
```

Semantics:

The <remote call>s of the <parallel call statement> create new processes that execute in parallel with each other and with the optional <alongwith part>. The <forall statement> performs a parallel remote procedure call on a particular routine of each module pointed to by <pointer set>. The <cobegin statement> performs a parallel remote

procedure call on routines of the <remote call list>. Certain restrictions exist on the parameters of the parallel remote call. In order to prevent concurrent write access to parameters, the const mode is required for all parallel call parameters. Pointers to unguarded types are not allowed altogether in parallel calls, since the objects they point to will no longer be exclusively accessible. The <alongwith part> may not modify any pointers to guarded modules, since such pointers may be shared by module instances in which the parallel processes are executing. The return of each <remote call> of a <parallel call statement> terminates the process created by the <remote call>. Upon termination, a process automatically exits all crowds it has entered. A <parallel call statement> terminates when all calls have returned.

4.14 Crowd Statements

Syntax:

```
<incrowd statement> ::= incrowd <crowd name> <using part> do
                          <statement>
<crowd name> ::= <module or capability name> . <crowd ID>
<using part> ::= <empty>
               |  using <ID>
```

Semantics:

The <incrowd statement> guarantees a process executing the <statement> is a member of the crowd specified by <crowd name>. Before the <statement> is executed, a check is made to see if the process is a member of the crowd. If not, the Enter procedure for the crowd is implicitly called. After the <statement> has executed, the Exit procedure for the crowd is implicitly called if the process was not originally a member of the crowd at the begining of <incrowd statement>. The optional <using part> allows binding of the <module name> to a simple <ID>. The <ID> can be used inside <statement> to prefix exported names of the module. The <ID> is especially useful in nested <incrowd statement>s using <module name>s of the same <module type>. The <ID> is local to the <incrowd statement> and is implicitly declared (like ALOGOL-68 for-loop indices [Van Wijngaarden et al. 76]). If the <using part> is empty, the module's exported names are implicitly prefixed by <module name> (as with the PASCAL with statement). Calls to the crowd Exit procedure are not allowed within <statement> or any procedures called within it.

4.15 Predefined Entities

New, Init, Dispose

New is a procedure analogous to PASCAL's New procedure. It takes a non-exported pointer (not a capability) and allocates an instance of the type pointed to. Its only difference from PASCAL's New is with respect to allocation of module types. If the module type has a routine called Init, the routine is automatically executed as part of New. Its actual parameters may be specified as extra parameters to New, analogous to tag values for records with variants. The main purpose of Init, of course, is for initialization of the module.

The procedure Dispose is similar to Pascal's Dispose. The object pointed to by its parameter becomes inaccessible from the module instance from which Dispose on the object was called. Like New, the parameter must be a non-exported pointer and not a capability. Since all unguarded objects are exclusively accessible from a module instance, resources for such objects (such as storage and processors) may be freed immediately after Dispose has been called on them. Resources for a guarded module may be freed only when Dispose has been called on a pointer to the guarded module from every module instance containing a pointer to the guarded module.

ModuleCopy

ModuleCopy may be used to create copies of module instances. It is a predefined function of every <module type> and may only be called when no other process is a member of any crowd of the module instance being copied. (ModuleCopy is executed from a special ModuleCopyCrowd.) ModuleCopy takes one parameter, a pointer to the module instance to be copied, and returns a pointer to the copy it creates. The entire var part of the module plus all objects reachable through pointers in the var part are copied. If other modules are reachable through pointers, ModuleCopy is also called on them. If the module is guarded, the var part of the guard is initialized in the copy by the programmer-provided procedure InitGuard, which is implicitly called by ModuleCopy. Initialization of the guard is necessary since the crowds and queues of the copy are free of processes, and the guard's var part must reflect such a state.

<crowd name>.Enter, <crowd name>.Exit, <crowd name>.Empty

The procedure <crowd name>.Enter is a procedure of the <guard part> of a module. Its execution by a process results in the process becoming a member of the crowd <crowd name>. Each crowd must have exactly one Enter procedure. Calls on Enter are queued until no other routine in the <guard part> is executing. The procedure <crowd name>.Exit

is identical to <crowd name>.Enter except its execution by a process results in the process leaving the crowd. Enter and Exit may be explicitly called or called implicitly as a result of an <incrowd statement>. Exit for a crowd may also be called implicitly when a process terminates without leaving the crowd. The actual parameters of an implicit Exit call are those that were bound when the corresponding Enter was called. Consequently, Enter and Exit must have the same routine form.

The Boolean function <crowd name>.Empty indicates whether crowd <crowd name> has any processes in it. It may only be called from routines of the <guard part> controlling membership of crowd <crowd ID>. A process is a member of a crowd between the time it returns from Enter and the time it calls Exit.

Queue

Queue is a predefined <module type> having parameterless routines Wait, Signal, Signalled, and Empty. Unlike other <module type>s, variables of type Queue may be declared. When a Queue variable is copied, however, a new instance of Queue is not created, but a reference to the original Queue instance is copied. Queue variables may only be declared in the <guard part> of a <module type> and therefore, its routines may only be called from within the <guard

part>. The Wait procedure suspends a process on a FIFO queue and releases mutual exclusion on the <guard part>. The Signal procedure will take the process at the head of the FIFO queue and allow that process to return from the Wait procedure that suspended it on the queue. Such a signalled process will have priority over all other processes waiting to execute a guard routine (with the exception of earlier signalled processes). Signal called on an empty Queue is a null operation. Unlike Signal in Concurrent Pascal, Signal in TOMO does not imply immediate return of the guard routine. The Boolean function Signalled indicates whether a process at the head of the queue has been signalled by a previous statement of the same guard routine. The Empty function returns true only if the FIFO queue is empty.

Ref

Ref takes a name of a routine as a parameter and returns a reference to that routine. The value returned by Ref may be assigned to a routine reference variable whose routine form is the same as that of Ref's parameter.

4.16 Conclusion

This chapter has presented detailed specifications of TOMO. The stage is now set for the merger of TOMO with the

AI features found in TELOS [LeBlanc 77], which is the subject of the next chapter.

Chapter 5

The AI Features: TOTEL := TOMO + TELOS

5.1 Introduction

This chapter describes how the base language developed earlier accommodates those language features for AI applications found in TELOS. The features of interest with respect to concurrency are complex control structures (including those involving demons), data contexts, and associative retrieval. Each of these features presents problems when combined with parallel processes. The following sections discuss these problems and propose solutions using the facilities in TOMO to implement the features. The resulting language is called TOTEL.

The discussion explores the implications of merging TOMO with TELOS instead of providing a complete definition of TOTEL. Features of TELOS not affected by concurrency are not discussed. We also omit a discussion of data abstraction mechanisms, which are provided by capsules in TELOS, since we have already seen in Chapter 3 how modules provide such facilities in a concurrent environment. We have replaced capsules by modules in TOTEL.

Before examining the merger of the features in detail, we mention some common problems. First, since TELOS was designed under the assumption of sequential execution, some of its features do not carry over directly to a parallel processing environment. Part of the problem of merging the AI features with TOMO arises from the need to reconceptualize these features for parallelism. A related problem is that TELOS is still an evolving language. Simpler extensions of PASCAL to accommodate the desired AI features are still being sought. The following sections will therefore attempt to deal with each feature in a manner independent from the way it currently exists in TELOS.

5.2 Complex Control Structures

We first expand on the ideas presented in Chapter 3 concerning the use of guards to control processes. Other TOTEL features such as data contexts and the data base require the knowledge of techniques developed here to explain their operations.

5.2.1 Guards as Control Abstraction Mechanisms

As we have seen in Chapter 3, a guard is a control abstraction mechanism that controls membership of crowds in order to coordinate processes. A process is suspended on a

queue when it requests but cannot be allowed to enter a crowd. The guard maintains such process queues to implement a strategy for coordinating processes to attack a problem. The guard, therefore, is a scheduler of processes. A process can call guard routines (either explicitly or implicitly when the process enters or exits a crowd) to interact with other processes under the guard's coordination. We shall refer to a coordination of processes as an inter-process control structure. For example, a simple producer-consumer situation is an inter-process control structure where the producer process, after it produces a value, calls a guard routine that will allow a consumer process to take the value. Several consumer processes may be competing to enter the crowd to take the value, and the guard routine determines which consumer process(es) succeeds. The choice of the consumer(s) may be very simple (particularly if there is only one competing consumer) or very complex, for example, involving associative retrieval of process queues.

An alternate view of the coordinating role of guards is to think of guarded modules as messengers that take messages from one process to another. The destination of messages is part of the strategy implemented by the guard. The use of messages to control processes is a special case of viewing general control structures as "patterns of passing messages", advocated by Hewitt [Hewitt 77]. Although Hewitt views

all control structures as implemented by message passing, we have limited this notion to those control structures defined over processes. Instead of providing a fixed set of inter-process control structures, we provide an abstraction mechanism to implement such control structures based on message passing using guarded modules as messengers.

5.2.2 Guards Versus Overseers

Although TELOS does not have facilities for parallel processes, it does allow coroutines (a degenerate case of parallel processes) coordinated by a special kind of procedure called an overseer, which has the power to create, invoke, and terminate coroutines. It also gains control when a coroutine it has invoked suspends execution. An overseer is a control abstraction mechanism that represents a strategy to accomplish a particular task involving the use of coroutines. The details of how the coroutines are coordinated to accomplish the task are hidden from the caller of the overseer.

Although TOTEL could incorporate an overseer-like entity, there are many reasons not to do so. First, little would be gained, since the guard of a module can accomplish many of the same things an overseer can. Second, it is not necessary that an overseer gain control when a process sus-

pends execution in TOTEL, since no transfer of control occurs during suspension. Execution of the program will continue as long as there are other processes still executing. (A major purpose for overseers in TELOS is to clearly define where control is transferred when a coroutine suspends. A major shortcoming of SIMULA 67 [Dahl et al. 70] is the lack of such a feature.) Third, distinguishing overseers from other kinds of procedures adds to the complexity of the language. A major criticism of TELOS has been the large number of procedure-like entities. Fourth, overseers are centralized facilities for coordinating processes. While such facilities pose no problems in a serial environment, they can cause contention problems in a parallel environment. In TOTEL, coordination of a set of processes can be distributed among several guarded modules rather than centralized to just one. A good example of distributed coordination is a pipeline where processes that represent the pipeline's stages are coordinated by guarded modules that interface the stages. In TELOS a single overseer would have coordinated and interfaced the stages. For all these reasons, overseers are not incorporated into TOTEL and instead, guards are used to coordinate processes.

The absence of overseers, however, means that any routine may create processes by performing a parallel remote procedure call. Any routine may also create new module in-

stances on which to invoke procedures in parallel. Although in TELOS creation and invocation of coroutines are restricted to overseers, such a restriction stems from the need to have a clearly defined point where control is transferred after a coroutine suspends. As stated above, such transfer of control does not occur when a TOTEL process suspends, and therefore, such a restriction is not necessary in TOTEL.

5.2.3 Demons and Events

AI applications often require demons: processes that are implicitly invoked when certain triggering conditions exist over the data. Ideally, a demon is triggered at the moment its triggering condition becomes true. Such an ideal demon, however, would require evaluation of the triggering condition after each computational step that could affect the value of the condition. Execution of a program with such demons would degenerate to evaluating triggering conditions most of the time. Most implementations of demons therefore evaluate triggering conditions only when the program performs certain actions, such as a procedure call or a modification to the associative data base. In most applications of demons the level of resolution provided by such actions is sufficient to adequately monitor the triggering conditions. A very common use of demons, for example, is to monitor data base activity so that global consistency is

maintained after a local change is made. Such demons need only monitor their triggering conditions after modification operations on the data base.

Demons can be implemented in TOTEL using guards. The guard routines are an ideal place for demons in TOTEL to monitor the activity of processes, since guards control all inter-process interactions. Such monitoring should provide enough resolution for most applications of demons. A TOTEL demon can be implemented as a suspended process waiting for a particular condition. Computations using demons are thus interpreted as a special kind of inter-process control structure implemented by guards whose scheduling decisions depend on the triggering conditions of the demons. Guards can also implement parallel production systems similar to Hearsay-II [Fennnel and Lesser 75], where the blackboard is implemented as one or more guarded modules. Whenever the blackboard is updated, the guard routines evaluate relevant preconditions of productions, which are suspended processes. The preconditions may also be evaluated by signalling suspended precondition-evaluator processes.

Guards also subsume many of the functions that events in TELOS provide. Events provide software interrupts in TELOS. When a TELOS statement raises an event, control is transferred to an event handler for that event. Each TELOS

statement may have event handlers for events raised as a result of that statement's execution (including those raised in routines called from the statement). A handler consists of a body of statements and may receive parameters from the point where the event was raised. A handler may also be associated with a group of statements enclosed in a block. Events are classified into three categories according to their intended use. Suspend events provide coroutine suspensions, signal events provide general communication (including demon triggering), and escape events provide exception handling. We have already seen the use of guards and queue variables to handle process suspensions and demons in TOTEL. A TOTEL process suspends itself by calling a guard routine that performs a Wait on a queue rather than by raising a suspend event handled by an overseer. Demons are triggered in TOTEL by guard routines that monitor inter-process activity rather than by event handlers for signal events. Guarded modules can also act as messengers or communication channels for general communication purposes. We therefore do not provide suspend and signal events in TOTEL.

Guards, however, cannot adequately take the place of escape events, since exception handling requires abnormal transfer of control out of the statement causing the exception. Escape events are therefore included in TOTEL. When

an escape event is raised, control is transferred to its handler and then to the statement following the handler (handlers are associated with a statement by specifying them immediately after the statement). An escape event handler may have an empty body; such an "empty" handler is used if only transfer of control is required. If the statement is a parallel remote procedure call, its event handlers execute serially even though events may be raised in parallel. Escape events raised in a parallel procedure cannot transfer control beyond the parallel call statement that called it. Control transfers to the statement following the parallel call after all calls have returned and invoked handlers attached to the parallel call have executed. If a parallel call contains an alongwith part (see Chapter 4), its event handlers must have empty bodies, since otherwise, a raised event would cause concurrent execution of a handler and the alongwith part that is unchecked by a guard. In such cases, more complex communication can be established by calling a "messenger" module routine before the escape event is raised.

5.2.4 Generators

As will be seen below, certain data context and associative data base operations are generators, a special form of coroutines that return values on each suspension. Such gen-

erators are treated as producer processes sending values to the generator's callers, which are consumer processes. The coordination of the generator and its caller takes place in a guarded module of type Channel (see Chapter 4), which also provides buffers for values that have been produced but not yet consumed. In addition, either the generator, its caller, or an external agent may signal termination through the Channel module. An external agent is a process that executes in parallel with the generator and rescues the caller from suspending indefinitely when no more values can be generated. One way to use generators in TOTEL follows this outline:

```
Buf.Init;
cobegin
  GenMod->.Gen (Buf)
alongwith
  while Buf.More (Val) do begin
    { use Val }
    ...
  end {while}
end {cobegin}
```

The generator GenMod->.Gen is called in parallel with the while loop that receives generated values through the Channel variable Buf. Generation and use of values can occur concurrently. The body of the while loop may optionally contain provisions for termination, such as:

```
if <termination condition> then begin
  Buf.Fin;
  <exit while loop>
end
```

Generators raise the issue of behavior with respect to side-effects produced by processes executing in parallel. A generator may be defined so that the values generated are those that existed at the time the generator was started. Such a snapshot generator is unaffected by side-effects that create new values after the generator has started. An alternative form of a generator is one that is sensitive to side-effects that produce new values. In a serial language this second kind of generator terminates when there are no further values to generate. Since creation and generation of values in a serial language are strictly ordered in time, termination under such a situation is quite suitable. In a parallel language, however, events that create and generate values are only partially ordered. Terminating when no values exist is not well-defined in the sense that such a situation may be transient. A process may at a later time create a new value that should be generated. For example, a generator may be used to associatively retrieve all objects in the data base matching a pattern. Other processes may store new objects matching the pattern into the data base while the generator is active. If no ordering exists over the executions of the processes doing the retrieval and the stores, terminating the retrieval generator when no objects matching the pattern exist in the data base is quite arbi-

trary. The set of objects retrieved from the data base depends on the relative speeds of the processes doing the retrieval and the stores. To remove such dependencies, a generator that is affected by side-effects for a parallel language should not terminate when it runs out of values to generate. We therefore propose the idea of a persistent generator: one that never terminates unless told explicitly, either by its caller or by some other external agent (to prevent the caller from suspending indefinitely). The generators used for data context and associative data base operations are of this persistent variety.

Snapshot generators are more difficult to provide for shared collections of objects such as data contexts and the data base. In an asynchronous environment, it is neither meaningful nor possible to get values of a collection of objects at one instant of time unless the collection is modified in serialized steps that leave the collection in well-defined states. The programmer decides what such well-defined states are, so it can not be determined *a priori* when snapshots may be taken. We therefore do not provide predefined snapshot generators for data contexts or the data base, but allow the user to define new generators in accordance with the intended well-defined states. Persistent generators may be used to implement the snapshot generators.

5.3 Data Contexts

Many recent AI languages provide a data context mechanism that provides tentative or alternative modifications of data objects. In TELOS contexts are organized into a tree structure; each path from the root represents a sequence of modifications done to context-relative data objects. A new context may be created as an offspring of an existing context by the CreateContext function. The offspring inherits the parent's values of context-relative data objects. Subsequent modifications of context-relative data objects in the offspring context do not affect the values that exist in the parent. A TELOS program always executes in some context and is free to switch to another context that it can reference through a context-reference variable. The switch is accomplished by the construct:

```
InContext C do <statement>
```

where <statement> and procedures called from within <statement> execute in the context referenced by the context-reference variable C. References to offspring contexts may be obtained through the generator SubContexts. Finally, a subtree of contexts may be removed from the context tree through the Release procedure. More details of

the TELOS data context mechanism may be found elsewhere [Honda et al. 77, LeBlanc 77].

In order to provide a similar data-context facility in a parallel processing environment, we must design operations on both the context tree and on context-relative objects so that their integrity is maintained when they are accessed by parallel processes. We will first deal with the context tree operations. The context tree appears to the user of TOTEL as a predefined module that exports the type ContextRef and is implicitly imported by all programmer-defined modules. Depending on the actual implementation of the context module, ContextRef may be a capability or an exported pointer to a guarded module. In either case, crowds for the various context-tree operations will be associated with each context node to impose constraints on concurrent execution of the operations. We define for each context node two crowds, ContextRead and ContextWrite. The execution of a context-tree operation automatically puts the executing process in one of these crowds. The rules for their membership are those for the readers-writers problem: ContextRead may have one or more members only if ContextWrite is empty, and ContextWrite may have just one member only if ContextRead is empty. The following are the definitions of the context-tree operations in TOTEL:

CreateContext : function (C : ContextRef) : ContextRef

A new context node is created as an offspring of the context referenced by C. The calling process is put in crowd ContextRead of the new context so that the new context cannot be released (see Release below) until Exit of ContextRead is explicitly called by the calling process. The process is also a member of C's ContextRead crowd during the execution of CreateContext. The values of context-relative objects in the new context are initially those that exist in C.

Release : procedure (C : ContextRef)

The subtree rooted by context C is erased from the context tree. The execution of Release will be delayed if the ContextRead or ContextWrite crowd of any of the contexts in the subtree is non-empty. The subtree appears to all other processes to have been erased at the start of execution of Release (after appropriate delays). After execution of Release has started, references to contexts in the subtree being released will result in an error. Release is therefore an indivisible operation.

InContext C do <statement>

The <statement> and any procedures called from within it execute in context C. The process executing <statement> is a member of the ContextRead crowd of C during <statement>'s execution. An attempt to release C as a result of executing <statement> is therefore an error.

SubContexts : procedure (C : ContextRef;
var Buf : ContextRefChannel)

SubContexts is a persistent generator that returns through Buf ContextRef values for all immediate offspring of context C. Buf is a variable of the predefined type ContextRefChannel, which is defined to be Channel of ContextRef (see Chapter 4). SubContexts feeds Buf with ContextRef values while the caller of SubContexts concurrently takes values from Buf. The caller is put in the ContextRead crowd of C while SubContexts is generating values. The caller also becomes a member of the ContextRead crowd for each context referenced by the ContextRef values returned. The caller can leave such crowds by calling the Exit procedures for these crowds.

Finalize : procedure (C1, C2 : ContextRef)

Finalize transfers all values of context-relative objects in C2 to C1. All offspring of C1 are then released. C1 must be an ancestor of C2. During execution of Finalize, the calling process is a member of C1's ContextWrite crowd. The execution of Finalize will be delayed if any descendants of C1 have non-empty ContextRead or ContextWrite crowds. The caller of Finalize is a member of the ContextRead crowds for all contexts on the path from C1 to C2 during the execution of Finalize. Like Release, all offspring of C1 appear to all other processes to have been released at the start of execution of Finalize (after appropriate delays).

As a convenience, it is not necessary to prefix an operation name with the name of the context tree module.

The context-tree operations are nearly identical to those found in TELOS. The presence of more than one process sharing the context tree may delay the execution of certain operations, but in many cases the effects of such delays are not logically visible. For example, CreateContext and SubContexts may be performed on the same context concurrently depending on how these operations have been implemented.

Since we do not assume any ordering of two parallel processes performing these operations, it makes no difference whether the two operations are performed concurrently or one precedes the other. Had a particular ordering of the operations been crucial to the correctness of the program, the two processes should have been serialized through a guarded module. For these reasons, no constraints on the concurrent execution of CreateContext and SubContexts are specified in the above definitions. In general, no constraints are specified on the concurrent execution of any combination of CreateContext, SubContexts, and InContext. Concurrent execution of these operations with Release or Finalize, however, is constrained so that contexts are not released or finalized while other operations on them are in progress.

We now move on to operations on context-relative objects. In TELOS there are two kinds of context-relative objects: <u>relative variables</u> and <u>data base objects</u>. Relative variables provide storage for values that are local to each context. In effect each context has a local copy of the relative variables, whose values are initially those of the corresponding relative variables in the parent context. Modifications to relative variables in a context do not affect the values of the corresponding relative variables in other contexts (even descendants). Because of their independence with respect to contexts, it is possible to permit

concurrent access to the same relative variables in different contexts. Relative variables in TOTEL as a subset of variables declared in the var part of modules.

Data base objects are objects in the associative data base (see below). Like relative variables, they have different values for different contexts. Unlike relative variables, modifications to a data base object propagate to all descendants of the context in which the modification occurs. We must therefore prohibit any read or write operations on a data base object in a context if the data base object is also undergoing a write operation in an ancestor context. This restriction can be easily enforced using the skeleton tree technique discussed elsewhere [Honda et al. 77, Wegbreit 76].

5.4 Associative Data Base

Many AI applications require associative retrieval of data objects. In TELOS the associative data base consists of data base objects (DBO), which are pointer closures of PASCAL heap objects. The pointer closure of an object is that object and the pointer closure of all other objects reachable through non-DBOP pointers from the object. A DBO may be referenced through a data base object pointer (DBOP), which may be part of a DBO so that DBOs can cross-reference

each other. A DBO may also contain routine reference variables as well as (in TOTEL) queue variables so that routines and suspended processes may be retrieved associatively.

The data base and the normal PASCAL stack and heap space (called the working space) are two distinct address spaces. In TELOS a working space object must be copied into the data base for it to be associatively retrievable. The Store function accomplishes this copying. A copy must be made to guarantee that only DBOPs reference DBOs, and that all DBOs are modified by special DBO operations that also update information required for associative retrieval. Store takes a pointer to a working-space object, copies its pointer closure into the data base, initializes information required for associative retrieval, and returns a DBOP to the newly created DBO. An inverse operation, called Fetch, takes a DBOP and copies the DBO it points to into working space and returns a pointer to the copy.

A DBOP may be dereferenced to read a component part of a DBO, but a DBOP must be passed to the Change or Replace procedure to modify a DBO. Change takes a DBO component and the new value it is to have and performs the change. It will update any information required for associative retrieval as a side-effect. Only non-structure-defining components may be modified by Change. In particular, pointers

(not DBOPs) in DBOs may not be changed by Change. If a structural change is desired, the DBO must be copied by Fetch into working space and the change made there. The original DBO may then be replaced with the modified object by the Replace procedure. Replace takes a DBOP and a pointer to a working space object and copies the pointer closure of the working space object in the place of the DBO referenced by the DBOP.

Finally, DBOPs to DBOs may be found associatively through the function Find or the generator FindEach. Find takes a pattern and returns a DBOP to a DBO that matches the pattern whereas FindEach takes a pattern and generates DBOPs that point to all DBOs matching the pattern. A pattern in TELOS is a pointer to a <u>parameter record</u> for a <u>match routine</u> matching a particular type. A parameter record consists of a pointer to a routine and its actual parameters. The match routine procedurally specifies the criteria that a DBO must meet in order for it to be retrieved. Match routines are restricted to match a particular type. A pattern in TELOS, therefore, can only match objects of its match routine's match type. The match routine is applied against every DBO in a <u>candidate set</u> specified by the <u>Index part</u> of the match routine. The Index part specifies values that must exist in a DBO before the match routine can be applied to it, narrowing the set of objects that the matchroutine must

test. Match routines are never explicitly called; when used for data base retrieval purposes, they are called implicitly during a Find or a FindEach. The basic steps of FindEach(Pat) are:

1) Build the candidate set with all DBOs of the appropriate type containing values required by the Index part of Pat's match routine.

2) For each object in the candidate set, apply Pat's match routine. If the match routine succeeds, return a DBOP to the object.

A dual operation, PatternGet, takes an object and returns DBOPs to all DBOs containing patterns (pointers to parameter records for match routines) that match the object. More details of associative retrieval in TELOS may be found elsewhere [LeBlanc 77].

When such an associative data base is to be shared by parallel processes, several problems arise. First, required values in a DBO in the candidate set could change before the match routine is applied to it. We could therefore retrieve objects that no longer match the Index part of the pattern. Application of the match routine will therefore require reading the object while preventing other processes from

modifying it. Similarly, if DBOPs are buffered by the generator FindEach after successful application of the match routine, it is possible for the DBO to have changed values before the DBOP to it is unbuffered. Again, we could retrieve objects that no longer match the pattern. Finally, once FindEach has returned a DBOP, it is possible for the DBO to be modified by another process so that it no longer matches the pattern. Such modification can take place between the time the DBOP is returned and the time the DBO is first accessed by FindEach's caller, again having the effect of retrieving objects not matching the pattern. FindEach must therefore return objects that are locked from change since the time the Index part was first tried on them, and the caller of FindEach must release the lock after it is through with the object. Similar steps must be taken for Find. As we will see, controlling membership in crowds for the object returned by FindEach or Find can accomplish this locking.

Another problem we must face is the contention that may result from all processes sharing a central data base facility. Although removal of such contention is an implementation problem rather than a language design issue, a good language design should, nevertheless, facilitate efficient implementation. One way to get around the contention problem is to partition the data base into independent parts so

that access is distributed. Although data bases in past AI languages have been global and monolithic, we shall later see that there are several advantages to having several independent data bases, each local to a module.

A third problem arises from the fact that patterns are in effect generalized names for objects. All unguarded objects in TOMO must be exclusively accessible from a guarded module or through capabilities. If we are allowed to access objects in the data base by pattern, how can we maintain control of aliasing in order to guarantee exclusive accessibility? One solution is to only allow guarded objects into the data base. A less restrictive solution is to use capabilities for DBOPs, allowing DBOs to be either guarded or unguarded types. The latter approach is taken in TOTEL.

A more difficult problem arises with respect to maintaining data base consistency. When a DBO is modified, other DBOs may also need to be modified to keep the data base consistent. Often in AI applications the routine initiating the modification does not know of other DBOs that must be modified. Past AI languages have relied on data base demons to maintain consistency by interrupting program execution when a DBO modification that requires other DBO modifications occurs. Unfortunately, such a solution is difficult to apply in a parallel processing environment. Suppose two

DBOs, X and Y, contain data that mutually depend on each other. If a change is made in one, a demon is activated to change the other to maintain consistency. If a process changes X while another process reads Y and then X, it is possible for the second process to read the old value of Y before the demon has a chance to update it, and then to read the new value of X. The second process would read inconsistent values of X and Y. Even worse, if one process changes X while a second process changes Y, the final values of X and Y would be inconsistent since the demon activated by the change on X would change Y, while the demon activated by the change on Y would change X. Clearly, exclusive access to both X and Y is required for a change on either of them. In AI applications, however, it is often not possible to define explicitly all DBOs that need to be locked together for exclusive access. As we will see, partitioning the data base decreases the severity of this problem.

Having seen the problems of providing an associative data base in a parallel environment, we now discuss how we cope with these problems in TOTEL. As alluded to earlier, TOTEL does not incorporate a global, monolithic data base. Instead, the data base facility is composed of independent data bases that are local to module instances. The data base facility is provided as a structuring method, similar

to PASCAL files and TOMO channels. We may define a data base type by specifying:

DB_Type = data base of <type ID>

Data base instances may be declared as variables in the var part of a module. Like channels and queues, a data base type appears to the programmer as a module type whose routines are the data base operations. Assignment of a data base variable assigns only a reference to the actual data base instance (like queues and channels). A DBOP type for DBOs in a data base of type DB_Type may be declared by specifying:

DBOP_Type = DB_Type.DBOP

where "DBOP" is a capability type exported from module DB_Type. Each data base module (type) therefore defines and exports the DBOP type for the DBO type it stores. A DBOP has crowds DBORead and DBOWrite, whose memberships are controlled by the data base module that defines and exports the DBOP type. The execution of a data base operation automatically puts the executing process in one of these crowds during the operation. Access to every DBO is therefore protected by at least the guard of the DBO-type's data base module. If the DBO is a guarded module, it will further be

protected by its own guard. The rules for membership in DBORead and DBOWrite are the following: 1) a process may be a member of DBORead for a DBO in context C if and only if no other process is a member of DBOWrite for that DBO in C or in any of its ancestors and 2) a process may be a member of DBOWRite for a DBO in context C if and only if no other process is a member of DBOWrite for that DBO in any ancestor of C, and no other process is a member of DBORead or DBOWrite for that DBO in C or in any of its descendants. A process is a member of the DBORead crowd of a DBOP while the DBOP is dereferenced by the process.

There are several reasons why the TOTEL data base facility consists of many local data bases rather than a global monolithic one. First, we believe that the trend in future AI programs will be to distribute knowledge over several "expert" processes or modules. Lenat's beings [Lenat 75], Hewitt's actors [Hewitt 73,77], and Fennel and Lesser's knowledge sources [Fennel and Lesser 75] are examples of such "expert" entities. Second, applications that require a global data base are not precluded since data base instances may be declared in the outermost module. By storing only "global" (universally needed) information in such global data bases, however, we may be able to keep them small and less frequently accessed. Furthermore, global data bases can hold directories to local data bases similar to a

tree-structured file system. Information in a local data base can therefore be found from other parts of the program. Third, we are able to ameliorate the contention problem since access by processes will be distributed across several independent data base instances. Fourth, since data base instances can be declared local to a module, it is possible to encapsulate a data base with routines for accessing it. We can therefore define abstract data types that are implemented using associative data base facilities. A global data base cannot be used to implement abstract data types since objects stored in them are accessible from outside the abstract data type (like global variables). Finally, the guard of a module can protect access to a data base instance in the same way it can protect access to other variables of the module. Even though DBOs are individually protected through DBOPs, which are capabilities, a collection of DBOs may be more easily protected by encapsulating a data base instance in a guarded module. Consistency within a data base instance is also easier to maintain since the guard of the encapsulating module can restrict access to the data base during DBO modifications. Demon processes (as described in 5.2.3) for maintaining data base consistency can also use the guard to monitor data base activity.

The definitions of the data base module operations for TOTEL follow. Assume the following declarations:

```
type
  T_DB = data base of T;
  T_Ptr = ->T;
  T_DBOP = T_DB.DBOP { DBOP to T for T_DB }
  T_Pat = pattern for T;
          {pointer to a parameter record
           for a match routine matching T}
```

Store : function (Obj : T_Ptr) : T_DBOP

A new DBO of type T is created in the current context and the pointer closure of Obj-> is copied into the new DBO. A DBOP pointing to the new DBO is returned as the value of Store, and the caller of Store is made a member of the DBORead crowd of the DBOP. Obj-> must be of a sharable type (see Chapter 3) since a call to Store is a remote call. The new DBO is not retrievable until execution of Store has completed. If Obj-> is a guarded module, Store requires execution of the ModuleCopy procedure (see Chapter 4) of Obj->.

Change : procedure (var Comp : <DBO Component>;
NewValue : CompType)

NewValue is assigned to a DBO component designated by Comp. The assignment only has effect in the current context and its descendants. If Comp is

part of an unguarded DBO, the calling process, during the execution of Change, is a member of the DBOWrite crowd of the DBOP used to designate Comp. Change on a component of a guarded DBO is protected by the DBO's guard rather than by its DBOP.

Replace : procedure (Ptr : T_DBOP; Obj : T_Ptr)

The pointer closure of Obj-> replaces Ptr-> in the current context. During the execution of Replace, the calling process is a member of the DBOWrite crowd of Ptr, and the calling process is made a member of the DBORead crowd of Ptr immediately before Replace returns.

Find : function (Ptn : T_Pat) : T_DBOP

Find returns a DBOP to a DBO matching Ptn in the data base. If no such DBO exists at the time Find is called, Nil is returned. The caller of Find is made a member of DBORead crowd for the DBOP returned (if not Nil).

FindEach : procedure (Ptn : T_Pat;
var Buf : T_DBOP_Channel)

FindEach is a persistent generator that returns through Buf DBOPs to all DBOs that match Ptn in the current context. The caller of FindEach is

made a member of the DBORead crowd of each DBOP returned. The type T_DBOP_Channel is defined to be Channel of T_DBOP (see Chapter 4).

```
PatternGet : procedure (var Obj : <any type>;
                        var DBOP_Buf : T_DBOP_Channel;
                        var Ptr_Buf : T_Ptr_Channel)
```

PatternGet is a persistent generator that searches the type T's data base for DBOs with top level records containing or pointing to a matchroutine parameter record that matches Obj. For each such DBO found, PatternGet returns through DBOP_Buf the DBOP to the DBO, and through Ptr_Buf a pointer to a working space copy made of the DBO by PatternGet. The caller of PatternGet is made a member of DBORead crowd of each DBOP returned.

```
Fetch : function (Obj : T_DBOP) : T_Ptr
```

Fetch creates a working space copy of the pointer closure of Obj-> and returns a pointer to the copy. During the execution of Fetch, the calling process is a member of the DBORead crowd of Obj.

```
Delete : procedure (Obj : T_DBOP)
```

Delete replaces the value of the DBO referenced by Obj in the current context by the existing value

of the DBO in the parent context. If no value for the DBO exists in the parent, or if no parent exists, the DBO is removed from the data base. During the execution of Delete, the calling process is a member of the DBOWrite crowd of Obj.

Processes that are left in the DBORead crowd for a DBOP by one of the above operations (Store, Replace, Find, FindEach, or PatternGet) may leave the DBORead crowd by explicitly calling the Exit procedure for that crowd.

5.5 Conclusion

This chapter has discussed issues of merging the AI-features of TELOS with the concurrency features of TOMO to produce the concurrent AI language TOTEL. We have seen how modules, guards, and capabilities can be used to provide control abstractions (including those for generators and demons), context trees, and data bases for associative retrieval. Some of the features of TELOS are modified in TOTEL because of constraints imposed by concurrency not present in TELOS. Other TELOS features are subsumed by TOMO features that provide the same functions for a concurrent environment.

Chapter 6

Major Results and Future Research

6.1 The Major Results

The previous chapters have presented various ideas, which we now summarize. The base language TOMO incorporates many existing and new concepts for parallel languages. The major component of TOMO, the module, extends the capabilities of monitors. Like serializers and Owicki's shared classes, modules are an improvement over monitors because they do not impose overly strict mutual exclusion rules. Mutual exclusion is only required in the guard of the module, which regulates processes entering crowds to access the data in the module. We have borrowed crowds from serializers, which use them to keep track of processes accessing shared data. TOMO also uses crowds to restrict access to shared data: only members of a module's crowd may access the module's data. Crowds have limited views, which restrict the set of operations a crowd member may perfom on the module data. Modules, therefore, are an improvement over serializers because they constrain all access to shared data to be from crowds. Modules are also an improvement over shared classes because they syntactically isolate variables and statements required for access control from the

rest of the module. Such variables and statements are encapsulated in the guard of each module.

Modules in TOMO, however, are more than mechanisms for controlling access to shared data. More generally, they are restricted referencing environments that limit the mutual interference of all TOMO processes. Unlike languages such as Concurrent PASCAL, TOMO has no process definitions; only module definitions exist at compile time. A TOMO program consists entirely of module definitions, similar to EUCLID and PLITS. These definitions specify the actions of processes that execute in instances of the modules. Simplicity and uniformity are achieved by basing all semantics of TOMO on modules.

Processes interact in TOMO through remote procedure calls and module guards, which manage process queues. The parallel remote procedure call specifies parallelism as one step of a serial algorithm, enhancing conceptual clarity. Processes started by a parallel call interact through guarded modules to form an inter-process control structure. Since each process may itself perform parallel calls, a running TOMO program resembles a hierarchy of Concurrent PASCAL programs. Techniques for implementing various inter-process control structures such as data-flow networks and pipelines were discussed. The serial remote procedure call takes the

place of messages in PLITS and other message-based languages, and provides a common form of message passing following the procedure call-return pattern. Remote procedure calls together with queues are used to implement other forms of message passing. Details of the implementation of message passing as well as inter-process control structures are isolated in the guard of a module. Guards are therefore a control abstraction mechanism.

The incrowd statement allows syntactic specification of code regions that always execute in a crowd, and therefore enables elimination of run-time checks for crowd membership when modules are accessed. Such regions are similar to conditional critical regions, except processes entering the regions (crowds) may be scheduled more explicitly by a guard procedure. The incrowd statement can also force an indivisible sequence of operations to be performed on the shared data (membership in a crowd may be maintained over several operations). Serializers lack this capability since they combine entering and leaving a crowd with performing an operation on the shared data. We have separated these steps to provide greater flexibility. Security is not sacrificed by such a separation, since crowd membership can be tested when an operation is performed.

We have also extended the ideas of Silberschatz, et al. on the use of capabilities with monitors. TOMO capabilities allow guarded access to otherwise unguarded components of a data structure. Components can be accessed without the knowledge of internal implementation details of the data structure, yet the access can still be regulated by the same guard that regulates operations on the entire data structure. The same guard is required for both component and structure operations, since these operations can mutually interfere with each other. Use of capabilities is consistent with the crowd concept for guarded modules: a process joins a crowd of a capability in order to access the object named by the capabilty. Management of capabilities is distributed over guards of modules whose components are accessed by the capabilities. No central facility for capability management is required.

In order to accompish secure sharing of general data structures, we impose restrictions on parameter passing for remote procedure calls so that all unguarded-type objects in TOMO are always exclusively accessible from a guarded module and therefore protected by unique guards (no two guards independently protect the same object). Passing pointers to unguarded types by revocation preserves the exclusive accessibilty of unguarded objects. The use of modules to implement data structures allows pointers to entire struc-

tures to be passed by revocation without destroying structural links. Pointers to modules in TOMO are intended to provide access and not to define structural properties.

Features for AI applications are taken from TELOS and incorporated into TOMO using TOMO constructs. Both the data context mechanism and the associative data base are predefined modules. The associative data base facility is composed of independent data base modules that can be declared locally to a module to reduce problems arising from concurrent access. Capabilities provide guarded access to a data base object or to a data context. Constraints on concurrent execution of context and data base operations are expressed as constraints on crowd membership. Demons and other complex control structures are implemented using guards. Guards may associatively retrieve queues of suspended processes to route messages and invoke processes by pattern. Because TOMO constructs are used to provide the special AI features, we avoid "cross-product" interactions between TOMO features for parallelism and the AI features.

To sum up, TOMO, together with its AI extensions, provides constructs for meeting the requirements for parallel AI processing outlined in Chapter 2. Flexible sharing of data is provided by guarded modules and capabilities. Guarded modules also allow implementation of complex control

structures and message passing. The "traditional" AI features are adapted for a concurrent environment and implemented by TOMO constructs.

6.2 Future Research

Many problems remain to be tackled. A major research problem is the implementation of TOMO on a suitable multiprocessor machine. One possiblity is to map a module instance to a physical processor. The module's private data can be stored in the private local memory of that processor. Remote procedure calls could be implemented with inter-processor message primitives. A non-private module (a module in which several processes may execute) could be implemented as a time-shared processor, or as a cluster of processors sharing a common local memory, such as in Cm* [Swan et al. 77]. Casey and Shelness have suggested a similar approach to structuring distributed processing systems [Casey and Shelness 78]. Since TOMO programs dynamically allocate modules, there is a problem of how processors are allocated dynamically to an executing program. It may be possible to determine from the static hierarchy of modules an efficient processor allocation strategy for a given physical processor topology. A study of what processor topologies more readily accommodate possible module topologies of TOMO programs would also be of interest.

A TOMO implementor might also exploit information associated with each crowd and queue to implement run-time checks for deadlocks. Both crowds and queues record the identity of processes in them, and this information (together with additional information relating queues to crowds) could be used by a deadlock detector process. Horn has suggested that such a process could run in parallel with a program [Horn 78] looking for deadlocks as the program executes (similar to an incremental garbage collector [Dijkstra et al. 75, Steele 75]). Such a detector could raise deadlock escape events, which can be handled by the program or the operating system (resulting in the program's termination). Other potentially costly run-time checks could also be implemented by parallel checking processes. For example, the steps necessary to start a remote procedure call may be executed in parallel with a check for crowd membership. If the process is not a member of the proper crowd, the call must be cancelled (which must include restoration of modified data values) and an escape event raised. Such methods of parallel checking exist at the hardware level in high-performance machines, such as the Amdahl 470 (where, for example, the validity of data in the cache is checked in parallel with access to main memory).

The TOMO langauge itself can be improved in many ways. One simple improvement is to extend the use of views and

crowds. Views are similar in intent to qualified types suggested by Jones and Liskov [Jones and Liskov 78], where a type is qualified by a specification of operations that are permissible on that type. Qualified types are useful in formal parameter specifications to restrict the set of operations permissible for a parameter. Feldman's PLITS [Feldman 76] can similarly constrain access to messages by allowing only a subset of message slots (fields of a PLITS message record) to be accessible from a PLITS module. A useful extension of TOMO would be to use views of crowds to similarly qualify formal parameter types. Processes executing a procedure having such parameters would be constrained to be members of the crowds whose views qualify the parameters.

Another improvement is type parameterization of modules. We deliberately ignored this aspect of modules since it is not of interest with respect to concurrency. Furthermore, a simple macro facility could provide some aspects of type parameterization without language modifications. Nevertheless, one simple method to achieve limited type-parameterized modules would be to allow only pointer types to parameterize module definitions. Within modules only operations common to pointer types would be allowed on variables of parameterized types. Only one set of module routines would therefore need to be compiled for each

parameterized module definition. Clusters in CLU [Liskov et al. 77] take a similar approach to type parameterization by making all variables references to objects.

A more important area needing improvement is that of queues for process suspensions. The difficulty with queues is that once a process suspends on one, that process must rely on another process to signal the queue to awaken it. A process must always check to see if suspension is safe (that is, whether another process will eventually awaken it) before actually suspending. Such a check cannot always be performed at the time of the suspension. An alternative to queues is busy waiting, where a process continuously checks a wait condition. Since the waiting process is never suspended, it can also check conditions to see if it should stop waiting and move on to some other computation. In a multiprogramming environment busy waiting wastes valuable processor time. In a multiprocessor environment, however, processor time per se is not the scarce resource. A compromise between busy waiting and queues is conditional waits, which are used in conjunction with monitors. When a process suspends with a conditional wait in a monitor, it specifies a condition (a Boolean expression) that must be true for the process to continue. Every time some other process exits the monitor, the conditions of all suspended processes in the monitor are evaluated. We could extend conditional

waits so that each suspended process could also determine at such times whether to continue waiting or proceed to some other activity. Conditional waits and busy waiting, however, do not allow explicit storage of suspended processes in data structures, which is possible with queues (by storing the queues themselves in the data structures). Such explicit storage of suspended processes is required, for example, for associative retrieval of suspended processes. The formulation of the "right" mechanism for process suspension is an area of research needing further investigation.

Another area needing improvement in TOMO is the verbosity of the programs. Modules require too many lines of specifications. In addition, specifications for one entity, such as a routine, are distributed throughout the module in places that are textually distant from each other. An exported module routine requires its routine form to be specified in one place, the fact that it is exported in another place, and its body in yet another place. The nesting of modules also separates module specification from implementation. (PASCAL procedure declarations also suffer in the same way from nesting.) Syntactic improvements to enhance readability are therefore needed in TOMO.

6.3 Conclusion

We have presented in the above chapters the design of a parallel programming language for AI applications. This chapter has summarized the major results of the design effort. Several areas of possible improvement have been suggested. Still to be tackled is the implementation of the language, which by itself is a major research problem. We have outlined one possible implementation approach.

Language design is an iterative process, where each iteration includes the language's specification, implementation experience, and actual use. The process requires constant feedback and criticism from the language's users and implementors as well as from other language designers. This dissertation is just one, though significant, step of this process. The important part of that step is not the actual language specifications but the ideas behind them. We have strived to explicitly state the ideas that influenced the design decisions. Many subtle but important points, however, remain implicit in the language specifications. Further work on the specifications should help to better understand the ideas and to further refine them.

Appendix

Example Program Segments

Program 3-1

```
{  A readers-writers solution using
   a monitor [Hoare 74, p.556]    }

var
  Buffer : T; { a buffer of some type T is to be shared }

class readers and writers:monitor;
  begin
    readercount : integer;
    busy : Boolean;
    OKtoread, OKtowrite : condition; { Queue in TOMO }
  procedure startread;
  begin
    if busy or OKtowrite.queue then OKtoread.wait;
    readercount := readercount + 1;
    OKtoread.signal;
    { Once a reader can start, they all can }
  end startread;
  procedure endread;
  begin
    readercount := readercount - 1;
    if readercount = 0 then OKtowrite.signal
  end endread;
  procedure startwrite;
  begin
    if readercount <> 0 or busy then OKtoread.wait;
    busy := true
  end startwrite;
  procedure endwrite;
  begin
    busy := false;
    if OKtoread.queue then OKtoread.signal
                      else OKtowrite.signal
  end endwrite;
  {init code}
  readercount := 0;
  busy := false
end readers and writers;
```

Program 3-2

```
ReaderWriter = module
{  Hoare's solution to the readers-writers problem [Hoa74]
   using modules  }
{  The type T is the type of the buffer being shared  }
  modspec
    type
      EnterExitForm : procedure;
      ReadForm : function : T;
      WriteForm : procedure (Item : T);
    guard
      Reader : crowd (Read)
      routine
        Enter, Exit : EnterExitForm;
      end;
      Writer : crowd (Write, Read)
      routine
        Enter, Exit : EnterExitForm;
      end
    end;
    routine
      Read : ReadForm;
      Write : WriteForm;
  end {modspec};

  var
    Buffer : T;
  guard
    var
      QR, QW : Queue;
    routine
      Reader.Enter : EnterExitForm;
      begin
        if not Writer.Empty or not QW.Empty then
          QR.Wait;
        QR.Signal
      end;

      Reader.Exit : EnterExitForm;
      begin
        { assured : Writer.Empty }
        if Reader.Empty then
          QW.Signal
      end;

      Writer.Enter : EnterExitForm;
      begin
        if not Writer.Empty or not Reader.Empty then
          QW.Wait
      end;
```



```
{ ReaderWriter (continued) }

      Writer.Exit : EnterExitForm;
      begin
        { assured : Writer.Empty and Reader.Empty }
        if QR.Empty then
          QW.Signal
        else
          QR.Signal
      end
    end {guard};
  routine
    Read : ReadForm;
    begin
      Read := Buffer
    end;

    Write : WriteForm (Item);
    begin
      Buffer := Item
    end
end {ReaderWriter}
```

Program 3-3

```
ReaderWriter = module
{  Readers-writers solution where writers
   have preemptive priority over readers }
{  Type T is the type of the buffer being shared  }
  modspec
    type
      ReadForm : function : T;
      WriteForm : procedure (Item : T);
      InitForm : procedure;

    guard  {empty}  end;

    routine
      Read : ReadForm;
      Write : WriteForm;
      Init : InitForm;
  end {modspec};
  var
    Buffer : T;
    WriteEntryCount, WriteExitCount : Integer;
  routine
    Init : InitForm;
    begin
      WriteEntryCount := 0;
      WriteExitCount := 0
    end;

    Read : ReadForm;
    var
      WExitCountSave : Integer;
    begin
      repeat
        WExitCountSave := WriteExitCount;
        Read := Buffer
      until WriteEntryCount = WExitCountSave
    end;

    Write : WriteForm (Item);
    var
      WExitCountSave : Integer;
    begin
      repeat
        WExitCountSave := WriteExitCount;
        WriteEntryCount := WriteEntryCount + 1;
        Buffer := Item;
        WriteExitCount := WriteExitCount + 1
      until WriteEntryCount - 1 = WExitCountSave
    end
end {ReaderWriter}
```


Program 3-4

```
SharedList = module
{  Implements a shared singly linked list
   whose elements can be accessed by capabilities  }
{  Assume type Attribute has been declared as a record
   having AttrFields as a component  }
  modspec
    type
      AttrCap = cap->Attribute;
      Func = function (ID : Symbol) : AttrCap;
      GuardProc = procedure;
      AttrGuardProc = procedure;
      InitProc : procedure;
    guard
      Inserter : crowd (Insert)
      routine
        Enter, Exit : GuardProc;
      end;
      AttrWriter : AttrCap.crowd (var AttrFields)
      routine
        Enter, Exit : AttrGuardProc;
      end;
      AttrReader : AttrCap.crowd (const AttrFields)
      routine
        Enter, Exit : AttrGuardProc;
      end;
    end;
    routine
      Insert, Find : Func;
      Init : InitProc;
 end {modspec};

 type
   Node = record
     Sym : Symbol;
     Atr : AttrCap;
     Link : NodeRef
   end;

   NodeRef = ->Node;

 var
   Head : NodeRef;

 guard
   var
     Writing : Boolean;
     NumReading : Integer;
     QI, QW, QR : Queue;
```

```
{ SharedList (continued) }

    routine
      Inserter.Enter : GuardProc;
      begin
        if not Inserter.Empty then
          QI.Wait
      end;

      Inserter.Exit : GuardProc;
      begin
        { assured : Inserter.Empty }
        QI.Signal
      end;

      AttrReader.Enter : AttrGuardProc;
      begin
        if Writing or not QW.Empty then
          QR.Wait;
        NumReading := NumReading + 1;
        QR.Signal
      end;

      AttrReader.Exit : AttrGuardProc;
      begin
        { assured : not Writing }
        NumReading := NumReading - 1;
        if NumReading = 0 then
          QW.Signal
      end;

      AttrWriter.Enter : AttrGuardProc;
      begin
        if Writing or NumReading > 0 then
          QW.Wait;
        Writing := True
      end;

      AttrWriter.Exit : AttrGuardProc;
      begin
        { assured : NumReading = 0 }
        Writing := False;
        if QR.Empty then QW.Signal
                    else QR.Signal
      end;

      InitGuard : procedure;
      begin
        Writing := False; NumReading := 0
      end;
    end {guard};
```



```
{ SharedList (continued) }

  routine
    Insert : Func (ID);
    var
      NewNode : NodeRef;
    begin
      New (NewNode);
      with NewNode-> do begin
        Sym := ID;
        New (Atr);
        Insert := Atr;
        Link := Head
      end;
      Head := NewNode
    end;

    Find : Func (ID);
    var
      NodePtr : NodeRef;
    begin
      NodePtr := Head;
      Find := Nil;
      while NodePtr <> Nil do with NodePtr-> do
        if Sym = ID then begin
          Find := Atr;
          NodePtr := Nil
        end else
          NodePtr := Link
    end;

    Init : InitProc;
    begin
      Head := Nil;
      InitGuard
    end;
end {SharedLIst};
```

Program 3-5

```
SymTab = module
{ Implements a bucket-hash symbol table using SharedList }
{  Assume type Symbol has been declared with a
   field Hash that contains its hash value  }
  modspec
    type
      SymAttr = SharedList.AttrCap;
      SymTabFunc = function (ID : Symbol) : SymAttr;
    guard {empty} end;
    routine
      Enter, Lookup : SymTabFunc;
  end {modspec};

  type
    HashRange = 0..HashLim;
    BucketHashTable = array [HashRange] of ->SharedList;

  var
    BHT : BucketHashTable;
```



```
{ SymTab (continued) }

  routine
    HashFun : function (ID : Symbol) : HashRange;
      begin
        HashFun := ID.Hash;
        if BHT [ID.Hash] = Nil then New (BHT [ID.Hash])
      end;

    Enter : SymTabFunc (ID);
    var
      Attr : SymAttr;
    begin
      with BHT [HashFun (ID)]-> do begin
        Attr := Find (ID);
        if Attr <> Nil then
          { duplicate declaration }
        else
          incrowd Inserter do Enter := Insert (ID)
    end end;

    Lookup : SymTabFunc (ID);
    var
      Attr : SymAttr;
    begin
      with BHT [HashFun (ID)]-> do begin
        Attr := Find (ID);
        if Attr = Nil then
          {undecled ID};
        Lookup := Attr
    end end;

     Init : procedure;
     var
       I : HashRange;
     begin
       for I := 0 to HashLim do
         New (BHT [I])
     end;
end {SymTab}
```

Program 3-6

```
NetSorter = module
{ Sorts N+1 integers using a data-flow network }
  modspec
    type
      SortProc = procedure (var A : IntArray);
      InitProc = procedure;
    guard  {empty}  end;
    routine
      Sort : SortProc;
      Init : InitProc;
  end {modspec};

  type
    IndexRange = 0..N; { N must be odd }

    ArrayElemPath = module
      modspec
        type
          CopyFunc : function : Integer;
          GuardProc=procedure (OperatorIndex : IndexRange);
          InitProc=procedure (I : IndexRange; V : Integer);
        guard
          Operate : crowd (var Val)
          routine
            Enter, Exit : GuardProc;
          end;
        end;
        routine
          InitGuard : procedure;
          CopOut : CopyFunc;
          Init : InitProc;
      end {modspec for ArrayElemPath};

      var
        Index : IndexRange;
        Val : Integer;
```



```
{ NetSorter (continued) }

      guard
        type
          States = (FirstLev, SecondLev);
        var
          State : States;
          SecondOpQ : Queue;
        routine
          Operate.Enter : GuardProc (OperatorIndex);
          begin
            if odd (OperatorIndex) then
              if State = FirstLev then
                SecondOpQ.Wait
          end;

          Operate.Exit : GuardProc (OperatorIndex);
          begin
            if odd (operatorIndex) then
              State :=.FirstLev
            else begin
              State := SecondLev;
              SecondOpQ.Signal
            end
          end;

          InitGuard : procedure;
          begin
            State := FirstLev
          end;
      end {guard};

      routine
        Init : InitProc(I, V);
        begin
          Index := I;
          Val := V;
          InitGuard
        end;
    end {ArrayElemPath module};

    Path = ->ArrayElemPath;
    PathArray = array [IndexRange] of Path;
```

```
{ NetSorter (continued) }

    TestAndSwap = module
      modspec
        type
          TSProc = procedure (const Paths : PathArray);
          InitProc = procedure (TSI : TSIndexRange);
        guard private end;
        routine
          TestSwap : TSProc;
          Init : InitProc;
      end {modspec};

      var
        TSIndex : IndexRange;

      routine
        TestSwap : TSProc (Paths);
        var
          Temp : Integer;
        begin
          incrowd Paths [TSIndex]->.Operate (TSIndex)
          using Path1 do
            incrowd Paths [TSIndex+1]->.Operate (TSIndex)
            using Path2 do
              if Path1.Val > Path2.Val then begin
                Temp := Path1.Val;
                Path1.Val := Path2.Val;
                Path2.Val := Temp
              end
        end;

      Init : InitProc (TSI);
      begin
        TSIndex := TSI
      end;
    end {TestAndSwap};

    Operator = ->TestAndSwap;

  var
    OperatorSet : set of Operator;
```



```
{ Net Sorter (continued) }

  routine
    Init : InitProc;
    var
      OpI : Operator;
      I : IndexRange;
    begin
      { Set up the operator nodes }
      for I := 0 to N - 1 do begin
        New (OpI, I); {extra parm is for OpI->.Init}
        OperatorSet := OperatorSet + [OpI]
      end
    end;

    Sort : SortProc (A);
    var
      I : IndexRange;
      Paths : PathArray;
      Oprtr : Operator;
    begin
      { Set up the paths of network }
      for I := 0 to N do
        New (Paths [I], I, A [I]);
        {extra parms are for Paths[I]->.Init}
      { activate network N div 2 + 1 times }
      for I := 0 to N div 2 do
        forall Oprtr in OperatorSet do
          Oprtr->.TestSwap (Paths)
      { copy results of sort back into A }
      for I := 0 to N do
        A [I] := Paths [I].CopOut
    end
end {NetSorter}
```

References

[Atkinson and Hewitt 77] Atkinson, R. and Hewitt, C. "Synchronization in Actor Systems", ACM SIGART-SIGPLAN Symposium on Principles of Programming Languages, January 1977, pp. 267-280.

[Baker and Hewitt 77] Baker, H. G. and Hewitt, C. "The Incremental Garbage Collection of Processes", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, Rochester, New York, Aug. 1977.

[Bobrow and Raphael 74] Bobrow, D. G. and Raphael, B. "New Programming Languages for Artificial Intelligence Research", Computing Surveys, Vol. 6, No. 3 (Sept. 1974), pp. 1-174.

[Bobrow and Winograd 76] Bobrow, D. G. and Winograd, T. "An Overview of KRL, a Knowledge Representation Language", Stanford University Tech. Report, Nov. 1976.

[Brinch Hansen 73] Brinch Hansen, P. "Concurrent Programming Concepts", Computing Surveys, Vol. 5, No. 4 (Dec. 1973), pp. 223-245.

[Brinch Hansen 75] Brinch Hansen, P. "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering Vol. SE-1, No. 2 (June 1975), pp. 199-207.

[Brinch Hansen 77] Brinch Hansen, P. "Distributed Processes: A Concurrent Programming Concept", Computer Sciences Department, University of Southern California, Sept. 1977.

[Casey and Shelness 77] Casey, L. and Shelness, N. "A Domain Structure for Distributed Computer Systems", Proceedings of the Sixth Symposium on Operating Systems Principles, Purdue University, Nov. 1977.

[Dahl et al. 70] Dahl, O. J., Myhrhaug, B. and Nygaard, K. "The SIMULA 67 Common Base Language", Norwegian Computing Center Publication S-22, Oslo, 1970.

[Davies 73] Davies, D. J. M. "Popler 1.5 Reference Manual", University of Edinburgh, TPU Report No. 1, May 1973.

[Dennis 73] Dennis, J. B. "First Version of a Data Flow Procedure Language", Computation Structures Group Memo 92, Laboratory for Computer Science, MIT, Cambridge, Mass., Nov. 1973.

[Despain and Patterson 77] Despain, A. M. and Patterson, D. A. "X-Tree: A Tree Structured Multi-Processor Computer Architecture", Computer Science Division, University of California, Berkeley, California, 1977.

[Dijkstra 68] Dijkstra, E. W. "Cooperating Sequential Processes", in Programming Languages, F. Genuys, (Ed.) Academic Press, New York, New York, 1968.

[Dijkstra 75] Dijkstra, E. W. "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", CACM, Vol. 18, No. 8 (Aug. 1975), pp. 453-457.

[Dijkstra et al. 75] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. "On-the-fly Garbage Collection: An Exercise in Cooperation", Dijkstra note EWD496, June 1975. (Also in Lecture Notes for Computer Science 46, Springer-Verlag, New York, 1976.)

[Duff 76] Duff, M. J. B. "CLIP 4, A Large Scale Integrated Circuit Array Processor", International Joint Conference on Pattern Recognition, 1976, pp. 728-733.

[Fahlman 75] Fahlman, S. E. "Symbol-mapping and Frames", SIGART Newsletter No. 53 (Aug. 1975), pp. 7-8.

[Feldman 76] Feldman, J. A. "A Programming Methodology for Distributed Computing (among other things)", TR9, Computer Sciences Department, University of Rochester, September 1976.

[Fennel and Lesser 75] Fennel, R. D. and Lesser, V. R. "Parallelism in AI Problem Solving: A Case Study of Hearsay II", Carnegie-Mellon University Technical Report, October 1975.

[Finkel and Solomon 77] Finkel, R. A. and Solomon, M. H. "Processor Interconnection Strategies", Computer Sciences Department Technical Report #301, University of Wisconsin, Madison, Wisc., July 1977.

[Friedman and Wise 78] Friedman, D. P. and Wise, D. S. "Applicative Multiprogramming", Computer Science Department Tech. Report No. 72, Indiana University, Bloomington, Indiana, January 1978.

[Habermann 75] Habermann, A. N. "Path Expressions", Carnegie-Mellon University Technical Report, June 1975.

[Hewitt 72] Hewitt, C. Description and Theoretical Analysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in a Robot. AI Memo No. 251, MIT Project MAC, April 1972.

[Hewitt 73] Hewitt, C. "A Universal ACTOR Formalism for Artificial Intelligence", Proceedings of the Third International Joint Conference on Artificial Intelligence, Aug. 1973, pp. 235-245.

[Hewitt 77] Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages", Artificial Intelligence, Vol. 8, No. 3 (June 1977), pp. 323-364.

[Hoare 73] Hoare, C. A. R. "Towards a Theory of Parallel Programming" in Operating Systems Techniques, C. A. R. Hoare and R. H. Perrot, (eds.), Academic Press, New York, New York, 1973.

[Hoare 74] Hoare, C. A. R. "Monitors: An Operating System Structuring Concept", CACM, Vol. 17, No. 10 (Oct. 1974), pp. 549-557.

[Hoare 77] Hoare, C. A. R. "Communicating Sequential Processes", Computer Science Department, The Queen's University, Belfast, Northern Ireland, March 1977.

[Honda et al. 77] Honda, M., Travis, L. E., LeBlanc, R. J., and Zeigler, S. F. "An Improved Data Context Mechanism", MACC Tech. Report No. 47, University of Wisconsin, Madison, Aug. 1977.

[Horn 78] Horn, Frank Hartmut, personal communication, Madison, Wisconsin, May 1978.

[Jensen and Wirth 76] Jensen, K. and Wirth, N. "PASCAL-User Manual and Report", in Lecture Notes in Computer Science, G. Goos and J. Hartmanis (Eds.), Springer-Verlag, Berlin, 1976.

[Jones et al. 77] Jones, A. K., Chansler, R. J., Durham, I., Feiler, P., and Schwans, K. "Software Management of Cm* -- A Distributed Multiprocessor", Proceedings of the National Computer Conference, Vol. 46, pp. 657-663, AFIPS Press, 1977.

[Jones and Liskov 78] Jones, A. K. and Liskov, B. H. "A Language Extension for Expressing Constraints on Data Access", CACM, Vol. 21, No. 5 (May 1978), pp. 358-367.

[Kaplan 73] Kaplan, R. M. "A Multi-Processing Approach to Natural Language" Proceedings of the 1973 National Computer Conference, AFIPS Press, 1973, pp. 435-440.

[Knuth 73] Knuth, D. E. The Art of Computer Programming, Vol. III: Searching and Sorting. Addison-Wesley, Reading, Mass., 1973.

[Lamport 77] Lamport, L. "Concurrent Reading and Writing", CACM, Vol. 20, No. 11 (Nov. 1977), pp. 806-811.

[Lampson et al. 77] Lampson, B. W., Horning, J. J., London, J. G., and Popek, G. L. "Report on the Programming Language Euclid", SIGPLAN Notices, Vol. 12, No. 2 (Feb. 1977).

[LeBlanc 77] LeBlanc, R. J. "Specification and Rationale of TELOS, a PASCAL-Based Artificial Intelligence Programming Language", Tech. Report No. 309, Computer Sciences Department, University of Wisconsin, Madison, Wisc., Dec. 1977.

[LeFaivre 74] LeFaivre, R. A. "Fuzzy Problem Solving", MACC Tech. Report No. 37, University of Wisconsin, Madison, Wisc., Sept. 1974.

[Lenat 75] Lenat, D. B. "Beings: Knowledge as Interacting Experts" Proceedings of the Fourth International Joint Conference on Artificial Intelligence, Sept. 1975, pp. 126-133.

[Lesser and Erman 77] Lesser, V. R. and Erman, L. D. "A Retrospective View of the Hearsay-II Architecture", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Mass., Aug. 1977, pp. 790-800.

[Liskov et al. 77] Liskov, B. H., Snyder, A., Atkinson, R. and Schaffert, C. "Abstraction Mechanisms in CLU", CACM, Vol. 20, No. 8 (Aug. 1977), pp. 564-576.

[McDermott and Sussman 72] McDermott, D. V. and Sussman, G. J. "The Conniver Reference Manual", AI Memo No. 259, MIT Project MAC, May 1972.

[McGraw and Andrews 77] McGraw, J. R. and Andrews, G. R. "Language Features for Process Interaction", Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March 1977, pp. 114-127.

[Newell 73] Newell, A. "Production Systems: Models of Control Structures." in Visual Information Processing, W. C. Chase (ed.), Academic Press, 1973, pp. 463-526.

[Noyce 77] Noyce, R. N. "Microelectronics", Scientific American, Vol. 237, No. 3 (Sept. 1977), pp. 62-69.

[Owicki 77] Owicki, S. "Verifying Concurrent Programs with Shared Data Classes", DSL-TR-147, Stanford University Digital Systems Lab, Stanford, Calif., Aug 1977.

[Owicki and Gries 76] Owicki, S. and Gries, D. "Verifying Properties of Parallel Programs: An Axiomatic Approach", CACM, Vol. 19, No. 5 (May 1976), pp. 279-285.

[Shaw et al. 77] Shaw, M., Wulf, W. A. , and London, R. L. "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators", CACM, Vol. 20, No. 8 (Aug. 1977), pp. 553-563.

[Silberschatz et al. 77] Silberschatz, A., Kieburtz, R. B. and Bernstein, A. J. "Extending Concurrent PASCAL to allow dynamic resource management", IEEE Transactions on Software Engineering", Vol. SE-3, No. 3 (May 1977), pp. 210-217.

[Solomon and Finkel 78] Solomon, M. H. and Finkel, R. A. "Roscoe: A Multi-Microcomputer Operating System", Computer Sciences Department Tech. Report #321, University of Wisconsin, Madison, Wisconsin, May 1978.

[Steele 75] Steele, G. L. Jr. "Multiprocessing Compactifying Garbage Collection", CACM, Vol. 18 (Sept. 1975), pp. 495-508.

[Swan et al. 77] Swan, R. J., Fuller, and S. H., Siewiorek, D. P. "Cm* -- A Modular Multi-processor", Proceedings of the National Computer Conference, Vol. 46, pp. 637-644, AFIPS Press, 1977.

[Travis et al. 77] Travis, L. E., Honda, M., LeBlanc, R. J. and Zeigler, S. F. "Design Rationale for TELOS, a PASCAL-Based AI Language", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, Rochester, New York, Aug. 1977.

[Uhr 75] Uhr, L. M. "A Wholistic Integrated Cognitive System (SEER-T1) That Interacts With its Environment Over Time", Computer Sciences Dept. Tech. Rep., University of Wisconsin, 1975.

[VanLehn 73] VanLehn, K. A. (Ed.) "SAIL User Manual", Stanford Artificial Intelligence Laboratory Memo AIM-204, July 1973.

[Van Wijngaarden et al. 76] Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., and Fisker, R. G. Revised Report on the Algorithmic Language ALGOL 68, Springer-Verlag, 1976.

[Wegbreit 76] Wegbreit, B. "Faster Retrieval from Context Trees", CACM, Vol. 19, No. 9 (Sept. 1976), pp. 526-529.

[Wirth 77] Wirth, N. "Modula: a Language for Modular Multiprogramming", Software -- Practice and Experience, Vol. 7 (1977), pp. 3-35.

[Wilber 76] Wilber, B. "A QLISP Reference Manual", SRI AI Note 118, 1976.

[Woods 70] Woods, W. "Transition Network Grammars for Natural Language Analysis" CACM, Vol. 13, No. 10 (Oct. 1970), pp. 591-606.