A Distributed Object-Oriented Graphical Programming System

John Evans

UUCS-90-017

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

September 26, 1990

Abstract

This report presents the design of a distributed parallel object system (DPOS) and its implementation using a graphical editing interface. DPOS brings together concepts of object-oriented programming and graphical programming with aspects of modern functional languages. Programs are defined as networks of active processes called "Process Objects" and interconnecting communications lines. These active objects are independent single threaded programs that employ much of the modularity, encapsulation of function, and encapsulation of data found in sequential object-oriented programming. The system defines a clear and simple approach to generating and managing parallelism and interprocess communication in a distributed parallel environment. DPOS contributes several new solutions to the problems of distributed parallel programming that are improvements over existing systems. The key improvements of this system include: a more complete and versatile means of dynamic process creation; the specification of complex network topologies in an intuitively clear and understandable way; seperation of the management of parallelism from the definition of computation; automatic resolution of low level critical section issues; the ability to design and develop separate processes as traditional single threaded programs; the encapsulation and incremental development of programs subnetworks; application of graphical programming concepts to high level programming.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

The need for computers that solve larger and more complex problems has motivated a great deal of recent research. The technology to produce larger and faster computers at reasonable cost has advanced rapidly. A significant direction that these technological developments have taken is in the area of distributed parallel computer systems.

Within the research area of distributed parallel computer systems the technology to develop software has not kept up with the advances in hardware development [27]. There are, however, recent software developments that hold great promise for distributed parallel software systems. Also, due to the impact of distributed parallel computer architectures on algorithm design, there is reason to believe that the characteristics of distributed parallel algorithms may be reasonably grouped for study. This research is directed specifically at distributed parallel computer systems which are refered to simply as distributed.

Object-oriented programming has developed over the years in sequential programming as a means of encapsulating functionality and data in the development of computer programs. It has proven to be an excellent approach for managing large scale and complex programs. It has also been used for modeling various types of real world systems that are complex and concurrent in nature such as factory systems, human cognitive systems, communication networks and social systems. The encapsulation of functionality and data is a requirement of distributed computers and

the applicability of object-oriented programming for modeling concurrent systems make it a natural choice for extension to distributed systems.

Visual programming systems are a relatively new and controversial area of software design. The potential for the representation of complex networks in visual programming systems, however, make them especially suited for application to distributed programming systems.

Other relatively recent developments in software design also have utility in the framework of distributed computing. Among these are particular functional programming concepts such as first class status for functions and procedures and, most recently, the development of 'lazy' functional languages.

1.2 Purpose

This report presents the design of a distributed parallel object system (DPOS) and its implementation using a graphical editing interface, together with three surveys of background information. These include a survey of parallel algorithm characteristics, a survey of existing graphical programming technology, a discussion of sequential object-oriented characteristics and a survey of distributed object languages and systems. Finally, a programming methodology is presented and used in conjunction with the object system to develop and implement an example program.

1.2.1 Surveys of Background Information

The surveys describe fundamental characteristics of existing systems. In addition they discuss implications for new developments, including the distributed object system (DPOS). This document contains background surveys on:

1. Distributed parallel algorithm characteristics. This covers characteristics of distributed algorithms that are thought to be fundamental to many parallel algorithms and discusses reasons for believing that the algorithm characteristics are representative of parallel programming in general.

2. Distributed parallel object-oriented languages and systems. This survey discusses the characteristics of sequential object-oriented programming and compares and contrasts them with the needs of distributed object-oriented programming. The survey also discusses the qualities of a number of existing parallel object languages and systems.

3. Visual programming systems. This survey describes a number of existing visual programming technologies and graphical schemes used by computer scientists, discusses assets and liabilities of visual programming in general, and considers the application of visual programming technology to distributed programming.

1.2.2 Distributed Parallel Object System

The second goal of this project is to study the design and implementation of a distributed parallel object system (DPOS). The system adapts the concepts of sequential object-oriented programming and existing models of distributed programming and it innovatively addresses the needs of the distributed algorithm characteristics studied. Specifically the system supports:

1. The use of algorithm characteristics that are germain to distributed programming.

2. The management of concurrency issues foreign to sequential programming. In particular it should support high-level distributed programming with a minimum of understanding of low-level parallel programming.

3. Development of programs that are highly modular and that can be effectively designed as reuseable program components.

4. Development of programs employing the complex topologies typical of distributed algorithms.

5. The effective documentation of programs and the integration of documentation into the programming process.

The DPOS implementation generates and uses Butterfly Scheme source language code on a BBN Butterfly parallel processor [26].

1.2.3 Visual Parallel Programming

The third goal of this project is the application of visual programming to the needs of DPOS. This includes:

1. The design of the object system itself. The system is designed to allow topological constructions of process networks in a way not easily manageable in a textual language system. The process networks are, however, easily and effectively manageable in a graphical programming environment.

2. Adaptation of an existing visual diagram system "Viper"[29] to the needs of DPOS.

3. The implementation of a post processor for the visual diagram system that will interpret the diagrams correctly, attribute semantic meaning to the diagrams, and generate appropriate output in the target language.

1.3 Overview of this document

The following is a list of the remaining parts of this report:

Chapter 2 discusses the characteristics of distributed algorithms.

Chapter 3 compares and contrasts sequential object-oriented programming with the needs of distributed object-oriented programming.

Chapter 4 describes previous advances in visual programming systems and discusses the application of visual programming technology to distributed programming.

Chapter 5 describes the design of DPOS.

Chapter 6 describes the application of a graphical programming interface to DPOS.

Chapter 7 compares the system with sequential and parallel closure style object implementations. A programming methodology is also presented and used to develop an example program.

Chapter 8 discusses contributions and conclusions reached from this project.

CHAPTER 2

DISTRIBUTED PARALLEL ALGORITHMS

2.1 Introduction

There are several significant differences between distributed parallel programming and traditional sequential programming. Distributed parallel programming differs from sequential programming in that there is no global environment for variables, hence no global data. There are multiple threads of activity (processes) which may be simultaneously active on different processors or simply interleaved if on the same processor. It also differs in that it requires a semantics for communication between the processors (message passing). This semantics controls synchronization between the processes and is often unlike that in sequential programming. Finally, it also differs in that there is a significantly greater cost for communication between processors than for memory access on a uniprocessor. These differences affect the way programmers develop programs, the kinds of algorithms used, the portability of programs and the design of parallel programming systems[18,6].

The architecture of parallel computers has a greater impact on the characteristics of programs than does the architecture of sequential computers. This impact affects not only the performance of the programs, but their structure as well and in some cases dictates features of programming languages. As an example, the language OCCAM[23], a descendent of CSP[13], was designed to work harmoniously with the architecture of the transputer chip and is the primary programming language

for transputer networks. Distributed parallel computing systems have a greater effect on program design than shared memory systems because of the necessary encapsulation of program and data units on different processors and also because of the potential for process networks of virtually unlimited size. In addition to the hardware, the programming language and system of a parallel computer affects the development of software. The programming language and operating system used on a parallel computer must reflect the demands and capabilities of the computer architecture and also the needs of the programmer.

2.2 Parallel Algorithm Characteristics

Independent of specific computer architectures and languages, several broad classifications of parallel algorithms have been studied that employ specific strategies for the creation and management of parallelism. Attempts at classification of sequential algorithms have had only limited success at encapsulating the broad range of algorithm characteristics. There is reason to believe, however, that the classificiation of parallel algorithm characteristics may be more generally representative of parallel algorithms. This is because of the common requirements for the creation and management of parallelism, and the encapsulation requirements of distributed memory parallel processors. The reconciliation of algorithm types with parallel architectures is an important consideration in the development of distributed parallel programming systems, and one that has only had limited success in the research to date.

2.3 Parallel Algorithms

Among those algorithm types studied are the "Compute, Aggregate, Broadcast," pipeline (and systolic) and "Divide and Conquer" strategies[22,9]. Each has specific implications for the design of distributed parallel programming environments.

The "Compute, Aggregate, Broadcast" algorithm type is characterized by periodic cycling between sequential global decision making and distributed parallelism.

Typically, some controlling process broadcasts messages to a group of server processes that carry out tasks in parallel and periodically send their results to the controller (aggregation). It then decides what to do next and rebroadcasts to the servers the information necessary for the next cycle. The synchronization in this process may not be global to the whole program since it may only be localized to a part of the program. It also may not be complete if only partial results are needed to make the continuation decision. Almost all parallel algorithms have some attributes of this algorithm type, since they involve some kind of root decision making process that initiates the parallelism and then collects the results.

One implication of this algorithm type is the need for a "one to many" and "many to one" communication protocol. Few parallel programming systems support broadcasting or aggregation explicitly. Notable examples are "Broadcasting Sequential Processes" (BSP)[11] and Petri Nets[8]. The Petri net model is a primitive programming model, not generally useful for modeling broadcasting in a high-level language. The BSP model is more like high-level programming language models, similar to CSP with broadcasting as the basic communication protocol. Petri Nets also support aggregation of a sort, but BSP does not. While broadcasting and aggregation are generally useful concepts in parallel programming, they are not without drawbacks. The major drawback of this type of communication is that it requires synchronization between groups of communicators thereby reducing parallel performance. Also, the overhead for communication is at best log(n), (when using a binary tree network structure) which can dominate the computation performance in fine-grain algorithms.

Pipeline and systolic algorithms are the most heavily researched type of parallel algorithms. Their distinguishing characteristic is one-directional, one-way data flow. Systolic algorithms are a special case of pipeline algorithms. The distinction between systolic and pipeline algorithms is that true systolic algorithms are fine

grained, synchronized computations with only local connections and regular process network structure.

In the simplest case, a pipeline algorithm takes advantage of the lengthy and repeated calculations present in a sequential algorithm. The sequence of calculations is broken into chunks which are each handled by a separate process. Sequences of data transmissions keep the pipeline full. The latency of starting and terminating the pipe is O(length of pipe). Pipeline process structures do not have to be linear and can follow any kind of network pattern, including cycles. It is important for the maintenence of parallelism that the stages in the pipeline have similar granularity so that processes are not kept waiting; thus causing bubbles in the pipe. Buffering messages and asynchronous message passing can be useful in addition to replication of stages in the pipe to smooth out the flow and maintain parallelism. These algorithms are also notable in that the process network is often similar in structure to traditional data structures and abstract data types like trees, streams or arrays.

Divide and Conquer parallel algorithms are different from sequential divide and conquer algorithms. Sequential divide and conquer algorithms involve the solution of a large problem by combining the solutions of similar independent smaller problems. Parallel divide and conquer algorithms are characterized by multiple similar processes where each solves a similar problem, possibly including communication and cooperation between the processes. The process network here is freeform and may involve one-way or two-way communication between processes depending on the system implementation and program design. The topology of this algorithm type is a general graph. Two-way communication presents a special problem in a parallel environment. The sending process is required either to block immediately and wait for a response, or else define some sort of critical section semantics (as with futures [26]). Blocking is effectively synchronization and critical sections are an extra burden for programmers.

2.4 Conclusion

Types of programming algorithms as well as computer architectures have significant implications for the design of distributed programming systems and languages. These implications include:

1. Data. The data space must be partitioned between processes. Global data are impractical. A modular programming approach is implied by the architecture as well as algorithm types.

2. Communication. Message passing paradigms must be included to facilitate communication between processes. These paradigms fall into several general types. Broadcast/aggregate, two-way (function call) and one-way (pipeline). Few if any existing parallel programming systems support all three. The integration of these paradigms with the modes of communication (such as guarded message passing) is also a problem that has not been adequately addressed by existing parallel programming systems.

3. Multiple Threads. Programs are partitioned into multiple processes. The processes require different kinds and amounts of communication. Processes may perform distinctly different functions or may perform identical tasks depending on the algorithm type.

4. Similarities to Data Structures. Parallel algorithm structures are similar in construction to data structures. This implies that similar constructs and concepts may be used in composing parallel programs similar to those used to define abstract data types.

CHAPTER 3

DISTRIBUTED OBJECT-ORIENTED PROGRAMMING

3.1 Introduction

The characteristics of object-oriented programming make it particularly suitable for a distributed parallel programming environment. In particular, object-oriented programming involves separate computational objects with their own encapsulated data and functions. This level of encapsulation is not only desirable but necessary for efficient distributed parallel computing. The way in which objects communicate via message passing is similar enough to the requirements of distributed processing that it can be easily adapted. Object-oriented programming, because of its high degree of modularity and encapsulation, is well-suited to the building of large and complex systems. Its principals are even being applied to the design of large hardware and software systems. In the sequential programming world, object-oriented programming techniques are often used to simulate parallelism.

3.2 Parallel vs. Sequential Programming

3.2.1 Communication

Object-oriented concurrent programming differs significantly from object-oriented sequential programming in several ways. In the sequential version, message passing always means transfer of control. Since there is only one thread of control, the initiator is active before communication, inactive during the interaction and active again after it is over. The responder is inactive before, active during and inactive

after communication. So even if the data flow is one directional the interaction is always two directional because the thread must return before the initiator may proceed. In distributed parallel programming this restriction is not present or desirable. Although programs that follow the convention can be used, they do not perform well because the suspension of the initiator process forces a synchronization between the two processes; they behave as a single process even if on different processors.

In the general case of distributed parallel programming both parties may be active before, during and after the communication, so there is no notion of initiator and responder. There is the concept of information flow between processes. The freedom to have processes remain active at all times allows much more complex, data-flow based, object relationships. A great deal of research has been devoted to the handling of the problem of data flow between computational objects in sequential programming via data stream programming and lazy evaluation. These concepts, because of their close relation to parallelism and message passing, are very useful in conceptualizing about distributed parallel programming. There is also a great deal of research into data flow algorithms in parallel processing (i.e., systolic array and pipeline algorithms), although not necessarily in connection with object-oriented programming.

3.2.2 Inheritance

A second major difference between sequential and parallel object-oriented programming involves the concept of inheritance. In sequential programming inheritance means sharing of behaviors and data associated with an object. In distributed parallel programming the sharing of data is a major problem, because of the difficulties associated with remote access of data. Controlling the sequence and atomic access of data structures and implementing shared data concepts requires that accesses to the data elements be handled as remote accesses and critical

sections. Both of these are undesirable because of the added burden they place on the programmer and their execution time-cost. Moreover, while the semantic meaning of inheritance is generally difficult to analyze for sequential programs, it is even more difficult for parallel programs. These semantic difficulties[5] have prompted many designers of object-oriented parallel languages to omit inheritance properties altogether. Inheritance is also unreasonable in an environment where objects do not usually return to a top-level dispatch function every time an incoming message is received, so it is unclear at what level the inherited behaviors would be implemented.

3.2.3 Active Objects

A third major difference between sequential and parallel object-oriented programming is that since parallel objects remain active between incoming messages they do not need to save state information. This means that they are essentially independent functional programs. Information that would be saved in state variables in a sequential object may be passed on as parameter values in a recursive parallel process. This allows the desirable properties of functional programming (i.e., less specification of sequence of evaluation, ease of analysis) to be better maintained within the framework of individual objects. The possibility of non-terminating activity also allows objects to be developed as perpetual processes similar to viewing streams as perpetual processes or infinite data structures in sequential programming. This brings with it the same potential benefits that exist in sequential programming; the programmer need not provide termination criteria to all processes. Termination criteria may be provided only at control points in the data flow network. Complex interactions of processes do not have to be explicitly provided by the programmer. They can be easily implemented because sequencing information for individual computations is implicit in the communication network as propagation in the data flow, and does not have to be provided explicitly by the

programmer.

3.3 Existing Systems

Several object-oriented parallel programming systems and languages have been developed during the last fifteen years to address the problems of distributed parallel programming. A partial list of these includes ABCL/1, ACTORS, ADA, CCS, concurrent smalltalk, CSP and OCCAM [27,17,20,2,14,16,31]. Figure 3.1 compares various aspects of communication and process structure of these languages. Object-oriented parallel programming is defined here as a programming and design methodology (style) in which the system to be constructed is modeled as a collection of concurrently executable program modules, called objects, that interact with one another by sending messages [32]. All of these languages fall into this category. CSP is the oldest and with the exception of Concurrent Smalltalk, Petri Nets and CCS they are all extensions of the concepts of the original CSP as presented in 1978. CSP and CCS are often not considered to be programming languages but only language fragments or denotational languages. The only languages that are used commercially are ADA and OCCAM.

Figure 3.1 is a table of object-oriented parallel programming systems and language features employed by them. The following is a summary of the language features presented in Figure 3.1.

1. 1 or 2 way communication - does the system allow one way communication only or is there two way communication.
2. multicasting - does the system support multicasting or broadcasting
3. queues - does the system provide for buffered message sending as well as unbuffered
4. blocking out - is output required to block waiting for a receiver
5. blocking in - is input required to block waiting for a sender
6. trigger - can messages be constructed using more than two participating processes

	CSP	ADA	CCS	S.T.	OCC	ABCL	ACT1	Petri Nets
Communication Mechanism								
2 way	no	yes	no	yes	no	yes	yes	no
multicasting	no	no	no	no	no	no	no	yes
queues	no	no	no	no	no	no	no	yes
blocking out	no	yes	yes	yes	yes	y/n	yes	y/n
blocking in	no	no	yes	yes	yes	no	no	yes
trigger	no	no	no	no	no	no	no	yes
guarded in	yes	yes	yes	no	yes	yes	no	no
guarded out	yes	no	yes	no	no	no	no	no
multiple send.	no	yes	yes	yes	yes	yes	yes	yes
multiple rec.	no	no	yes	no	no	no	no	no
channels	yes	no	yes	no	yes	no	no	no
Processes								
synchronized	yes	yes	yes	yes	yes	yes	yes	no
proc. topology	stat.	child	n/a	dyn.	dyn.	child	child	stat.

Figure 3.1. Table of Language Features

7. guarded in - are input guards supported

8. guarded out - are output guards supported

9. multiple send - is there the possibility for nondeterministic receipt of a communication by multiple processes

10. multiple rec. - is there the possibility for nondeterministic receipt of a communication from multiple processes

11. channels - is communication direct between two processes or via a named intermediary (a channel)

12. synchronized - is message passing tightly synchronized

13. process topology - is the topology static or dynamic. Some systems allow only a tree structured topology (i.e. new children of existing parents may be created)

From Figure 3.1 it can be seen that several features are standard through most of the languages while others are quite rare. One way communication is standard, two way communication either implies synchronization or creates semantic problems and is less commonly employed. Guarded communication is message passing where messages are screened for content (i.e., message type) before the transaction is accomplished. This allows a receiver to receive only a selected type of message. Guarded input is common but guarded output is rare except in denotational languages. It is a desirable feature but generally more difficult to implement. Multicasting is a process by which a single message is sent simultaneously to a number of recipients. The only model discussed in this chapter that supports multicasting is the Petri Net model, but multicasting is supported on other systems and is known to be very useful in "Compute, Aggregate, Broadcast" algorithms. Allowing nondeterministic multiple senders or receivers is crucial for effective dynamic process creation and also for process replication. This is because it eliminates the requirement that processes in a process network be notified when a new link is established. Instead of notification, a new process can simply use the existing links. Multiple senders and receivers are rarely fully supported with the result that in most systems only parent/child relationships between dynamically created processes are possible. Without this, it is generally awkward to dynamically create a process that can communicate with two or more existing processes. The existence of named channels facilitates the implementation of both multiple senders and receivers and allows for completely dynamic process allocation. OCCAM succeeds only partially at this by having hardware channels between processors and allowing named channels on a single processor. Most of the systems require process synchronization at the time of communication; this is because, with the exception of Petri Nets, they do not support any message buffering.

3.4 Programming Extensions

3.4.1 Processes as First Class Objects

Several concepts of modern functional languages can be applied to distributed parallel programming systems. Among these is the ability to treat functions or objects as first class citizens in the language. Applying this to distributed object-oriented programming implies that processes, objects and communication connections might be composable as structures. This forms a direct relationship between the process network structure and data structures within the parallel programming language allowing the versatility and familiarity of abstract data types in defining process networks. This is an intuitive way of dealing with the network structure since many parallel algorithms view process networks as having a regular structure.

3.4.2 Lazy Evaluation

A second quality of modern functional languages is lazy evaluation. Lazy evaluation is the ability to specify arguments and a function to be applied without actually applying it, leaving a 'delayed' function call that can be 'forced' later upon demand. This is used in sequential programming to define infinite data structures and infinite sequential computations and is used as the basis of stream computation[1]. The concept of stream computation is directly applicable to distributed parallel processing and is the underlying model for pipeline processes and data flow networks. However, in parallel processing the stream does not need to be evaluated lazily but can be controlled by blocking of eager stream processes upon message sending. A second use of lazy evaluation is the composition of processes in a network to define virtual (lazily evaluated) process networks. The use of lazy evaluation allows programs to control the extent of computation from a single point that demands only the computations it needs rather than to have all parts of the program know in advance how much computation to do. This simplifies program control flow considerably by eliminating unnecessary and premature decisions in

low-level functions. In a parallel process network this allows the definition and analysis of the virtual process network topologically, separate from consideration of the program control flow.

CHAPTER 4

VISUAL PROGRAMMING

4.1 Introduction

Historically, visual programming environments[28,24] have been used primarily in the sequential programming world and specifically in the areas of describing data, information about data, software design, visualization of program execution[19] and documentation. For data representation this has taken the form of information tables and menus, array or tree diagrams, list structure diagrams, object inheritance trees and others. Visual software design has involved flowcharts, pretty printing programs, Petri Nets, state diagrams, icon systems, and data flow diagrams. Visualization of program execution has involved information about data or control flow, representations such as flow charts, along with animation and snapshots of program state.

4.2 Visual Programming Tools

The various means of visually representing programming have individual strengths and weaknesses. Each generally focuses on one particular aspect of programming to the exclusion of others:

1. Flowcharts[10] illustrate fine-grain program control flow. They do not contain information about data representation or capture large-grain program structure effectively.

2. Petri Nets and related token systems show fine-grain network control flow. They are the only commonly used systems that address parallel activities such as critical sections and computational dependencies, but they are also typically only fine-grained and contain no information about data (except for tokens).

3. Data flow diagrams are generally larger grained diagrams that show data transfers and a level of program modularity and are useful in object-oriented types of systems[25]. They do not generally contain information about data representation or fine-grained control flow.

4. State diagrams[15] are an abstraction of both data and control flow that is effective for representing input parsing, data-driven systems and output generation where input or output controls the program. They are less effective in other applications where this kind of tight data/control relationship does not exist. Even when they are useful they may not accurately reflect the actual program structure but only an abstraction of it. Also, while they do show data and control flow relationships, they show very little about the actual data structure or program control structures.

5. Programming icon systems[12] use icons to represent logical or control relationships and often contain aspects of flowcharts and state diagrams. They are essentially the descendents of these other approaches and suffer from similar limitations. However, they often make incremental improvements in certain areas such as the representation of recursion, the animation of program execution, or the increased ability to represent program abstraction.

6. Data diagrams and list diagrams effectively show information about abstract data structures. Both do this without showing anything at all about program structure or control flow.

7. Menus and tables show information at a user level that is highly readable but not really representative of the internals of the programming. These may be associated with interaction methods, to implement a kind of graphical interaction reminiscent of object-oriented programming that is highly abstracted from the actual program structure.

4.3 Problems and Benefits

A primary use of traditional visual programming systems has been in teaching beginning programming concepts. Visual systems such as flow charts or icon systems are very easily understood by beginners as well as experienced programmers. Relationships in control flow are more visually evident in the diagrams as compared to a textual system. The process is made less abstract by the two-dimensional images of the program elements and added visual clues (such as icon shapes or color). They aid the programmer in assembling the program from a set of concrete parts. The reduction of abstraction and the addition of visual clues while making the program easily readable are not perceived by experienced programmers as beneficial. One reason for this is that the experienced programmer does not need added visual information to understand the program relationships and control flow. Programmers typically perceive the extra information as visual clutter reducing the amount of information that can be manipulated on the screen. A second reason is that abstraction, while compounding an already difficult situation for the beginner, is an asset for the experienced programmer to whom anything that reduces his/her ability to abstract program elements is actually a limitation on effectiveness. Newer visual computing environments, such as window systems at the operating system level, are not perceived by experienced programmers as a limitation because they afford interaction at a very high-level and assist the programmer in more traditional low-level activities.

A benefit of data diagrams is that they afford experienced, inexperienced and even non-programmers with views of complex data abstractions that are not directly evident in the program. For example, a tree structure can be represented as a list in a program text, but is not easily readable, and a list with a loop in it is not even easily representable (and is very difficult to understand) in text. A problem with this kind of visual program information is that it is usually distinct from the program and through the course of program development may become obsolete. This makes matters worse by providing a clear but incorrect view of the program data.

Data flow diagrams and Petri Nets have the same advantage that complex data diagrams possess. The program control structures represented are nonlinear, and may contain intersections and cycles that are not easily represented or evident in program text, but are easily seen in a diagram.

A general problem of visual programming systems is that there are traditional existing systems that accomplish the same thing with greater flexibility, more succinctly or in a way that is already clearly understood by the programmer. For instance arithmetic or the syntax of an IF-THEN-ELSE statement is very clearly stated in textual notation. Graphical programming systems that try to make a new symbolic system to accomplish the same thing invariably fail either to be as succinct or as clearly understood.

4.4 Visual Programming Goals

Visual programming systems offer real benefits to experienced programmers only where they can be used more effectively than existing systems. Thus those properties which allow topological information to be incorporated into the programming environment that cannot be accomodated in a purely textual environment are an important attribute for a useful visual programming system.

One basic quality of visual programming systems is the topology of the two-dimensional surface. Text is inherently one-dimensional and takes on limited two-dimensional qualities (via indentation) when printed on a page. Graphical methods can make full use of the two-dimensional surface and by extension (such as layering) take on limited three-dimensional qualities. Qualities of programs that are multi-dimensional such as network relationships, complex data or dynamic relationships offer the greatest opportunity for visual programming. This is born out by the fact that the primary visual systems regularly employed by experienced programmers are data diagrams, flow diagrams, Petri Nets and state diagrams all of which represent relationships that are multidimensional and often at a high level of abstraction (at least relative to flowcharts), whereas arithmetic and low-level program text are almost exclusively manipulated and presented as text.

Visual artifacts of the graphical system, such as icons, convey topological information beyond the concepts they represent. For example, relative size may convey a sense of relative importance. The introduction of new visual information is appropriate only where this facilitates increased programmer effectiveness and contributes meaningful information to the process.

Visual systems should preserve or extend the desirable qualities of textual systems such as ease of abstraction, encapsulation, and ease of manipulation. Abstraction and encapsulation may be accomplished through the topological qualities of the visual system such as blocks or nesting of icons.

People understand and retain visual information more readily then textual information. Visual programming can excel at providing easily readable information, multiple views of different aspects of a program or different views of the same program. This is accomplished by providing a graphical framework for the organization of the programming process, and by merging the documentation of the program with the program development.

4.5 Parallel Visual Programming

The extension of visual programming techniques to parallel programming has not been widely developed; however, its usefulness has been obvious to parallel language and system designers. Petri Nets are most often described visually and visual diagrams are often used to explain process networks in object-oriented concurrent programming research. There are several aspects of distributed programming that have nonlinear topology, are difficult to explain textually (in a linear system) and are very easy to explain diagrammatically. These include:

1. The encapsulation of data and computation within processes.

2. The connectivity within the process network itself.

3. The existence of multiple simultaneous activities.

A single line of text can describe the computation of a sequential program at any given point in time and the relationships between functions in the evaluation tree of a program can be described textually, though many more complex relationships may exist that are not easily seen in the text. In a distributed parallel network, however, there exist multiple simultaneous relationships and activities that are effectively beyond the limits of textual systems.

Distributed programming is inherently more complex than single threaded programming. This fact makes the opportunity to combine documentation and program development a potential asset in a parallel programming environment.

The topology of visual programming provides the potential for the effective topological analysis of programs by viewing them as sets and graphs. They can be extended to visually analyze dynamic aspects of parallel programming such as data flow, critical path, critical section, process load distribution and deadlock situations. All of these are generally described graphically but not at present directly integrated into program development.

CHAPTER 5

DISTRIBUTED PARALLEL OBJECT SYSTEM (DPOS)

5.1 Introduction

This section describes a Distributed Parallel Object System (DPOS) that innovatively addresses several concerns of distributed parallel programming. A DPOS program is called a 'virtual process network'. A communicating process network model is used. A DPOS program is defined as a network of active processes called Process Objects (POs) and communication lines called Channels that are grouped into subnetworks called Network Modules (NMs).

Channels are predefined types in the system and are used for communication. Data are written to and read from channels by the Process Objects. The synchronization between Process Objects that access channels is handled automatically by the Channel.

Process Objects are single threaded program functions with calling parameters identical to traditional program functions. The connection links (Channels) between Process Objects appear as functions passed in as calling parameters. The control flow of Process Objects is internal. The progress of computation, however, may be controlled by regulating message traffic into and out of the Process Object.

Network Modules (NMs) are subnetworks composed of Channels, POs, and other NMs. The NMs have locally defined variables that contain data, Channel instances, NM instances or PO instances. These local variables constitute the local scope of

the NM. The topology of a Virtual Process Network may not be flat. Subnetworks having local scope may be nested, recursive and mutually recursive.

The DPOS environment stratifies a parallel program into two layers; object layer and system layer. The object layer is composed of individual discrete process objects (PO's). The process objects are defined using a text editor. The system layer contains the NM definitions and specifies instances of POs, Channels and NMs. The system layer encapsulates parallel programming issues such as multiple threads of control, process distribution, limited data sharing and interprocess communication protocols. Network Module and Process Object definitions are the user-defined types of DPOS and networks are made of instances of these types. The program network is defined topologically as a network diagram. The semantics of DPOS at the system level is declarative rather than procedural or functional. The definition of the network is analogous to the definition of abstract data types such as trees and graphs in high-level programming languages.

The implementation of DPOS discussed here uses a visual editing system to generate NM definitions. Channel type definitions are predefined. Process Object definitions are created by the user with a text editor. Butterfly Scheme is the target language and all examples are written using this language.

5.2 Classes, Instances and Inheritance

Traditional object-oriented systems are often categorized in terms of the system attributes of classes, instances and inheritance. The DPOS system is discussed in terms of the concerns of parallel programming rather than in traditional object-oriented terms and priority has been given to parallel programming issues over object-oriented programming issues in the system design. In DPOS the functionality of objects and classes has been extended to encapsulate parallel programming semantics as well as the functionality of object-oriented programming. The definitions of process objects and network modules in DPOS correspond to

class definitions in traditional object-oriented programming. The identifiers for process objects and network modules specify both a class name and an instance name that is unique within the local scope. Inheritance of data slot types and functionality in DPOS is accomplished in terms of the NMs and POs encapsulated within each NM class definition. If an NM definition includes other NM and PO definitions then it inherits the behavior and data of the encapsulated instances. Sharing of data slots is often possible in traditional object-oriented programming systems. Because of the inherent synchronization problems of shared variables in distributed programming systems, it is not supported in DPOS. Instead, DPOS incorporates 'channel' definitions. Channel definitions allow common access to data values and sharing of data values between process objects only in a framework of strict synchronization control as data messages. In traditional object-oriented programming systems, methods encapsulate functionality and structure interaction with objects. In DPOS, channel communication structures interact with process objects at specified points in computation. The relationship between methods and channels is discussed under programming methodology and development in Chapter 8.

5.3 The Producers and Consumers Problem

The Producers and Consumers example in Figures 5.1 and 5.2 shows the basic elements of the virtual process network as graphically defined network modules and process objects. The 'prod' and 'con' POs are contained within network modules 'producers' and 'consumers' respectively (see Figure 5.1). The parameters to the 'producers' NM are shown in Figure 5.3. 'Producers' and 'consumers' NMs are contained within the virtual process network 'producers-consumers'. The text output corresponding to the process objects is also shown in Figure 5.4. This example uses an asynchronous one-way communication channel between the producers and consumers. The network modules are nested such that the 'prod'

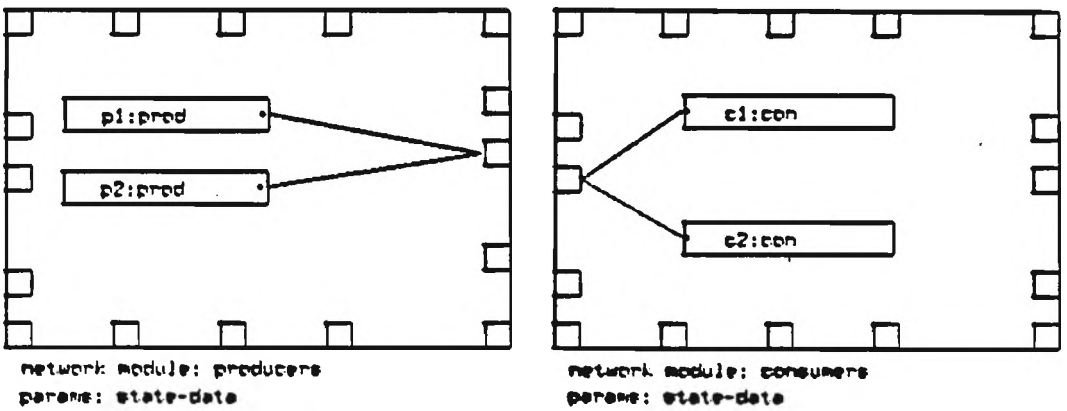


Figure 5.1. Producers and Consumers Network Modules

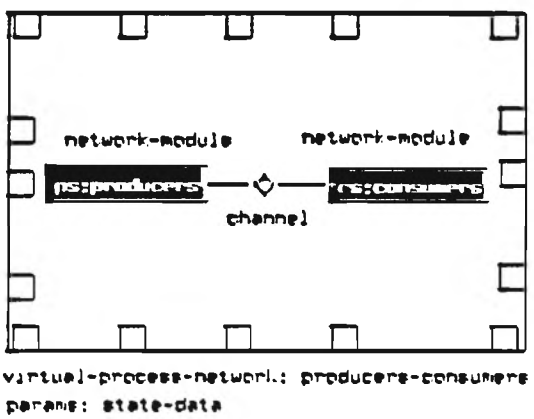


Figure 5.2. Producers and Consumers Virtual Process Network

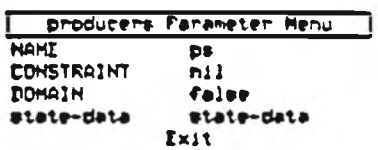


Figure 5.3. Producers Network Module Parameter Menu

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Sample Program:  PRODUCERS AND CONSUMERS
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  PROCESS OBJECT DEFINTIIONS FOR PROD AND CON
;;  PROCESS OBJECTS
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; The producer 'prod' type:
;; parameters:    channel - a communication channel
;;                state-data - numeric data
;; function:  recurse until not-quitting-time returns nil
;;            sending results of (product state-data) to channel

(define (prod channel state-data)
   (if (not-quitting-time state-data)
     (begin (channel 'send (product state-data))
                (prod channel (update state-data)))))

;; The consumer 'con' type:
;; parameters:    inchannel - a communication channel
;;                state-data - numeric data
;; function:  recurse until no more input data arrives
;;            reading input from inchannel and updating state-data

(define (con inchannel state-data)
  (let ((new-data (inchannel 'recieve))
      (print (new-results state-data new-data))
      (con inchannel (update-results state-data new-data)))

;undefined functions here are:
;  produce-update-state - returns a pair of (product new-state)
;  not-quitting-time
;  update-results
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Figure 5.4. Prod and Con Process Object Definitions

and 'con' POs are within network modules 'producers' and 'consumers' respectively. 'Producers' and 'consumers' are nested within 'producers-consumers'.

In addition several advanced aspects of this example will be explained in following sections. These aspects include:

1. The asynchronous channel is defined within 'producers and consumers network module' and is a parameter channel to both 'producers' and 'consumers'.

2. The channel is shared nondeterministically by both producers and consumers. This use of channels is equivalent to nondeterministic merge and join. Since the granularity of the use of this channel is small it is reasonable to share it in this way.

3. The control flow of the consumers is shown without any termination condition. The control of these processes is indirect via the receipt of messages from the producer processes. (see Section 5.9)

5.4 The Object Layer (Process Objects)

Process Objects in this system are defined as independent processes or programs in the parallel environment. A Process Object (PO) is simply an active program in the programming environment. The definition of a PO is a class definition in the traditional object-oriented sense and may be instantiated many times within a process network. In the underlying model there is a single thread of control and program scope associated with each process object. It may contain procedures and global data (global only to the Process Object itself) and may be programmed in any style that the programmer wishes, including traditional, sequential, object-oriented programming. A PO may follow a bounded sequential computation or may be an unbounded cyclical computation (like an operating system process) that is I/O driven via its communication channels (see Section 5.9.)

Providing that the underlying language allows it, process objects may employ multiple threads of control and share data with other process objects in ways other than those defined within DPOS. This, however, violates the intent of the system which is to encapsulate all of the parallel issues at the system level. Programs which only use DPOS and the sequential features of the underlying language are referred to as *consistent*. All POs discussed in this report are *consistent*.

Multiple POs are simply multiple separate programs. They may or may not reside on the same physical processor. Unless the programmer explicitly uses an *inconsistent* PO it will be a a single threaded program on a single processor. Multiple POs may be allocated to the same processor without conflict. The text for individual POs is defined by the programmer outside of the DPOS system and is integrated into a DPOS program via the visual editing system. In DPOS, POs do not have direct knowledge of each other and may be treated as completely isolated units. Constraints between POs are handled at the system level as arguments to the PO definitions or at the object level as messages between POs.

Communication between POs is done via channels which encapsulate the semantics of the communication protocol being used. Different channels use different semantics whenever appropriate.

The instantiation of Process Objects may be delayed or constrained in exactly the same manner as Network Modules. Delays and constraints are discussed in Section 5.8.

5.5 Channels

5.5.1 Channel Concepts

Communication between POs is done via channels. Channels are independent entities within the system and are the only means of communication between POs. Channels are defined at the system level and are made accessible to the object layer as parameters to top-level invocations of POs. Channels may be viewed either as

shared data objects with strictly synchronized access for use by multiple POs or as conduits between Process Objects with synchronization protocols. From either view the effect of channels is to provide communication between Process Objects via shared access to computed values. A channel in its simplest form is similar to a pipe in UNIX or the channels defined in CSP. In both of these systems a channel is a simple communication link shared by two processes.

A DPOS channel has several additional attributes that differ from other parallel programming systems. Unlike the other object-oriented parallel programming systems discussed previously, DPOS channels completely encapsulate process communication protocols. This means that neither definition of the PO nor the language of the PO necessarily needs to incorporate the semantics of parallelism and interprocess communication. In other systems the semantics of interprocess communication is incorporated into the programming language itself. Not incorporating communication semantics into the programming language allows the programmer several options not available in existing systems. It allows the target language level implementation of Process Objects to be very similar to individual programs in a sequential programming environment. Access to channels can be viewed at this level as syntactically and semantically similar to external I/O stream access in sequential programming. The programmer using this system can program PO definitions entirely using the standard elements of sequential Scheme and the DPOS system without using any Butterfly Scheme extensions. Encapsulated communication protocols also allow the system to include a variety of protocols in much the same way as high-level, sequential, programming languages allow a variety of control flow constructs. This means that the program at the system level may take advantage of the communication protocol that is most applicable to a given program.

Within the PO, channels are treated as first-class Scheme procedure objects and thus can be assigned to variables, passed in parameter lists, and applied to

arguments. A channel in its simplest form appears in the PO somewhat like a traditional sequential programming object. A PO interacts with a channel by applying the channel to a list of arguments. For example, given a channel 'ch' a PO might read a message from the channel into a variable 'result' using the expression:

```
(set! result (ch 'receive))
```

OCCAM and the 1985 version of CSP also allow named channels. However, they must be treated as literals within the program so they are not composable and in OCCAM all channels accesssed from a process must exist on the same processor node. Channels in these systems do not encapsulate communication semantics but only specify a relationship between the channel users. The semantics of communication is a part of the base language and only a single communication protocol is provided.

A PO is completely encapsulated by its communication channels. Since all communication between POs is done via channels, POs are defined without any direct knowledge about each other. This also allows a higher degree of modularity than is possible within traditional object systems. A subnetwork of a parallel program may be completely isolated from the rest of the network by enclosing it in a contour that cuts through all of the channels that communicate with it. This facilitates reasoning about subparts of programs and incremental modular development. The subnetwork may be developed, tested, and debugged by defining it and the surrounding channels and interacting with it by communication through the channels.

Individual PO definitions may be debugged and tested on a uniprocessor in traditional Scheme by replacing the encapsulating channels with user interaction functions functions written in standard scheme. This is possible because POs are

single threaded, completely encapsulated by channels, and channel interaction is similar to stream input and output.

Dynamic process creation in the existing distributed programming systems discussed in this report requires the propagation of information about newly created processes to any existing processes that require access to them. This propagation may require substantial programmer effort. It may also require a substantial amount of interprocess communication. In DPOS, PO and NM instantiations may be delayed. This means that the NM or PO is not instantiated when the parent environment is instantiated, but is delayed until a triggering event takes place during program execution (See section 5.6.2). In DPOS dynamic process creation, when a delayed NM or PO is instantiated, it uses existing channels to communicate. These channels are already known by the existing POs and NMs and act as 'stubs' in the virtual process network. In this way, delayed instantiation may be used to dynamically create NMs without requiring additional propagation of information or additional interprocess communication.

Multiple POs may simultaneously compete to 'send to' or 'receive from' a channel. Competition for any channel interaction is resolved nondeterministically. An example use of this feature is the replication of identical POs for identical tasks in a pipeline. A pipeline link with above average task granularity may be replicated allowing multiple instances of the link to execute simultaneously and balance the load across the link. Nondeterministic branch or merge operations can be implemented very simply using a rudimentary channel with multiple senders competing to send or multiple receivers competing to receive.

5.5.2 Channel Types

Several communication protocols are supported in DPOS. The semantics of these protocols support one and two-way communication, guarded communication and buffered communication in the process network.

1. **Asynchronous 1 way.** This is really a single message buffer. The receiver blocks and waits if no message is in the channel. The sender may proceed after a message is received by the channel (but not necessarily the receiver process). If there is already a message in the channel then the sender will block until it is removed by some receiver. The 'read' message type will return a copy of the message without removing the message from the channel.

```
messages:  output  (ch 'send message)
           input   (ch 'receive)
           read    (ch 'read)
```

2. **Buffered Asynchronous 1 way.** Similar to Asynchronous 1 way but with a bounded buffer size greater than 1. This channel type requires that a buffer size be specified when it is defined.

```
messages:  output  (ch 'send message)
           input   (ch 'receive)
```

3. **Synchronous exchange.** Two parties are involved in the exchange. Both parties play the role of both sender and receiver. Message types 'a-trans' and 'b-trans' are defined. Whenever both a 'b-trans' and an 'a-trans' message are being sent to the same channel the transaction or 'rendezvous' takes place and messages are exchanged. If only one message type is being sent, the sender blocks until a message of the other type arrives. This channel type is an extension of those used in CSP and OCCAM. In these systems, however, the communication protocol is a part of the language itself and the flow of data is one way. This channel type may be used to effect CSP-like communication if one of the parties does not make use of the information it receives and the other sends no useful information.

```
messages:    (ch 'a-trans message)
             (ch 'b-trans message)
```

4. **Synchronous, 1-way guarded input (blocking output) with optional default continuation.** This channel type implements guarded input. The channel is defined with a list of potential message types. Output processes specify message type and message. Inputting processes specify a list of acceptable message types. Input and output processes block until a match is found between a message type of an output process and an entry in the input process guard list. A single input process may interact with a channel at a time and will block other input processes until its input request is satisfied. If a default continuation is specified then the input process is nonblocking. This means that if no output message is waiting that matches one of the guard list types, the input process continues without blocking. The input process receives a pair which consists of the message type and message or simply 'continue in the default continuation case.

```
messages:  output  (ch 'writeguard type message)
           input   (ch 'readguarded typelist)
```

   Typelist is a list of types, where a type is a simple integer. Definition of this channel type requires a list of types.

5. **Synchronous, 1-way guarded output (blocking input) with optional default continuation.** This channel type implements guarded output. The channel is defined with a list of potential message types. Inputting processes specify message type. Output processes specify a guard list of acceptable message types and list of corresponding messages. Output and input processes block until a match is found between a message type in the input process

and the output process guard list. If a default continuation is specified the output process is not blocked. This means that if no input message is waiting that matches one of the guard list types the output process continues without blocking and no message is sent. The output process receives a return value that consists of the message type or 'continue in the default continuation case.

```
messages:  output  (ch 'writeguarded typelist messagelist)
           input   (ch 'readguarded type)
```

Typelist is a list of types, where a type is a simple integer. Definition of this channel type requires a list of types.

6. **1-way / 2-way interface** This allows an interface between one-way and two-way message traffic. The channel is interfaced as a one way channel by one PO and as a two-way channel by another. From the two-way side two-way communication is carried out as a single step. Parameters are passed in and a result is received. The process on this side initiates the transaction. The two-way side is blocking. The process executing a two-way message is blocked until a return value is received. From the one-way side communication is carried out as two separate steps as in the 'asynchronous 1 way' channel. First a receive and then a send through the same channel.

```
messages:  2 way side  (ch 'two-way message)
           1 way side  (ch 'receive)
                       (ch 'send message)
```

5.6 Additional Attributes of Channels

In addition to having a 'type' as described above, channels may have several other attributes. The additional attributes do not modify the communication properties of the channel types as described above. The additional attributes are used in the definition and dynamic instantiation of the Virtual Process Network.

5.6.1 Parameter Channels

The formal parameters of Network Modules correspond to data values and to channel instances (see Section 5.7). A channel instance used as an actual parameter to a nested Network Module instance is referred to as a parameter channel within the scope of the NM instance. It is not referred to as a parameter channel outside of the Network Module instance unless it also corresponds to an actual parameter to the outlying NM. Thus the term parameter channel is used for any channel instance visible within a NM but defined at an outlying scope.

5.6.2 Delay Channels

The instantiation of NMs may be delayed (see Section 5.8). If a PO or NM instance is delayed then one of the channel instances used as an actual argument to the PO or NM must be a 'delay' channel. The delayed PO or NM is not instantiated until the first message is sent to the 'delay' channel from some NM or PO (See Subsection 5.9.2). The 'delay' attribute may only be specified in the definition of the NM that defines the channel instance. Thus, a parameter channel may not be used as 'delay' channel.

5.6.3 Streams of Channels

Multiple channels may be defined simultaneously as a stream of channels. A channel stream is similar to a Scheme object stream where the components of the stream are channels. Streams of channels are defined within the Network Module definition and may be of any channel type. Channels that are elements of a stream of channels may not have the 'delay' attribute. Access to elements of a stream of channels may be made within the process object or within the network module environment. Elements of a stream of channels or the entire stream may be used as actual parameters to NM and PO instances. Streams of channels are accessed via the functions 'head' and 'tail' if provided within the Scheme implementation or with the functions 'h-strm' and 't-strm' provided by the DPOS system. For

example a send through the second element of a stream of asynchronous channels would be:

```
((h-strm (t-strm channel-strm)) 'send message)
```

5.7 Virtual Process Network (VPN)

A DPOS program is a virtual process network composed of network modules, process objects and channels. Briefly, process objects (POs) are individual processes, channels are communication links between process objects, and network modules (NMs) are components for composing subnetworks of process objects channels and other Network Modules. A program is run by invoking a virtual process network as you would invoke a Scheme function call. The process network is virtual in that components of the network may not initially exist (and they may never exist during the execution of the program) but are instantiated as the program progresses. The process network includes the definition and connectivity of all potential POs in the program as well as all potential channels and NMs. The invocation of a Virtual Process Network is similar to a function call and may include input parameters which are used as constants in the scope of the Virtual Process Network. Invoking a Virtual Process Network does not block the process that invokes it and also does not return a meaningful value. It should be viewed as a side effecting event.

The Process Objects within a Virtual Process Network generally have no direct knowledge of the VPN or of any of the network modules, channels, or other process objects contained within it. The exceptions to this are the parameter channels to the PO and any values passed into the PO from the Network Module in which it is created.

The virtual process network has several properties germane to high-level programming languages. These include declarative definition of the network, recursive

network definitions, lazy instantiation of the network and constrained instantiation of the network.

5.8 Network Modules

Network Modules (NMs) are subnetworks used for defining a Virtual Process Network. Network Modules are composed of Process Objects, Channels and other Network Modules. Network Module definitions have formal parameters that corresponding to data values and channel instances. The distinction between an NM and a Virtual Process Network is that the NM has parameters that correspond to channels (parameter channels) for input and output and a VPN has no parameter channels. NMs are not visible to POs. Network Modules have local environment. This allows local variables and recursive NM definitions. Invoking an NM requires actual parameter arguments to be provided. The arguments are the access channels that connect to the NM as well as any values computed within the scope of the invoking environment. Network Modules may be nested and recursively or mutually recursively defined. In the traditional object-oriented framework a network module definition constitutes a class definition and may be instanced numerous times in the definition of a process network.

A Network Module instance may not be instantiated when the parent NM or VPN is instantiated. The instantiation of the NM may be delayed until a message is sent to a designated 'delay' channel (see Section 5.9.2). Also, the instantiation may be conditional in which case a constraint condition is evaluated to determine whether or not the instance is instantiated (see Section 5.9.1). Constraint conditions and instantiation delays are specified when the NM instance is specified in a VPN or NM definiton.

5.9 Control Flow

Several means of control flow exist in virtual process networks. Mechanisms exist for the control of the dynamic creation of the network as well as the control

of progress of the Process Objects within.

5.9.1 Constrained Instantiation

The instantiation of a Process Object or Network Module may be constrained. This is accomplished by defining a constraint condition upon its creation. When a parent NM is instantiated the constraint condition is evaluated. The constrained PO or NM is instantiated only if the result of the constraint is true. Constraint conditions may access constants, parameters, or channels in the parents scope and can communicate between the system and object layers. The specification of constraint conditions is described in Chapter 6.

5.9.2 Delayed Instantiation

Delayed Network Module or Process Object instances and corresponding 'delay' channels may be used to control the dynamic instantiation of the Virtual Process Network. The delayed instance is instantiated when the corresponding 'delay' channel is first accessed for communication by an existing PO. This first access of the 'delay' channel is interpreted by DPOS as a demand for the instantiation of the delayed instance. The system creates the instance and then processes the message as a normal message. Controlling VPN instantiation using delayed instantiation is similar to demand-driven, lazy evaluation in sequential, functional, programming. Demand-driven, lazy VPN instantiation shares the same advantages that demand-driven, lazy evaluation, programming has. The criterion for process creation does not need to be defined until the processes are actually needed and control can be via demand propagation. This makes the programmer's job simpler because demand propagation allows a single demand to serve as an implicit control for the creation of many elements and also postpones the actual decision until the program has the maximum information available. The programmer can define potentially infinite networks that become partially instantiated at run time the same

way that programmers in sequential lazy languages can define and use infinite data objects.

5.9.3 Blocking

Each PO in a virtual process network is an active process with its own active thread of control. POs may be defined by the programmer to have a bounded limit of computation typical of single-threaded user programs, or as server or generator processes that perform infinitely repeated computation typical of operating system programs. Whichever program type is developed, the progress of the program is regulated by its pattern of communication through its channels. Regulating a Process Object by its message passing behavior means that a Process Object only progresses in response to incoming messages and/or only progresses so long as it is able to send messages. Thus, blocking while waiting for a message to be sent or received is an effective means of control. Blocking does not consume CPU time except for the scheduling and descheduling of the process when it becomes blocked and unblocked. Process Objects that block waiting for input from channels are called 'lazy'. Programming with 'lazy' POs is similar to demand-driven, stream processing in sequential, lazy, functional programming. Process Objects that block while attempting to output to channels are called 'eager'. Programming with 'eager' POs is an effective means of generating parallelism without specifically using parallel branching at the PO level. This programming paradigm supports data flow, pipeline and "compute, aggregate, broadcast" algorithms.

5.9.4 Nondeterminism

The access to a given channel is nondeterministic. This is to say that multiple process objects may simultaneously attempt to access a single channel for input or output. The winner of the competition for access is resolved nondeterministically. This results in nondeterministic merge and fanout of message streams.

CHAPTER 6

DPOS VISUAL ENVIRONMENT

6.1 Introduction

The intent of using a visual programming environment in the implementation of DPOS is to facilitate the programmer in the topological definition of the Virtual Process Network. This allows topologically and interactively complex groupings of parallel processes to be designed in an intuitively clear manner. The design of the system layer of DPOS was intended to be implemented graphically from its inception and to encapsulate the parallel programming issues of network topology, synchronization control and dynamic process creation.

The primary components of the DPOS system, Process Objects, Network Modules and channels are graphically interfaced and are manipulated in constructing process networks. The primary use of the graphical interface is for building Network Module definitions composed of Channel, Process Object and Network Module instances. The channel definitions are an integral part of the system and channel type definitions are not modifiable by the programmer. The Process Objects are single threaded and are designed outside of the visual programming environment and incorporated into it. The programmer uses NMs, POs and channels as black boxes within the visual programming environment. The user defines Network Modules using other nested Network Modules, Process Objects and Channels by visually arranging them on the screen and specifying channel types, connections, Network Module nesting, delayed instantiation, delay channels, constraint conditions and static parameters to process objects and domains.

6.2 Representation Issues

The choice of details of the icons used for the elements is highly subjective. It is important not to over-illustrate either minor qualities of the system or unnecessary information in order to avoid visual clutter. It is also necessary to show enough information to convey the essential structure of the program relationships. Abstraction is a primary concern in visual line programming for two reasons. Firstly, it allows the programmer to define relationships without unnecessary specificity. Secondly, it provides the programmer with a means of reducing the visual clutter by encapsulation of low-level details within groupings. Using abstraction, the programmer can manipulate the program without being immersed in details. This system operates at a high level of abstraction (the Process Object and its parameters is the lowest level involved). The programmer also has the facility to define abstractions in the form of Network Modules.

6.2.1 Choice of Graphical Representations

Several decisions influenced the choice of graphical representation for different features of the DPOS system. The basic building blocks of the underlying system are blocks, connection lines and text. In order to allow more information to be represented within the image with the least additional clutter several different representational devices were employed. These include dashed lines, outlines, area filling and text concatenation.

The graphical features rely on the different perceptual elements of shape, area, texture, line and text. This allows the user to scann an image for visual cues rather than utilize the more detailed comparison required to differentiate text strings that are visually similar to one another. This is the same principal used in the design of traffic symbols that allow drivers to easily identify road conditions and regulations. For example, if the user is looking for information concerning the dynamic creation of objects he or she quickly scans the image for dashed outlines which indicate

delays and constraints. More detailed textual information may then be obtained for the object in question.

Text representation has been limited to labels for Network Module and Process Object instances, connection stream accessor functions, and to separate parameter menus. Comments are encouraged within the graphical image but are not mandatory. The choice of and use of comments is dependent on the complexity and level of visual clutter of individual windows.

6.3 Overview

The DPOS graphical editor is built upon a pre-existing graphical editing system called VIPER developed by Todd Spencer[29]. The existing system has been revised and extended to accomodate the specific needs of the DPOS system. The editing environment uses windows and mouse cursor inputs as well as text interaction (see Section 6.7).

The graphical editor is a block diagram manipulating system. The components of the system include Network Module, Process Object, and channel blocks. The tasks performed by a programmer using the graphical editor include:

1. Definition of Process Object blocks which are the graphical block images to be used to represent Process Objects within the graphical environment. As a by product the system also creates a Scheme source template to be used as the function definition header when the user defines the Process Object text.

2. Definition of Network Module blocks which are the graphical block images used to represent Network Modules within the graphical environment. These are composed of Process Object instances, channel instances and Network Module instances. Definition of a Network Module block creates as a by product a Scheme source file which may be executed interpretively or compiled and run in the parallel programming environment. When this code is executed, NM

definitions create the processes and subnetworks incorporated within Network Module.

The graphical editor is window oriented. The programmer opens one or more windows onto graphical data files (one data file per window). The programmer creates Process Object blocks and Network Module blocks by duplicating and then modifying existing templates. The edited templates are then 'script'ed out to produce three file types. These include the block template which may be reedited, the block image which may be incorporated as a component in other Network Module definitions, and the Scheme source file which is the Scheme definition corresponding to the Network Module (or the Scheme template in the case of a Process Object definition).

6.4 Block Diagram Definitions

This section describes the components and symbols of block diagrams generated using the DPOS graphical editor. Corresponding diagrams are shown in Figures 6.1 and 6.2. Legends of symbols used by the graphical editor are listed below with a short discussion of each. Note that the 'DOMAIN' parameter shown in the legends is to be used for the specification of locality of the Process Object or Network Module in a distributed environment for Concurrent Utah Scheme[30] output. However, Concurrent Utah Scheme is not incomplete as of the date of this report.

1. Basic Blocks: The basic blocks legend shows three instances of Network Module components. From top to bottom are: 'p1:p-sieve' Network Module instance p1 of class p-sieve, 'pu1:p-unit' Process Object pu1 of class p-unit, and an asynchronous channel instance. The distinguishing features are the reversed image characteristic of Network Modules, simple rectangle of the Process Object and the round shape of channel instances. Port locations defined for

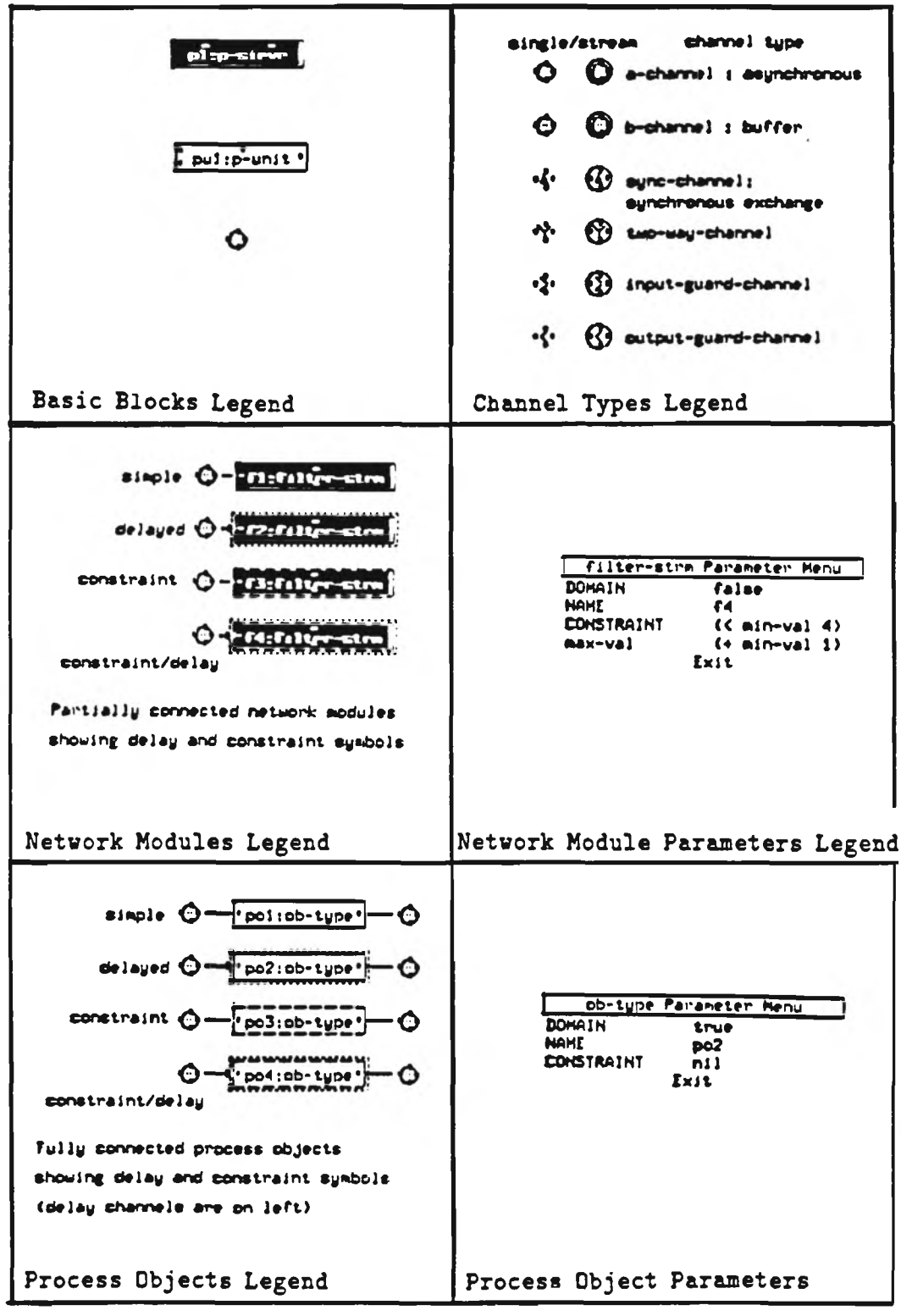


Figure 6.1. Block Diagram Definitions

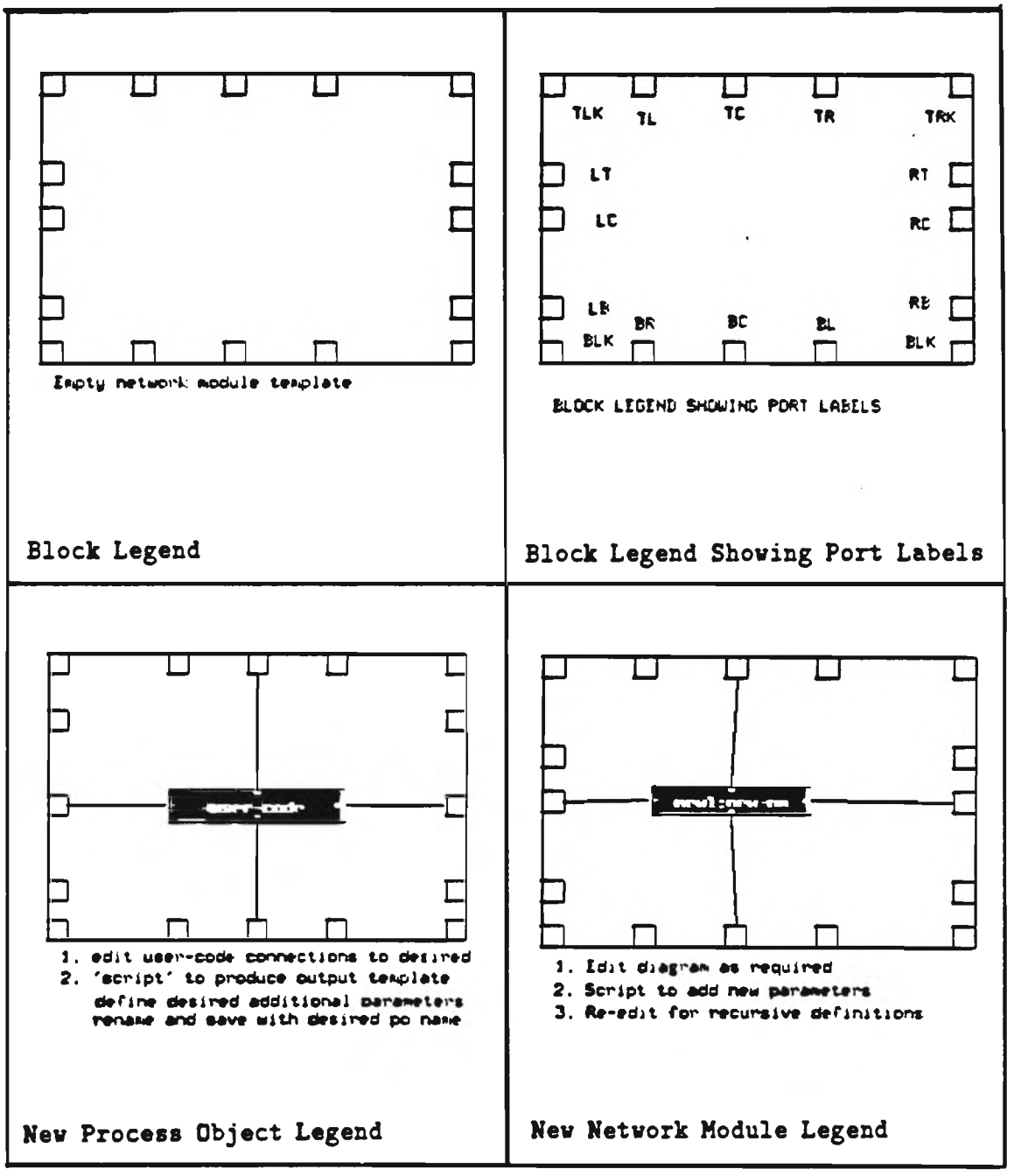


Figure 6.2. Block Diagram Definitions Continued

the particular class are indicated as small rectangles at block perimeters. Ports represent those formal parameters in the Process Object and Network Module definitions that correspond to channel instances. Port locations are used to form connections between channel, NM, and PO instances. (See Figures 6.1 and 6.2.)

2. Channel Types: The channel types legend lists the individual channel icons in single and stream form for channel types: a-channel (Asynchronous 1 way), b-channel (buffered asynchronous 1 way), sync-channel (synchronous exchange), two-way-channel (1 way / 2 way interface), input-guard-channel (synchronous 1 way guarded input), output-guard-channel (synchronous 1 way guarded output). The functionality of the channel types is discussed in Chapter 5.

3. Network Modules: The Network Modules legend lists Network Modules of an arbitrary class type 'filter-strm' with names 'f1' .. 'f4' connected to buffer channels. Delay relationships are indicated by an outer dashed outline and a rectangle at the connection point (to indicate which connected channel is the delay channel). Constraint conditions are indicated by an inner dashed line. The ports for the 'filter-strm' class are TC, BC, LC. These refer to top center, bottom center and left center respectively. Only LC is connected.

4. Network Module Parameters: Shows the parameter menu for the Network Module 'filter-strm.' The parameter 'DOMAIN' is set to 'false' indicating that there is no separate environment needed to instantiate the NM (this is true for all NMs). 'NAME' the instance name (see Network Modules above) of this Network Module is 'f4.' 'CONSTRAINT' indicates that a constraint condition has been placed on the creation of this module (< min-val 4) where 'min-val' must be lexically visible in the scope of the environment creating 'f4.' The parameters DOMAIN, NAME and CONSTRAINT are common to

all Network Modules. The only user defined parameter for the Network Module 'max-val' is the computed value (+ min-val 1).

5. Process Objects: Is similar to the Network Modules legend. The significant difference here is that the block images are not reversed. This Indicates that they have an associated domain (process) and is typical of all Process Objects. The ports for the 'ob-type' class are RC and LC.

6. Process Object Parameters: Shows the parameter menu for the Process Object 'PO2' of type 'ob-type.' 'DOMAIN' is true (for all Process Objects). 'CONSTRAINT' is set to 'nil' indicating no constraint on this Process Object and it has no user defined parameters. This legend is similar to the Network Module Parameters legend which shows different parameter options. The 'DOMAIN' and 'CONSTRAINT' parameters are system parameters and are not visible within the PO definitions.

7. Block Legend and Block Legend Showing Port Labels: Show empty templates for Network Modules (similar for Process Objects). This is the inside of a Network Module but without any inner constituents.

8. New Network Module Legend: The New Network Module Legend shows the template used for defining new Network Module classes. The user should delete the block 'new1:new-nm' and install new PO, channel, and NM instances as required (see Section 6.5).

9. New Process Object Legend: The New Process Object legend shows the template used for defining new Process Object classes. It is similar to the New Network Module Legend. The user deletes connections and adds connections as required for the channel connections for the class being defined. The user should not delete the block 'user-code.' (see Section 6.5)

6.5 Editing Command Summary

This section enumerates a subset of the commands available for the DPOS graphical editor. Additional commands may be found in the VIPER[29] thesis.

1. window: Creates a new editing window and environment. Placing the mouse in the window enables editing in the new window.

2. new-nm: Loads in a copy of the blank Network Module for editing.

3. new-po: Loads in a copy of the blank Process Object for editing.

4. script: Enters the scripting sequence for generating Scheme executable output (also generates block image files). The user answers a series of questions:

   (a) Model name ?: Enter the new class name or default if no change.

   (b) Model parameters menu appears.

   (c) Insert parameter ?: User may enter a user defined parameter for the class type. This does not include port definitions (those are generated automatically).

   (d) Delete parameter ?: User may delete a user defined parameter.

   (e) Continue editing parameters ?: User may loop again through insert, delete and continue.

   (f) Save current block diagram ?: This is identical to the save command listed below.

   If there are irregularities in the Network Module definiton, then the user is asked other questions also. These include specifying unedited parameters or names to instances within the window.

5. save: Saves the window contents for the graphical editor only. This does not produce Scheme executable files, only graphical image files for Process Object or Network Module definitions. The user is asked to specify a new name for the image.

6. exit: Exit system (or window if not root window).

7. zoom: Scales the window image (i.e., zoom .8 .8) will scale to .8x.8 scale.

8. read: Read in a Network Module of Process Object class template for editing (i.e., read filter-strm).

9. delete: Delete object under the cursor. The object may be a Network Module, Process Object, connection or a comment.

10. move: Move object under the cursor

11. edit: Edit object under the cursor. In the case of a Network Module or Process Object block this brings up the edit menu for the object. If it is a connection the user edits only the accessor function for the connection. This is useful only for a stream of channels and the accessor must be one of: h, t, ht, htt, tt. For head, tail, head of tail, head of tail of tail or tail of tail of the stream of channels.

12. ctrl C-l: Redraw window.

13. connect: Establish a connection between a Network Module or Process Object and a channel or port. The 'source' must be a Network Module or Process Object. The 'destination' must be a channel or port of the surrounding Network Module. This does not indicate direction of traffic flow. If a channel is selected as the destination then the user is queried whether the connection is to be a 'delay' connection.

14. copy: Copy the object under the cursor. The user postions the new copy with the mouse cursor.

15. get: Get a Network Module instance, Process Object instance or channel instance for inclusion in the Network Module currently being edited. (i.e., get a-channel) (i.e., get filter-strm)

16. parameters: Opens the help window with the parameter list for the Network Module or Process Object currently being edited.

17. resize: Resize a block in the current NM being edited.

18. help: Opens the 'help' menu.

19. clear: Clears the work space in the root window.

6.6 Editing Operations Summary

This section defines a set of common operations for generating programs using the DPOS graphical editor. Each item gives the operations to use in performing each task.

1. Creating a new Process Object class:

   (a) Use 'new-po' to get an instance of the blank PO template.

   (b) Edit the template by connecting 'user-code' to the desired ports.

   (c) Edit connections to specify accessor functions if the port represents a stream of channels.

   (d) Use 'script' to generate the Scheme executable file, parameters (other than port connections) and block images for use in other NM definitions. Be sure to change the name from 'new-po' or 'new-nm' to some other class name.

(e) Use the Scheme executable 'class-name.std' when editing the Process Object definition on a text processor. This file will contain a template showing the necessary calling sequence for the PO definition.

2. Creating a new Network Module class:

   (a) Use 'new-nm' to get an instance of the blank NM template.

   (b) Use 'delete' to remove the instance of 'new-nm' included. The 'new-nm' instance is a dummy instance to indicate how connections and instances should look within an NM definition.

   (c) Use 'get' to get instances of channel types, Process Object types and Network Module types for inclusion into the Network Module class.

   (d) Use 'connect' to connect included NMs and POs to channels and ports.

   (e) Use 'move' and 'resize' to configure the contained POs and NMs.

   (f) Use 'edit' to edit PO parameters, NM parameters and connection accessor functions (for stream channels only).

   (g) Use 'script' to generate the Scheme executable file, parameters (other than port connections) and block images for use in other NM definitions. Be sure to change the name from 'new-nm' to some other class name.

3. Editing an old Process Object or Network Module class: Use 'read class-name' to input the Network Module or Process Object class diagram for editing. Then proceed as in creating a new PO or NM after the initial 'new-nm' or 'new-po' step.

4. Attaching channel instances:

   (a) Use 'connect' to establish a connection from a PO or NM to a channel instance. Only one channel may be connected to any NM or PO port

(with the exception of stream channels). Any number of POs or NMs can be connected to a channel or template port. An NM or PO port must be selected as the source and a channel or template port must be the destination. Delays are specified at this time.

(b) Use 'edit' to edit the connection accessor function in the case of stream channels.

5. Editing channel instances: Use 'edit' to edit the channel definition menu. The user must specify the name of each channel instance in a class definition. In the case of buffer channels the user must specify the maximum size of the buffer. In the case of input and output guard channels the user must specify a guard list (list of integers) to be used as identifiers for guards (the list is 0...n) for the list of guards.

6. Editing instances of Network Module or Process Object: Use 'edit' to edit the Network Module or Process Object instance parameter menu. The user must specify a name for each instance. The user defined parameters must be specified. The parameters DOMAIN and CONSTRAINT have default values. The DOMAIN default value should not be edited.

7. Editing parameters of Network Module or Process Object classes: This is done when editing the Network Module or Process Object class by using the 'script' command. (See Item 1 above.)

6.7 Existing VIPER System Base

The existing VIPER system supports a wide range of graphical programming facilities for object-oriented programming and simulation. The VIPER system is written in Common Lisp using the Frobs object system[21], the X window system and Gnu Emacs for interfaces.

In VIPER, objects are represented as rectangular blocks and connection relationships (methods or physical links in the case of circuit simulation) are represented as arrows between blocks. Among the most advanced features is the system's ability to handle abstraction of groups of objects which may be contracted into icons or expanded as groups (groups, constractions and expansions). It is also capable of handling multiple connections to an object, to incorporate textual files as the object definitions, and to represent multiple levels of nesting with multiple display windows. Connections are represented as directional arcs and strict directionality of connections is enforced. Any connection may output to several others but may only input from a single source and only a single connection is allowed to any port on an object. Facilities are provided for text and numeric parameters to be applied to objects.

6.7.1 Modifications to VIPER

Modifications of the VIPER environment were necessary for several reasons. It was neccesary to accommodate nested lexically scoped network topologies, three types of blocks (Network Module, Process Object and channel), and specialized block attributes such as delays, constraints and stream channels.

The following is a list of modifications to the VIPER system.

1. The facilities for editing groups, expansions and contractions were found to be specialized for groupings within a flat network topology. Rather than work with these constructs the 'window' command was added to allow multiple editing windows to be used each with its own environment. This is more directly applicable to the nested scoping of the DPOS network topology.

2. Blocks were differentiated into the three primary types Process Objects, Network Modules and Channels.

3. Icon representations for Process Objects, Network Modules, Channel types, stream channels, constraints and delays were added.

4. Generic parameters were added to block definitions. These include:

   (a) NAME: for instances of all three types of objects.

   (b) DOMAIN: for instances of all three types of objects.

   (c) CONSTRAINT: for Process Object and Network Module instances.

   (d) BUFFER SIZE: for buffer channel instances.

   (e) GUARD LIST: for input and output guard channel instances.

   (f) STREAM: for all channel type instances.

5. Connections were modified in several ways for the DPOS environment.

   (a) Directionality of data flow is no longer defined within the environment.

   (b) Connections are restricted. 'Source' is restricted to Network Modules and Process Objects. 'Destination' is restricted to channel types.

   (c) Multiple 'destinations' are no longer supported. This means that only one channel may be connected to any one Process Object or Network Module port.

   (d) Multiple 'sources' are now supported. This means that multiple Network Modules or Process Objects may be connected to a single channel port.

   (e) Accessor functions for accessing stream channels has been added to connection definitions.

   (f) Delay was added as a property of connections.

   (g) Differentiation between ports in channel objects has been removed. This means that it does not matter to which port of a channel that a PO or NM connects. (The reverse is not true.)

6. Recursion in definitions of Process Objects and Network Modules is now supported.

7. Scheme post processing was added to generate Scheme output files along with the graphical block files. This attributes lexical scoping semantics to the network topology and allows mutual recursion in NM definitions.

8. Unmodified features of the VIPER system were generally not deleted. Several features were thought to be obsolete and not useful in editing DPOS networks and were left as is. These leftover features may not be completely compatible with the DPOS environment include 'Group', 'Contract' and 'Expand' operations. These are tailored to working with a flat network topology and do not generate nested scoping or allow modifications of port connections or multiple port connections. The utility of these features is replaced by the multiple windowing capacity and recursively nested scoping possible in the the DPOS environment.

9. Help Menus for information and accessing Scheme objects were added.

10. Systid features and help menus were deleted. These included Systid post processing and library functions.

CHAPTER 7

PROGRAMMING METHODOLOGY AND DEVELOPMENT

7.1 Introduction

This chapter presents a comparison between object program development using the DPOS system with sequential object definition in Scheme and with parallel object definition using explicit low-level parallel constructs (locks and futures) in Butterfly Scheme[26].

This chapter also describes a programming methodology (Dynamic Programming) and shows how the methodology may be extended to develop virtual process networks. The Sieve of Erasthones algorithm is used as an example program. The basic algorithm is developed using parallel dynamic programming methodology and implemented as a Network Module.

7.2 Implementation Comparison

This section discusses the implementation of a hypothetical closure style object as a single-threaded object in a sequential programming environment, as a parallel object that uses explicit synchronization and parallel constructs in 'Butterfly Scheme' and as a DPOS process network. The object has features and parallel implementation requirements typical of object definitions employing several methods with requirements for partially or fully sequential execution of methods.

This object is a component in a hypothetical network of objects. A description of the operation of the object is nested cycles of activities.

In each cycle the steps are:

1. Initialize object slot 'initvals'.

2. Inner cycle. Cycle number-of-updates times through update methods of types update-a and update-b with the stipulation that update-a and update-b must cycle in pairs. Either update-a or update-b may occur first but both must occur exactly once in each inner cycle. These methods propagate subresults to other objects.

3. Method 'result' is then used to claim results and reset the object.

In a single threaded environment it is assumed that the correct order of evaluation leads to an appropriate cycling of update methods.

```
(define (sequential-net-obj next-a next-b)
  (let ((initvals #f)
        (update-lst #f))
    (lambda (op . vals)
      (cond ((eq? op 'init) (set! initvals (car vals)))
            ((eq? op 'update-a)
             (let ((newval (process-a (car vals))))
               (set! update-lst (cons newval update-lst))
               (next-a 'update-a newval)))
            ((eq? op 'update-b)
             (let ((newval (process-b (car vals))))
                (set! update-lst (cons newval update-lst))
                (next-b 'update-b newval)))
            ((eq? op 'result)
              (final-processing
                 number-of-updates initvals update-lst))))))
```

In a parallel implementation the order of incoming methods is nondeterministic so the object must provide synchronization control for the sequence of method execution. This may be accomplished using locks (semaphores) and futures and retaining the same 'passive' object role.

```
(define (parallel-net-obj next-a next-b)
  (letrec
```

```
((make-lock! init-lock) (make-lock! update-lock)
 (make-lock! update-a)  (make-lock! update-b)
 (make-lock! result-lock) (number-of-updates #f)
 (update-count-a 0)(update-count-b 0)
 (update-count 0) (initvals #f) (update-lst #f)
 (unlock-test
  (lambda()
    (if (= update-count-a update-count-b)
      (begin (set! update-count (+ update-count 1))
             (if (< update-count-a number-of-updates)
               (begin (unlock-lock! update-a)
                      (unlock-lock! update-b)
                      (unlock-lock! update-lock))
               (unlock-lock! result-lock)))
      (unlock-lock! update-lock))))))
(lock-lock! update-lock) (lock-lock! result-lock)
(lock-lock! update-a) (lock-lock! update-b)
(lambda (op . vals )
  (cond ((eq? op 'init) (lock-lock! init-lock)
                        (set! number-of-updates (car vals))
                        (set! initvals (cdr vals))
                        (unlock-lock! update-a)
                        (unlock-lock! update-b)
                        (unlock-lock! update-lock))
        ((eq? op 'update-a)
          (lock-lock! update-a)
          (lock-lock! update-lock)
          (future (let ((newval (process-a (car vals))))
                    (set! update-lst (cons newval update-lst))
                    (next-a 'update-a newval)
                    (set! update-count-a (+ update-count-a 1))
                    (unlock-test))))
        ((eq? op 'update-b)
          (lock-lock! update-b)
          (lock-lock! update-lock)
          (future (let ((newval (process-b (car vals))))
                    (set! update-lst (cons newval update-lst))
                    (next-b 'update-b newval)
                    (set! update-count-b (+ update-count-b 1))
                    (unlock-test))))
         ((eq? op 'result)
           (lock-lock! result-lock)
```

```
        (let
          ((res (final-processing
                  number-of-updates initvals update-lst)))
          (unlock-lock! init-lock)
          res))))))
```

Modifying object descriptions this way requires a great deal of synchronization control to be added to the object definition. This approach also mixes parallel synchronization issues with the functionality issues of the object obscuring both.

Using Process Object semantics the object may be treated as a single threaded program using channels as surrogates for the methods of the above implementations. The Process Object definition uses guarded input channels IN-AB, OUT-A, OUT-B and asynchronous channels RESULT and INIT. Note the use of 'meth-list' and 'new-meth-list' with the guarded input channels to handle the nondeterminstic inner cycle by passing the appropriate guard type list to IN-AB.

```
(define (net-obj INIT RESULT IN-AB OUT-A OUT-B)
  (letrec
    ((update-cycle (lambda (count update-lst meth-list)
       (if (> count 0)
        (let* ((method (IN-AB 'readguard meth-list))
               (meth-type-a (if (eq? (car method) 'update-a) #T))
               (newval ((if meth-type-a process-a process-b)
                        (cadr method)))
               (new-meth-list (remove (car method) meth-list)))
          (if meth-type-a (OUT-A 'update-a newval)
                          (OUT-B 'update-b newval))
          (update-cycle (if new-meth-list count (- count 1))
                        (cons newval update-lst)
                        (if new-meth-list new-meth-list
                                          '(update-a update-b))))
        update-lst))))
    (let* ((vals (INIT 'receive))
           (number-of-updates (car vals))
           (initvals (cdr vals))
           (results
              (update-cycle
                number-of-updates initvals '(update-a update-b))))
```

```
    (RESULT 'send (final-processing
                        results number-of-updates initvals))))
(net-obj INIT RESULT IN-AB OUT-A OUT-B))
```

Using Process Object semantics greatly reduces the length and complexity of the object definition. The issues of managing parallelism and object functionality are separated. The Process Object definition contains no critical sections and may be developed and tested as a single threaded program on a uniprocessor using sequential channel definitions. The potential blocking points within the program are explicitly evident as accesses to the channels (shown capitalized).

7.3 Dynamic Programming

Dynamic Programming is a computational technique that converts multistage multivariable computational problems into a series of single or few variable problems. It is a methodology for developing algorithms based on the compositions of solutions of subproblems. The term Dynamic Programming does not refer to computer programming but to the fact that the computation is a series of discrete steps. This methodology has a wide range of uses in mathematics, engineering, business and science.

The use of a single methodology is not meant to imply that this is the only applicable approach. But only to present an example approach that has wide application and yields programs that work efficiently.

7.3.1 Properties of Dynamic Programming

There is not a specific algorithm used in dynamic programming, rather, it is a problem solving methodology. Dynamic programs are composed of a sequence of computation 'stages'. Each stage is composed of a set of computed states. Several properties distinguish Dynamic Programming from other problem solving methodologies.

1. A sequential decision problem with n decision variables is converted into n subproblems each with a single variable. These subproblems are called 'stages.'

2. The sum of computed states output from a stage plus the input data (sum of all output data from previous stages) are the complete information needed for all computated states at the next stage.

3. The principle of optimality is used to produce an optimal solution from composition of optimal subsolutions.

4. Bottom-up design is used. This distinguishes dynamic programming from divide and conquer strategies. Problems need not have a final result but may be continuous processes. Composition of subresults is not simply recomposition of the divided parts as in divide and conquer (i.e., the results may be used in several combinations at successive stages).

Many dynamic programming problems can be performed within a structured tableau and this is the approach used here. The tableau for a stage consists of a sequence of frames corresponding to stages. Each stage frame contains input data (from some previous stage), start data (aggregation of accumulated data), and a description of the states to be computed. The development of dynamic programming algorithms may be described as undertaking the following steps:

1. Define stages and states.

2. For each stage define decision variables, constraints and functions using only the input data and start data.

3. For each stage define output functions for generating results (optimal values) to be used in successive stages.

7.3.2 Application to Parallel Processing

Dynamic programming stages have the property that the computational outputs at one stage are the complete information needed for all computation at the next stage. Because of this, all computations for a stage are data and logic independent and may be carried out in parallel. Once a problem has been expressed as a dynamic program the potential parallelism at each stage is evident. The issue to be resolved in expressing it as a parallel dynamic program is the partitioning of each new stage into parallel processes. The criterion for this partitioning is to minimize the amount of data passed between processes from stage to stage. If the same processes can be reused in successive stages then the amount of data passed between stages can often be greatly reduced or eliminated.

7.4 Prime Number Sieve Program

The compution of prime numbers is commonly expressed as the generation of a list of primes by filtering successive odd numbers through the list of already computed primes. Numbers are tested for divisibility and against their square roots. For example the number 11 would be filtered through the list (3 5 7). The number 3 is less than the square root of 11 and 11 is not divisible by 3 so it must be compared against the next prime. The number 5 is greater than the square root of 11 so we know at that point that 11 is prime and should be added to the end of the list. The number 13 is then filtered through the resulting list and so on. This approach leads to implementations with two nested loops where neither loop has guaranteed logic of data independence between successive iterations.

Expressing the algorithm as a dynamic program takes a different form. Assuming that some number of primes has been previously computed, stage(i-1) will compute a subsequent set of primes and this subsequent set is input data to stage(i). The start data for stage(i) is the set of previously computed primes plus all odd integers not yet considered.

In general for any stage(i) the tableau is:
Start data: odd integers $> m$, all primes $< l$
Input data (computed in stage $i-1$): $l \leq$ primes$< m$
States(s): The filtering of all odd integers j: $m < j \leq m^2$

For example given stage(i) where $l = 4$ and $m = 9$:
Start data: odd integers > 9, primes: (3)
Input data: (5 7)
States(s): The filtering of all odd integers j: $9 < j \leq 81$

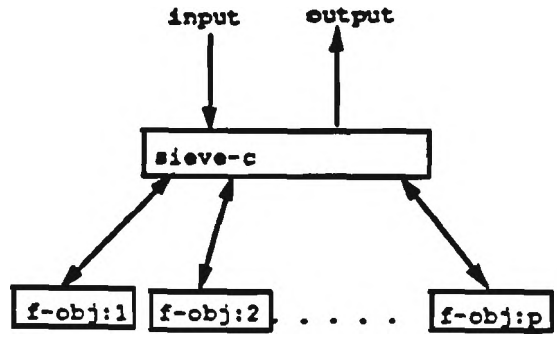
The states s may now be partitioned into p independent processes each filtering a consecutive subrange of the odd integers j. In between stages, the stage results of the p processes must be broadcast to all p processes so that the next stage computation may take place.

The parallel stage tableau for stage i is then:
Start data: odd integers $> m$, primes $< l$
Input data(from stage($i-1$)): $l \leq$ primes $< m$
States(s) are processes s1..sp each of which filters $1/p$ of the range of odd integers from m to m^2.

A diagram for this process network is shown in Figure 7.1. Figures 7.2,7.3 and 7.4 show the implementation of the process network. The processes f-obj:1 .. f-obj:p correspond to the state processes s1..sp. In the network shown, 'sieve-c' initializes the 'f-obj' processes with start parameters including a list of starting primes and range limits for filtering. Each f-obj filters its range of integers and returns the results to the sieve-c. Sieve-c then broadcasts the aggregation of the results along with new subrange limits to the f-obj processes and the cycle repeats.

The aggregation phase of the stage cycle causes a synchronization of the program while the sieve-c process concatenates the p result vectors and broadcasts them. During this time the f-obj processes must wait idle.

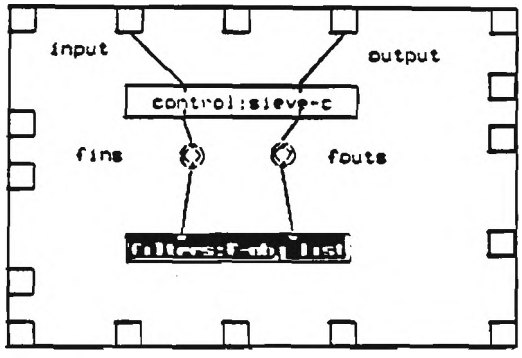
In the following subsection further refinement of the algorithm eliminates this synchronization by subdividing the dynamic program stages into substages.



f-obj: processes each process 1 segment of the range of integers from m to m*m.

sieve-c: broadcasts and aggregates subresults from f-obj processes

Figure 7.1. Sieve Network



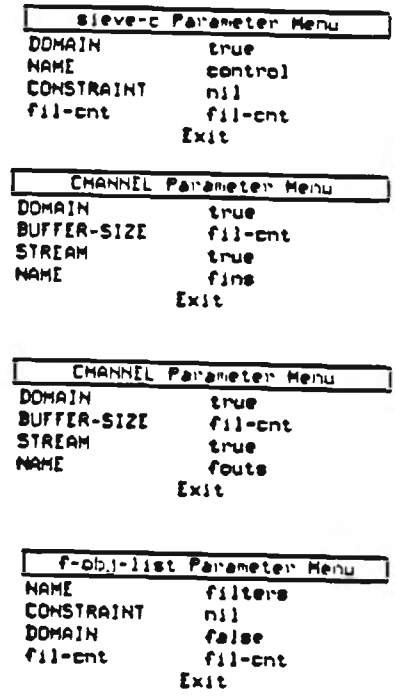


Figure 7.2. P-sieve Network Module and Parameter Menus

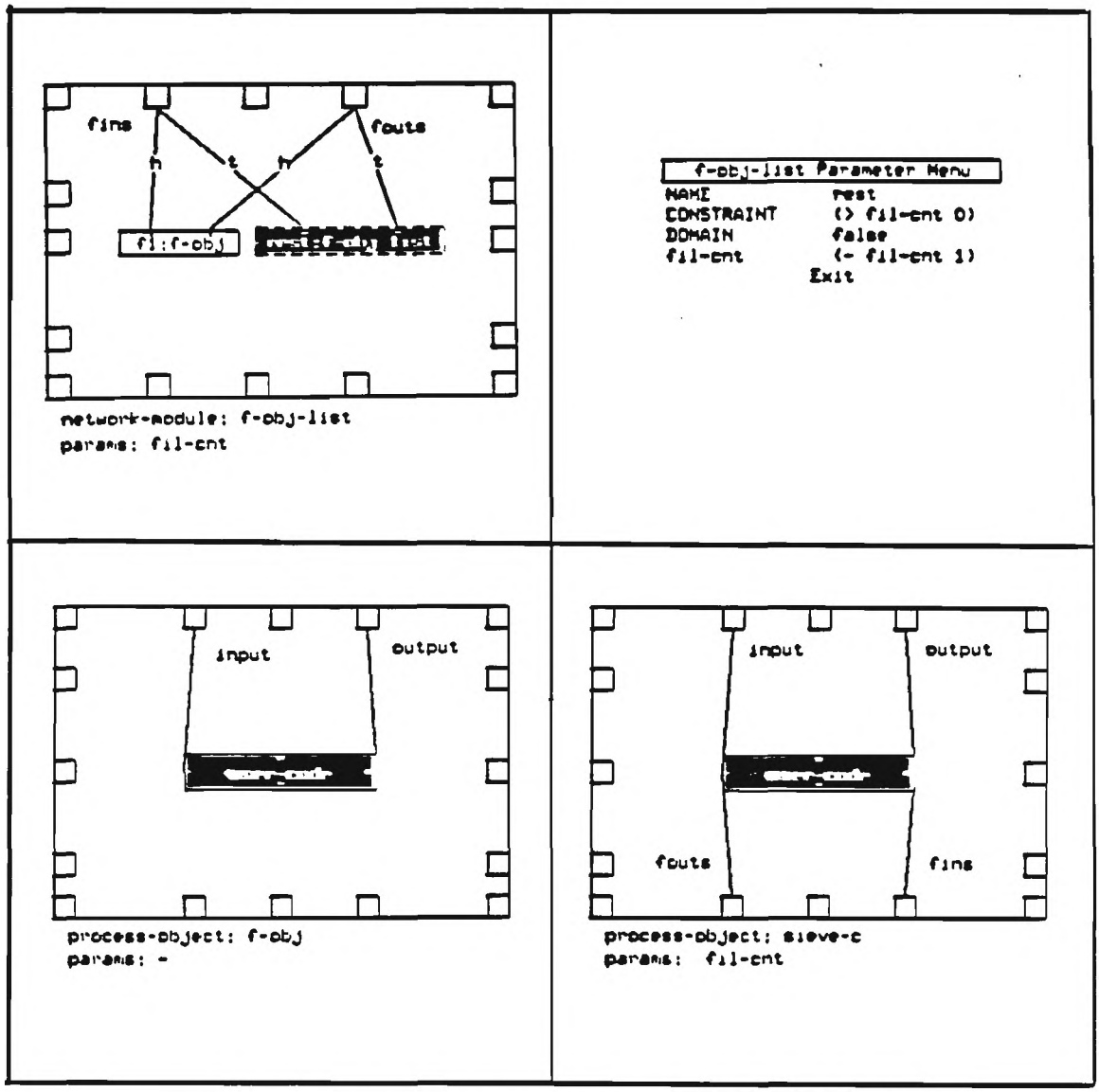


Figure 7.3. F-obj-list NM and Process Object Templates

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; F-OBJ process object definition
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Template output from graphical interface.
;; ParametersTL and TR correspond to graphical port locations
;;
;;(define (f-obj   TL TR)
;;  (letrec( (NIL (USER-CODE  TL TR)))
;;   (list  TL TR)))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Re defined f-obj using parameters from template.
;; TL is renamed: INVAL
;; TR is renamed: OUTVAL
;;
;; Function:  input maximum prime value and call f-obj2

(define (f-obj INVAL OUTVAL)
  (f-obj2 INVAL OUTVAL '((9 3)) (INVAL 'receive)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; f-obj2
;; Function:  input from INVAL: range limit values and a new list of primes
;;            if input is null then quit
;;            else
;;              add new primes to primes already received
;;              filter all odd integers within range limits
;;              output to OUTVAL: list of new primes computed
;;              repeat cycle
;;
;;  Note Input format: (start-of-range end-of-range new-prime-list)
;;       plst is a list of ((prime-squared prime) (prime-squared prime) ...)

(define (f-obj2 INVAL OUTVAL plst big-lim)
  (let*((limits (INVAL 'receive 1))
        (startlim (if (odd? (car limits)) (car limits) (+ (car limits) 1))))
    (if limits
      (begin (my-append plst (copylst2 (cddr limits)))
             (OUTVAL 'send (filter-range startlim (cadr limits) plst big-lim))
             (f-obj2 INVAL OUTVAL plst big-lim)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; helper functions

;;filter the range of integers

(define (filter-range startval endval plst lim)
  (cond ((> startval endval) #f)
        ((filter1 plst startval)
         (let* ((lim1 (if lim (* startval startval)))
                (lim2 (if (and lim1 (< lim1 lim)) lim)))
           (cons (list lim1 startval)
                 (filter-range (+ startval 2) endval plst lim2))))
        (#t (filter-range (+ startval 2) endval plst lim))))

;; filter one candidate against the list of primes computed

(define (filter1 plst candidate)
  (cond ((null? plst) candidate)
        ((> (caar plst) candidate) candidate)
        ((> (remainder candidate (cadar plst)) 0)
         (filter1 (cdr plst) candidate))))

;; copy the list

(define (copylst2 lst)
  (if (and lst (list? lst))
      (cons (copylst2 (car lst))(copylst2 (cdr lst)))
      lst))

;;side affecting append

(define (my-append lst1 lst2)
  (if (cdr lst1)
      (my-append (cdr lst1) lst2)
      (set-cdr!  lst1 lst2)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Figure 7.4. F-obj Process Object Definition

7.4.1 Refined Prime Number Sieve

The synchronization of the previous algorithm is due to the data dependencies between subsequent stages in the dynamic program. The f-obj processes must wait for the aggregation and broadcast of all results from stage(i) before proceeding to stage($i+1$).

The optimization of the previous dynamic program algorithm involves the subdivision of each stage into substages in such a way that there is no direct data dependency between consecutive substages. That is to say that the data from substage(i) is not needed for substage($i+1$) but is needed in some future substage.

In stage(i) the results of f-obj:1 is the computation of all primes between m and some number m1. If this data alone were used as the input to stage($i+1$) it would allow the computation of all primes between m^2 and $m1^2$. In stage(i) process f-obj:2 computes all primes between $m1$ and $m2$. If this data alone were used as input to stage($i+2$) it would allow computation of all primes between $m1^2$ and $m2^2$. In this case the computations of stage($i+2$) are not dependent on the results of stage($i+1$) but only on results of segment 2 of stage(i).

Redefining the input data for stages in this way eliminates the synchronization at stage boundaries. If stages are numbered using base p then input data for stage(i) comes from state(i mod p) of stage(truncate(i/p)). The dynamic programming tableau for this modified algorithm is the same except for this change and the same process network sketch is appropriate.

The total cost of computation of all primes up to n (without overhead) is between O(n) and O($n^{\frac{3}{2}}$). The total overhead cost of broadcasting all primes up to $\sqrt{n}$ is less than O($p\sqrt{n}$). Using constant $p \ll \sqrt{n}$ and the knowledge that the number of primes less than $\sqrt{n}$ is much less than the $\sqrt{n}$ we can see that the ratio of granularity/overhead and thus the computational efficiency is a continuously increasing function which is much greater than O($\sqrt{n}$).

7.4.2 Implementation of Prime Number Sieve

The implementation of the refined sieve algorithm is shown in Figures 7.2, 7.3 and 7.4. The stages followed in implementation of the Sieve algorithm follows:

1. Analysis and development of the algorithm as a parallel dynamic program.

2. The parallel dynamic program is sketched and the decision to use buffered communication channels is made in order to store substage results for future use. It is also decided to allow a variable number of **f-obj** filters represented by the parameter 'fil-cnt'.

3. Individual Process Object blocks for **sieve-c**, **f-obj** are designed using the graphical interface from Process Object templates (see figure 7.4).

4. The list of **f-obj** processes is designed as Network Module **f-obj-list** as a recursive Network Module definition with parameter 'fil-cnt' determining the length of the list.

5. The **f-obj-list** and **sieve-c** are incorporated into a Network Module definition p-sieve that encapsulates the entire definition of the parallel sieve program.

6. The **f-obj** process class is developed on a text editor by filling in templates output from the graphical interface and are tested independently on a uniprocessor.

7. The **sieve-c** process is developed on a text editor by filling in templates output from the graphical interface and are tested independently on a uniprocessor.

8. The final program is then tested on a parallel processor.

7.4.3 Incremental Testing

Before the entire program is tested on a parallel processor, the individual pieces are first tested on a sequential processor in standard 'Scheme.' This is done by using sequential definitions of channels. These sequential channel definitions interact with the user instead of with other Process Objects. The following steps are taken:

1. The channels necessary to completely encapsulate the Process Object to be tested are created:

```
(define ch1 (channel 'ch1))
(define ch2 (channel 'ch2))
(define ch3 (channel 'ch3))
```

2. The Process Object definition to be tested is invoked with channels as parameters and other parameters as necessary.

```
(class-name ch1 ch2 ch3)
```

3. The communication necessary for the operation of the Process Object is now effected between the user and the Process Object. The channels query the user for input and print output to the screen.

```
computer> channel ch1 requests input
user>     7
computer> channel ch2 receives output
computer> ....
```

This level of testing is a complete test of individual PO definitions and does not differ in any semantic way from the actions of the PO in a parallel environment.

After sequential testing of all Process Objects is finished, testing of Network Module definitions on a parallel processor begins. This process is similar to the testing on the sequential processor.

1. The channels necessary to completely encapsulate the Network Module to be tested are created:

```
(define ch1 (channel))
(define ch2 (channel))
(define ch3 (channel))
```

2. The Network Module definition to be tested is invoked with channel parameters and other parameters as necessary. In the parallel Scheme dialect used here this entails creation of a 'future' with the Network Module definition as its argument:

```
(define test1 (future (class-name ch1 ch2 ch3)))
```

3. The communication necessary for the operation of the Network Module is now effected between the user and the Network Module by sending and receiveing messages through channels. The channels are communicated as within the Process Object definitions:

```
user>     (ch1 'send 7)
user>     (ch2 'receive)
computer> ....
```

Incremental expansion of the subnetworks being tested may be accomplished by testing successively outer scoped NM definitions or adding individual Process Object definitions and channels. The only requirement for this method of testing is that the subnetwork begin tested must be completely encapsulated in the necessary channels.

CHAPTER 8

CONCLUSIONS

8.1 Evaluation

In practice the comparison of distributed parallel programming systems is difficult at best, because of the scarcity of models and implementations for comparison and the lack of definitive test criteria. A variety of evaluation results are presented in this report including:

1. Discussions of the semantic improvements of DPOS over existing programming systems in this chapter.

2. A comparison between closure object implementation in sequential programming and in parallel programming using traditional methods and using Process Object semantics in Chapter 6.

3. A comparison between three implementations of alpha beta search is given in Appendix A.

4. Performance statistics for several implemented programs are given in Appendix B.

5. A program design methodology for developing efficient DPOS programs in Chapter 6.

8.2 Contributions

The distributed parallel object system (DPOS) brings together concepts of object-oriented programming and graphical programming with aspects of modern functional languages.

The system defines a clear and simple approach to generating and managing parallelism and interprocess communication in a distributed parallel environment. It contributes several new solutions to the problems of distributed parallel programming that are improvements over existing systems:

1. Stratification: The DPOS system is stratified in order to allow the sequential and parallel aspects of distributed programming to be clearly and uniformly defined with a minimum need for proficiency at low-level parallel programming.

   (a) In the Process Object layer, the programmer develops sequential components of the parallel program in traditional sequential programming styles without the need to deal with concurrency issues such as critical sections. Communication between processes is syntactically similar to file accessing. In addition, the programmer can produce Process Objects that are regulated by interprocess communication (this is stream processing at a very high-level) reducing the need for limit control information and programming in the individual Process Object.

   (b) The Network Module Layer manages the concurrency issues of full dynamic process creation and interprocess communication (critical sections) at a high-level in a graphical environment that accurately and clearly reflects program topology.

2. Graphical Representation:

(a) Provides abstraction at the process network level, thus allowing programmers to develop programs abstractly rather than concretely as in other visual programming systems.

(b) High-level semantics of the graphical representation rather than a one to one correspondence between icons and language elements makes the visual programming aspect of the system a more powerful tool. The high-level semantics includes: concurrency issues of process creation, Network Module abstractions, delayed and recursive Network Modules.

(c) Allows programmers to develop complex topologies visually (and two-dimensionally) rather than textually.

(d) Serves as visual program documentation of process topology not easily described textually.

3. Lazy Evaluation and First Class process status have been incorporated into the DPOS system to provide the programmer with a wider range of programming options including:

   (a) Stream and delayed evaluation programming techniques.

   (b) Process structures that are similar to abstract data structures.

4. Encapsulation of Process Objects and Network Modules:

   (a) Allows more flexible dynamic process creation than is achieved by the other systems studied.

   (b) Encourages the analyses of programs and program fragments using graph theory and data flow analysis by defining a clearly defined network of processes.

   (c) Encourages modularity and reusability by establishing a clearly defined interface between the modules.

(d) Allows the incremental development and testing of modules which can be defined as isolated units along with their surrounding channels and communicated with by the tester via the surrounding channels.

5. Channel Semantics:

(a) Encapsulates the semantics of communication allowing the existence of multiple communication paradigms within the same program, permitting the programmer to program in a sequential style at the Process Object level as the semantics of communication are not incorporated into the base language.

(b) Consistent approach to the critical section issues of data sharing and communication relieves the programmer of resolving critical section problems.

8.3 Limitations

While the research goals of this project were met, several restrictions within the system were found to limit programming flexibility. In addition several potential extensions to the basic goals became obvious during implementation and testing.

At present the output from the graphical interface is limited to the single source language 'Butterfly' Scheme although programs have been translated into Concurrent Utah Lisp. Since the implementation is dependent only on common low-level parallel constructs there is reason to believe that output modules can be adapted to many existing language systems.

Program text editing for Process Objects and the editing source text for network modules have not been integrated into the graphical interface. Their inclusion would provide a more complete programming environment.

The semantic system constructs were designed for medium to large granularity programming. All example programs and discussions are limited to medium to large grain algorithms. Several small grain applications were tested and their

performance was found to be unsatisfactory. Smaller grain applications may be designed with this system in the cases where simplicity of implementation is the dominant requirement rather than parallel performance.

Several nonworking features of the preexisting graphical interface were not addressed in the implementation of the graphical interface for DPOS. The set of implemented features though has been found to allow a range of flexibility in programming.

The system layer has been restricted to the management of parallelism and program topology and has only minimal provision to do other types of computation. In addition the system has been designed with a minimal interaction between the system and Process Object layers. The ability for Process Objects to interact more freely with the system layer and for the system layer to carry out computations may be desirable in certain circumstances.

Passive objects have been limited to the channel types and no provision has been made for user defined passive objects.

8.4 Further Research

In the development of this project several possible directions for future research and refinement have arisen.

The stratification of DPOS gives the system high potential for isolating partitions of DPOS programs. This encapsulation has already been beneficial in incremental program development. However, the encapsulation also has good potential for monitoring and recording program performance statistics by monitoring the behavior of channel objects. This could be applied to statically analyzing program structure, collecting runtime statistics and monitoring runtime performance for program debugging. Coupled with the graphical interface there is the potential for graphical debugging as well as graphically analyzing or replaying runtime monitor data.

Because of the high semantic level of DPOS network module definitions and the reliance on a simple and relatively common set of parallel primitives. Network module definitions could be adapted to output several source languages from a single network module definition.

Passive object types have been limited to predefined channel class definitions. In certain situations it would be desirable to have user specified passive object types. This might require a substantial addition to the existing semantics to resolve critical section communication issues in order to maintain the clarity and simplicity of the system.

As actual program development experience increases, possible beneficial refinements in the basic system semantics to support specific programming paradigms have become evident. For instance in 'data flow' type programming, channel instances are used as one-directional communication. Indicating graphically and semantically-enforcing directionality of channels would add clarity to the network module definitions and also help to avoid high-level race and potential deadlock conditions in this type of programming.

APPENDIX A

ALPHA BETA SEARCH

A.1 Introduction

In this appendix the alpha beta search algorithm and parallel alpha beta search algorithm are discussed. A uniprocessor alpha beta program is presented and compared with two parallel versions. The first parallel program [3,4] uses explicit process creation. It employs semaphores and shared variables to manage interprocess communication. The second parallel program uses a DPOS virtual process network with asynchronous one-way channels for communication.

It is assumed throughout that the rudiments of alpha beta search are understood by the reader. It is also assumed that the reader is familiar with traditional methods of parallel programming using explicit process creation and semaphores and unfamiliar with programming using DPOS.

A.2 Sequential Alpha Beta Search

Function **node-1** below outlines an alpha beta search algorithm in Scheme. The function **node-1** searches a single node in the search tree. If the node is a leaf node then the board is evaluated. If the node is not a leaf node then **node-1** is called recursively to evaluate the next ply level and return the result. After searching at the present branch, the function either recursively searches the next sibling branch or returns the resulting values (**table**) from the present level. The function **prune-it?** is used to determine whether to continue searching at this level or if the remaining branches at this level should be pruned.

```
1  (define (node-1 ply board-set table)
2    (let ((board (car board-set))(return-value())(prune-result()))
3      (if (= ply *maxply*)
4        (set! return-value (static-eval-update board ply table))
5        (let*
6         ((next-boardset (new-move-list board ply))
7          (next-branch (node-1 (+ ply 1) next-boardset
8                               (make-new-table table))))
9      (set! return-value (value-of next-branch))))
10     (set! prune-result (prune-it? ply table return-value))
11     (if (or prune-result (not (cdr board-set)))
12       (make-return-value board table ply)
13       (node-1 ply (cdr board-set)
14            (update-result ply table return-value board)))))
```

A.3 Parallel Implementation Goals

Figure A.1 shows a complete four-level search tree. The tree shows leaf node values and return values for the nodes evaluated. The evaluated branches are shown in bold lines. The branches searched constitute the 'minimum set' of branches that must be searched for a tree of this configuration regardless of the returned node values. The remaining branches may need to be searched if the return values indicate that the leftmost branch does not yield the maximum value at the root node. Results of the leftmost branches cannot produce branch pruning until after the 'minimim set' of branches have been evaluated. The remaining branches may use earlier return values to determine the extent of further searching.

In the parallel implementations of alpha beta search, processes are created that correspond to the individual nodes in the search tree. Child processes of each node share a common value table that contains search limit values to be used in pruning and updating the parent process with final results.

Figure A.2 shows the previous search tree with the nodes labeled as type **A**, **B** or **C**. All children of type **C** nodes are type **B**. The leftmost children of type **A** and **B** nodes are type **A** and all others are type **C**. The type labels correspond to searching behaviors associated with nodes at various points in the tree. The searching

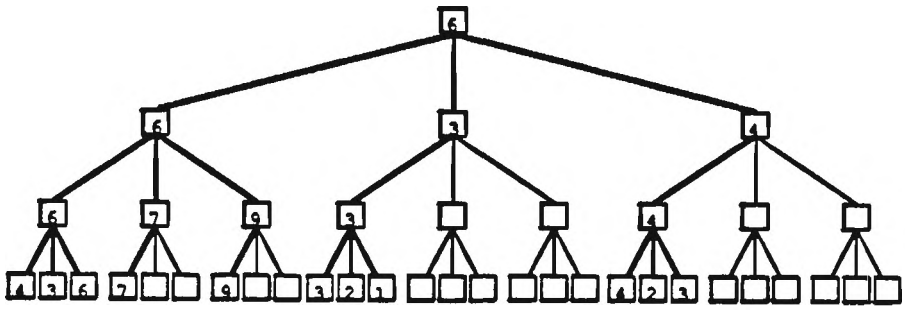


Figure A.1. Alpha Beta Search Tree Showing Minimum Branch Set

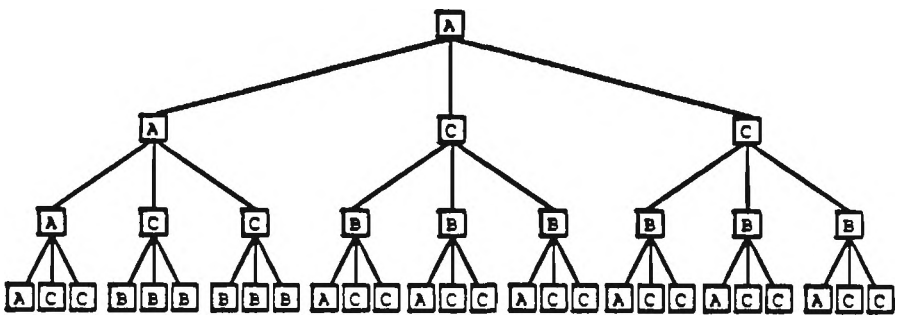


Figure A.2. Alpha Beta Search Tree Showing Node Types

behaviors determine the operation of processes in the parallel implementations. Once the process at a node is begun and it is determined not to be a leaf node, its child processes are evaluated according to the following rules:

1. All children of a type **A** or **B** node may be evaluated in parallel.

2. The tables used for pruning of the children of type **A** and **B** nodes may be accessed immediately after creation of the children.

3. The children of type **C** nodes must be evaluated sequentially.

4. The tables used for pruning of the children of type **C** nodes must not be accessed until the parents leftmost (type A) sibling completes. This enforces the use of later pruning information at type **C** nodes.

The evaluation rules stated above allow the 'minimum set' of branches to be evaluated in parallel. After the 'minimum set' of branches is searched, the remaining **B** siblings of each **C** parent are evaluated in sequence (there is still parallel evaluation between cousins). The evaluation of successive layers of the search tree follow the same rules.

The parallel programs outlined make use of several functions and variables that are not shown. These include:

1. **prune-it?**: This function returns true if searching at the present root node should be terminated. In the parallel programs the return value is the result value from the present node.

2. **static-evaluate**: This function evaluates the current board configuration.

3. **update-result**: This function updates the limit table for this node.

4. **static-eval-update**: This function evaluates the current board configuration and updates the current table.

5. ***maxply***: This constant indicates the maximum search level.

6. **make-new-table**: This function takes a current table and makes a new table to be used at subsequent ply levels.

7. **new-move-list**: This function takes the current board and makes a list of subsequent board positions.

8. **copy-value-if-applicable**: This function compares a parent-process table with a child-process table and updates the parent-process table as appropriate.

9. **copy-table-values**: Returns a copy of values in a table.

10. **make-semaphore**: This function creates a semaphore and initializes its count to zero.

A.4 Shared Memory Alpha Beta Search

This section discusses a parallel alpha beta search function **node-2**. **Node-2** explicitly creates processes using the process creation function **future**. Synchronization control between processes employs semaphores and shared variables. It is assumed that the reader is familiar with this form of parallel communication and synchronization control.

```
1 (define-process
2   (node-2 board myturn A-or-B-type parentleft ply parent-table
3           Done Left-sibling-done Parent-table-free)
4  (let ((result ()) (move-list ()) (num-moves())(quit())
5   (Left-offspring-done()) (mytable()) (New-table-free())
6        (cutoff()) (cnt()) (Offspring-done()) (My-table-free()))
7     (set! My-table-free (make-semaphore))
8     (set! Offspring-done (make-semaphore))
9     (set! Left-offspring-done (make-semaphore))
10    (set! mytable (copy-table parent-table Parent-table-free))
11    (set! New-table-free (make-semaphore))
12    (if (= ply *maxply*)
13      (set! result (static-eval-update board ply mytable))
14      (if A-or-B-type
15        (begin
16         (set! move-list (new-move-list board ply))
17         (set! num-moves (length move-list))
18         (do ((cnt 1 (+ cnt 1)))
19             ((eq? cnt num-moves))
20           (future
21              (node-2 (car move-list) (not myturn) (eq? cnt 1)
22                      left (+ ply 1) mytable Offspring-done
23                      Left-offspring-done New-table-free))
24           (set! move-list (cdr move-list)))
25         (do ((cnt 1 (+ cnt 1)))
26             ((eq? cnt num-moves))
27           (U Offspring-done)))
28        (do ((cnt 1 (+ cnt 1)))
29             ((or cutoff (eq? cnt num-moves)))
30          (future
31            (node-2 (car move-list) (not myturn) (eq? cnt 1)
32                    left (+ ply 1) mytable Offspring-done
33                    Left-offspring-done New-table-free))
```

```
34          (set! move-lst (cdr move-list))
35          (U Offspring-done)
36          (U Left-sibling-done)
37          (V Left-sibling-done)
38          (if (and (odd? ply) (<= (get-result mytable)
39                                  (get-result parent-table)))
40            (set! cutoff #t)
41            (if (and (even? ply) (>= (get-result mytable)
42                                     (get-result parent-table)))
43                (set! cutoff #t))))))
44    (result-to-parent parent-table my-table Parent-table-free)
45    (if (and A-or-B-type parentleft) (V Left-sibling-done))
46    (V done)))


47(define (result-to-parent parent-table my-table Parent-table-free)
48  (U Parent-table-free)
49  (let ((result (copy-value-if-applicable my-table parent-table)))
50      (V Parent-table-free)
51      result))


52 (define (copy-table parent-table Parent-table-free)
53   (U Parent-table-free)
54   (let ((result (copy-table-values parent-table)))
55     (V Parent-table-free)
56     result))
```

If the node process is type **A** or **B** it creates all of its children immediately (see lines 15 thru 24). It then enters a loop (lines 25 thru 27) where it repeatedly blocks decrementing semaphores until all of the children are completed. If the process is type **C** it creates its children and allows each to run in turn (lines 28 thru 43). Finally the process updates the parents table and enables potentially blocked sibling and parent processes by incrementing semaphore Left-sibling-done and Done (lines 44 thru 46).

A.5 DPOS Alpha Beta Search

This section outlines a parallel alpha beta search program using DPOS. A discussion of the NM definitions and Process Object **anode** is included to illustrate

the use of message passing, delayed instantiation and blocking to control parallelism. Figure A.3 shows the NM definitions superimposed on an alpha beta search tree and figure A.4 shows the network module definitions.

The network tree structure is defined by the NM definitions **atree**, **clst** and **blst** in figure A.4. **Atree** defines the children of type **A** and **B** nodes. **Clst** defines the siblings of type **A** nodes. **Blst** defines the children of type **C** nodes. Channels between **anode** type POs are used to communicate board configurations, pruning tables and to return final results to parent processes. Delay channels and blocking are used to manage the parallel evaluation.

```
1(define (anode PARENT PARBOARD CHILD SIBLING CH-SIBLING ply ntype)
2 (let ((board-set ())(board ())(move-list ())(prune-result ())
3       (old-table ()) (result ()) (mytable ())
4       (num-moves ()) (nodecount()) (ch-result())
5       (TABLE-CH (if (c? ntype) (table2 PARENT) (table1 PARENT))))
6   (set! board-set (PARBOARD 'receive 1))
7   (set! board (car board-set))
8   (if (not (b? ntype)) (PROPOGATE SIBLING (cdr board-set)))
9   (if (= ply *maxply*)
10    (begin
11     (set! ch-result (static-evaluate board ply))
12     (set! mytable (TABLE-CH 'receive)))
13    (begin
14     (set! move-list (new-move-list board ply))
15     (set! num-moves (length move-list))
16     (CH-SIBLING 'send move-list)
17     (set! old-table (TABLE-CH (if (c? ntype) 'read 'receive)))
18     ((table1 CHILD) 'send (make-new-table num-moves old-table))
19     (set! ch-result ((CHILD-RESULT CHILD) 'receive))
20     (set! mytable (if (c? ntype)(TABLE-CH 'receive) old-table))))
21  (set! result (update-result ply mytable ch-result board))
22  (set! nodecount (get-nodecount mytable))
23  (set! prune-result (prune-it? ply mytable ch-result))
24  (cond (prune-result ((CHILD-RESULT PARENT) 'send prune-result))
25        ((= nodecount 1) ((CHILD-RESULT PARENT) 'send result))
26        (#t (((if (not (b? ntype)) table2 table1) PARENT)
27             'send (add-nodecount (- nodecount 1) result))))
28  (if (b? node_type) (PROPOGATE SIBLING (cdr board-set)))))
```

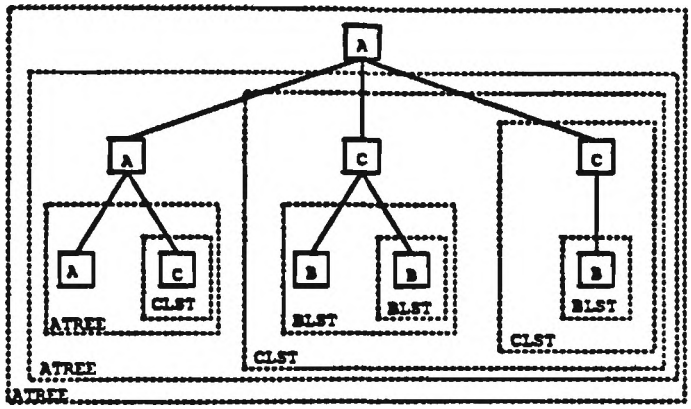


Figure A.3. Alpha Beta Search Tree Showing Superimposed Network Modules

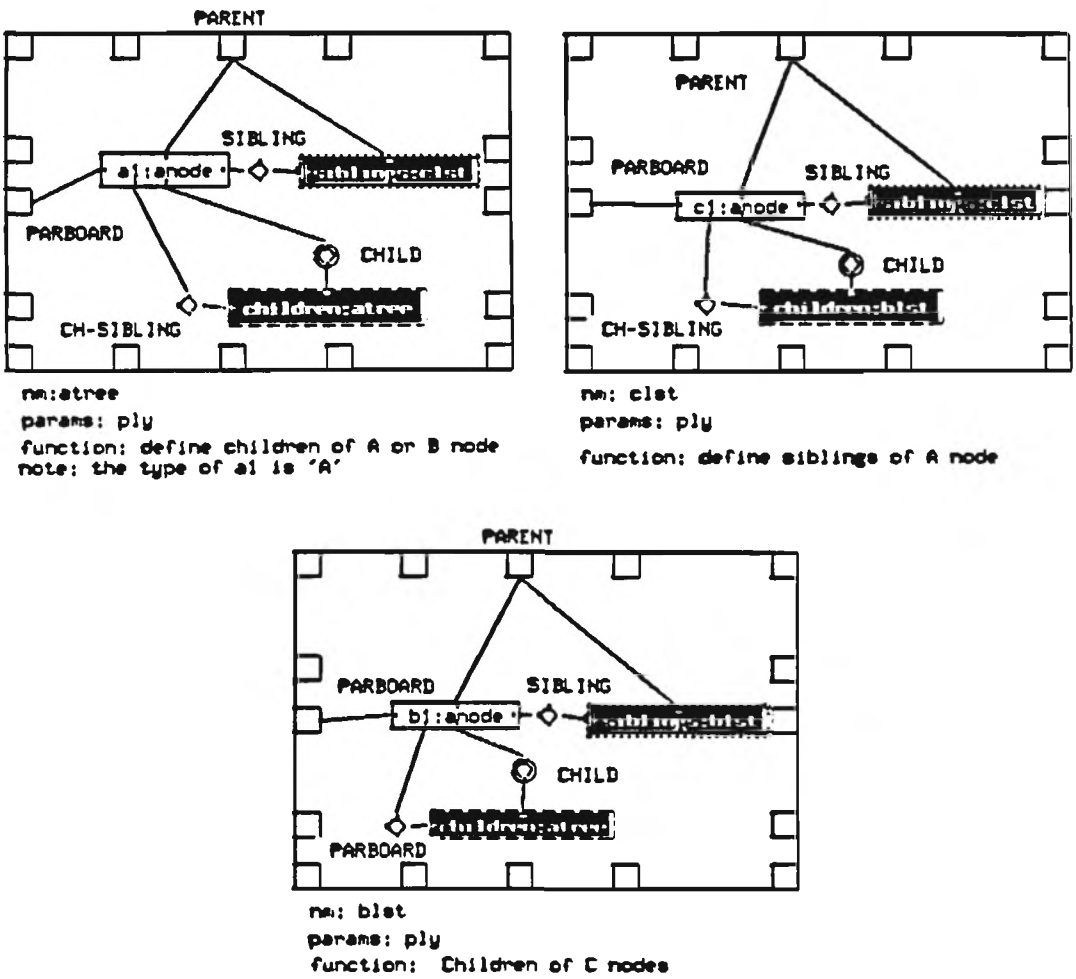


Figure A.4. Alpha Beta Search Network Module Definitions

```
29 (define CHILD-RESULT h-strm)

30 (define (PROPOGATE SIB mess) (if mess (SIB 'send mess)))

31 (define (table1 str) (h-strm (t-strm str)))
```

A.5.1 Delayed Instantiation of ANODE Process Objects

The SIBLING channel of each anode PO is used to input board configurations. Board configurations are propagated from the leftmost to the rightmost sibling (see Figure A.4.) The SIBLING channel is a delay channel. A child process will propagate the tail of its board set only if there are remaining configurations. This limits the instantiation of the child processes to the number of board configurations. Also, since **B** type processes are to execute in sequence rather than in parallel, **B** type processes do not propagate their board set until after they have completed computation, thus insuring that type **B** siblings proceed in sequence (see lines 8 and 28).

anode type	receives from table	sends to table
A	table1	table2
B	table1	table1
C	table2	table2

Figure A.5. Table Channel Usage

A.5.2 Blocking of ANODE Process Objects

The PARENT channel of each anode PO is a stream of channels (see Figure A.4.) Three elements of the stream are instantiated. These are called **table1**, **table2**, and **child-done**. Channels **table1** and **table2** are used to communicate table information. **Child-done** is used to return results to the parent node. The parent process blocks receiving from **child-done** until a child sends the final result

of all children to the channel. **Table1** is initialized by the parent process. Figure A.5 shows the pattern of usage of **table1** and **table2** by child processes. Type **C** processes receive from **table2** rather than **table1**. For any type **C** process, **Table2** will be initialized by the leftmost type **A** sibling, thus insuring that rule 4 is satisfied.

A.6 Comparison and Conclusions

The sequential alpha beta search algorithm is a depth-first pruning tree traversal. The parallel alpha beta search is a pruning tree traversal that combines aspects of depth-first and breadth-first search. This added complexity accounts in part for the increased length of the parallel programs.

In the DPOS program the tree network structure is encapsulated in the NM definitions. This encapsulation is reflected in the simplified structure of the DPOS Process Object definition **anode** when compared with the other function definitions presented. Both sequential function **node-1** and the shared memory function **node-2** define the tree traversal recursively. **Node-2** also uses loop constructs for control flow. The DPOS function **anode**, however, is reduced to a simple sequence of statements without looping or recursion.

In the shared memory function **node-2** critical sections are used to insure mutually exclusive accesses to shared data. In **node-2** semaphores are also used to regulate the progress of child processes, and explicit process creation is used to create parallelism. In the DPOS program, the management of parallelism is encapsulated within the NM definitions. This reduces the need for control mechanisms within the Process Object definition **anode**. The only parallel control mechanism used by **anode** is the sequence of access to its parameter channels.

The cost of nonlocal memory accessing is a primary concern in evaluating parallel program design. The shared memory alpha beta search function shown makes numerous accesses of shared data and semaphores and assumes that the cost of

these accesses is low. The critical section operations of the **copy-parent-table, result-to-parent** functions (lines 47 thru 56) of function **node-2** must be seen as detrimental to performance even on a semishared memory processor such as the **BBN Butterfly** where each semaphore and copy step involves nonlocal memory accessing. The DPOS program is more efficiently implementable on semishared and nonshared memory processors. The implementation of DPOS channels encapsulates the synchronization and copying operations within the channel. This encapsulation eliminates the need for repeated nonlocal memory accesses.

A.6.1 Conclusions

The visual editing system provides a simple means of designing parallel process networks. The encapsulation of network structure and many parallel issues within the DPOS system makes the creation and management of parallelism largely automatic. This encapsulation shows several improvements over more traditional methods of parallel programming.

1. Simplified control flow.

2. Simplified parallel control.

3. Reduced dependence on shared memory in program design.

APPENDIX B

EXAMPLE PROGRAMS

B.1 Introduction

This appendix includes three example programs. An example of the dining philosophers program is shown. A matrix multiplication program and a merge sort program are also shown along with corresponding performance statistics. Performance statistics are also shown for the prime number sieve program in Chapter 7.

B.2 Example Programs

Figures B.1, B.2 and B.3 show the a DPOS implementation of the dining philosophers problem. The dining philosophers program shows the use of guarded output channels to control nondetermine contention for access to **fork** process.

Figures B.4 and B.5 show a DPOS implementation of a matrix multiplication program. The matrix multiplication program is a network of server processes. The network is reused for successive matrix multiplications. The network receieves and multiplies streams of matrices. Right hand matrices are converted into columnar representation by **matb** Process Object. A list (**multipliers** Network Module) of vector multipliers (**row-mul** Process Objects) multiplies the matrices. **Resultant** Process Object collects and outputs the results. A sequential single processor version used for comparison is shown in Figure B.6. Performance statistics are shown for the multiplication program on the BBN butterfly using Butterfly Scheme. Running times are shown for the parallel program and for the single processor version.

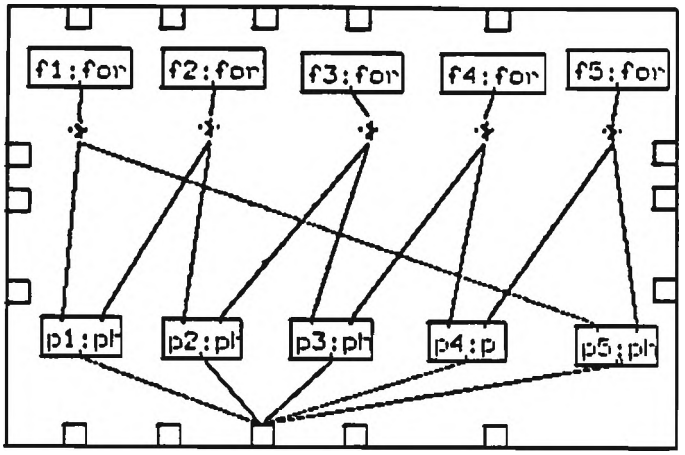


Figure B.1. Dining Philosophers Network Module

Figures B.7, B.8 and B.9 show Network Module and Process Object definitions for a Split Merge Sorting program[3,4,7]. Performance statistics are shown for Split Merge Sort compared with Quick Sort on a single processor node.

B.3 Performance Statistics

Figures B.10, B.11, B.12 show performance statistics for several programs presented in this thesis. The statistics presented are for DPOS programs implemented on the BBN Butterfly. Sequential programs were developed as a testbed for comparison purposes. Sequential program performance statistics are indicated by '*'. A minimum of three tests were run for each measurement. Several statistical measures were used:

1. Average Time: The average real running time of the program.
2. P: The number of BBN processor nodes used to execute the program.
3. Speedup: Sequential-run-time / Parallel-run-time
4. Efficiency: Speedup / Number-of-processors
5. Linear-percent: The ratio of the actual time saved to the maximum time saved if efficiency were 1.0.

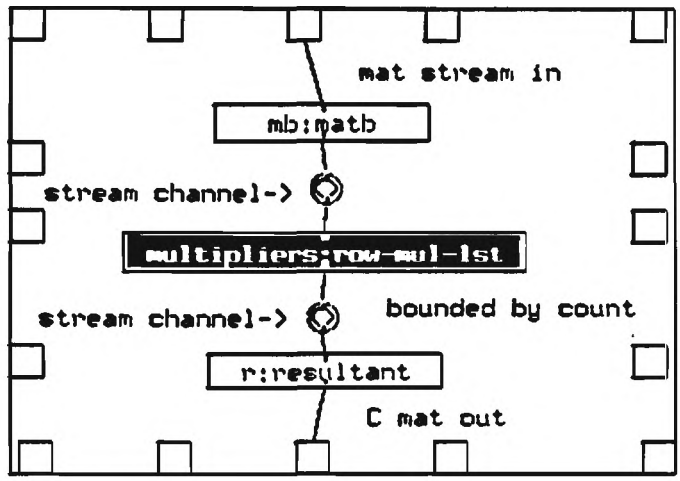
```
;; message types
(define grab 0)  (define transmit 1)  (define release 2)

(define (ph  OUTPUT  LEFTFORK  RIGHTFORK  id )
 (letrec
  ((guardlist (list transmit))
   (max 10)
   (self
    (lambda (cnt)
      (if (< cnt max)
        (begin
          ;;first grab left and right forks
          (LEFTFORK 'writeguard grab #f)
          (RIGHTFORK 'writeguard grab #f)
          ; now do any interaction desired
          (OUTPUT 'send
            (list 'phil id
                  (+ (cadr (LEFTFORK 'readguard guardlist))
                     (cadr (RIGHTFORK 'readguard guardlist)))))
            ;now release forks
            (LEFTFORK 'writeguard release #f)
            (RIGHTFORK 'writeguard release #f)
            (self (+ cnt 1)))))))
   (self 0)))
```

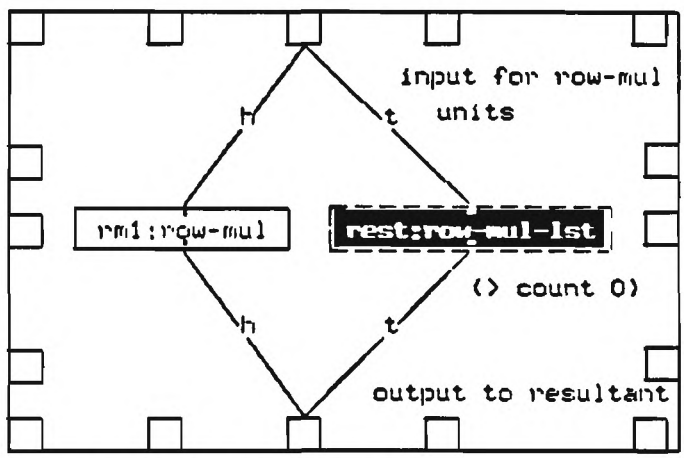
Figure B.2. Philosopher Process Object

```
; the fork Process Object
(define (fork   PHILOSOPHER )
 (letrec (
   (count 0)
   (self
    (lambda ()
      (begin
           ;wait to be grabbed
        (PHILOSOPHER 'readguard (list grab))
        ;after being grabbed  interact with grabber only
        (PHILOSOPHER 'writeguard transmit count)
        ;then wait to be released
        (PHILOSOPHER 'readguard (list release))
        (set! count (+ count 1))
        (self)))))
   (self)))
```

Figure B.3. Fork Process Object



network module:matmul
parameters: count



newtork module:row-mul-list
parameter: count

Figure B.4. Matrix Multiplication Network Modules

```
;; Process Object definition for 'row-mul' vector multipliers
;; receive a vector 'a-row' and a matrix (row by row)
;; multiply vector by matrix rows and send result
;; to TO-RESULTANT

(define (row-mul FROM-MATB TO-RESULTANT idnum)
  (let* ((a-row (FROM-MATB 'receive 1))
     (xdim (car a-row))
     (avec (caddr a-row))
     (mat-row (FROM-MATB 'receive 1))
     (zdim (cadr bmat)))
   (TO-RESULTANT 'send (cons xdim (mat-multiply bmat avec zdim)))
   (row-mul FROM-MATB TO-RESULTANT idnum)))

;; receive rows of the matrix to multiply by row-vec

(define (mat-multiply mat-row row-vec z  FROM-MATB)
  (if mat-row
    (mat-multiply
       (FROM-MATB 'receive 1)
       (vector-multiply row-vec mat-row)
        FROM-MATB)
    (list z row-vec)))
```

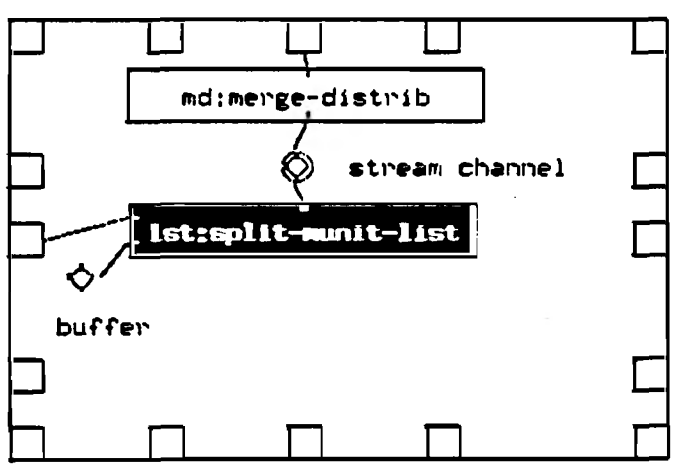
Figure B.5. Matrix Multiplication Process Objects

```
;; Sequential matrix multiplication program
(define (matrix-multiplier matlst)
   (matrix-multiply-list (car matlst) (cdr matlst)))

(define (matrix-multiply-list rowmat matlst)
  (if matlst
     (matrix-multiply-list
       (mat-multiply (make-vector (length rowmat))
                     (length rowmat)
                     rowmat
                     (convert-to-columns (car matlst)))
       (cdr matlst))
     rowmat))

(define (mat-multiply result-mat place rowmat column-mat)
  (if (>= place 0)
    (begin
      (vector-set! result-mat place
                   (vector-multiply (vector-ref rowmat place)
                                    column-mat))
      (mat-multiply result-mat (+ place 1) rowmat column-mat))
    result-mat))
```

Figure B.6. Sequential Matrix Multiplier



network-module: split-merge
parameters: list-size
description: sorting network

Figure B.7. Split Merge Network Module

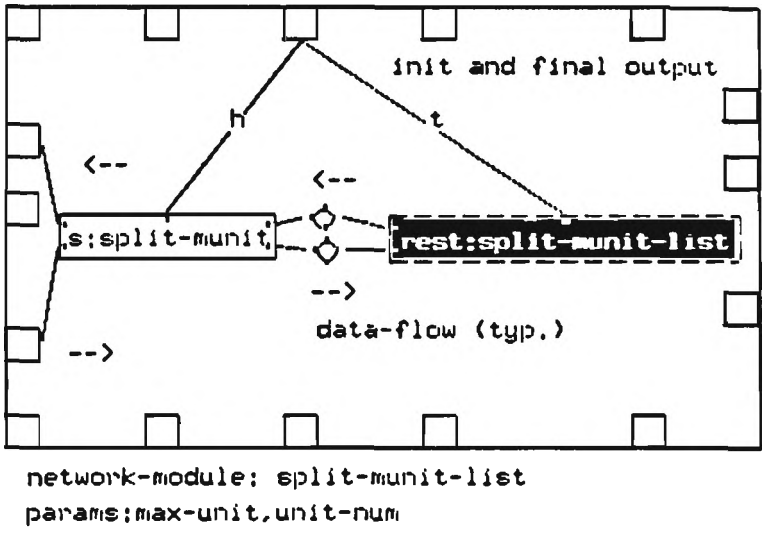


Figure B.8. Split Merge Unit List (split-munit-list) Network Module

```
(define
  (split-munit
     CONTROL LEFTOUT LETin RIGHTout RIGHTin max-unit unit-num)
  (letrec
    ((merge-iterations (- max-unit 1))
     (odd-unit (odd? unit-num))
     (self (lambda() (begin (merge-once) (self))))
     (last-unit (- max-unit 1))
     (merge-iter
      (lambda (cnt siza lsta sizb lstb)
        (if (< cnt merge-iterations)
          (let* ((res (merge-split lsta lstb siza 0))
                 (lst2 (merge-all (cadr res) (caddr res)))
                 (lst1 (car res)) (lst4 #f) (lst3 #f))
           (begin
             (if (> unit-num 0) (LEFTOUT 'send lst1))
             (if (< unit-num last-unit)
               (let ((res2 (merge-split lst2
                             (RIGHTin 'receive 1) sizb 0)))
               (begin
                 (set! lst4 (merge-all (cadr res2) (caddr res2)))
                 (set! lst3 (car res2))
                 (RIGHTout 'send lst4)))
                 (set! lst3 lst2))
             (merge-iter (+ cnt 1) siza
                (if (> unit-num 0) (LETin 'receive 1) lst1)
                sizb lst3)))
          (if (< unit-num merge-iterations)
            (last-merge lsta lstb RIGHTin)
            (merge-all lsta lstb)))))
     (merge-once
      (lambda ()
       (let* ((pair-of-lists (CONTROL 'receive 1))
              (id (car pair-of-lists))
              (len1 (car(cadr  pair-of-lists)))
              (lst2 (quick-sort (cadr(caddr pair-of-lists)) #f))
              (lst1 (quick-sort (cadr(cadr pair-of-lists)) #f))
              (len2 (car(caddr  pair-of-lists))))
        (LEFTOUT 'send
            (let ((res (merge-iter 0 len1 lst1 len2 lst2)))
              (if (< unit-num 1) (list id res) res)))))))
   (self)))
```

Figure B.9. Split Merge Sort Process Object Definition

Average Time	Processors	Speedup	Efficiency	linear-percent
196.000	1	-	1.00	1.000
62.604	4	3.118	0.78	0.906
36.400	8	5.385	0.67	0.930

Figure B.10. Matrix Multiplication Statistics

Average Time	Processors	Speedup	Efficiency	linear-percent
155.35	1	-	1.0	1.0
13.65	12	11.38	0.948	0.995

Figure B.11. Prime Number Program Statistics

Average Time	Processors	Speedup	Efficiency	linear-percent
58.33	1	-	1.00	1.0
19.10	4	3.05	0.76	0.897

Figure B.12. Split Merge Sorting Statistics

REFERENCES

[1] Abelson, H., Sussman, G. J., and Sussman, J. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[2] Agha, G., and Hewitt, C. *Concurrent Programming Using Actors. Computer Systems Series*, The MIT Press, 1987, pp. 37–53.

[3] Akl, S. G. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[4] Akl, S. G. *Parallel Sorting Algorithms.* Academic Press, 1985.

[5] America, P. *POOL-T: A Parallel Object-Oriented Language. Computer Systems Series*, The MIT Press, 1987, pp. 199–220.

[6] Babb, R. G., and DiNucci, D. C. *Design and Implementation of Parallel Programs. Scientific Computation Series*, The MIT Press, 1987, pp. 335–349.

[7] Baudet, G. M., and Stevenson, D. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers C-27*, 1 (January 1978), 84–87.

[8] Berztiss, A. T. Specification of visual representations of petri nets. In *Workshop on Visual Languages* (August 1987), pp. 225–233.

[9] Chandy, K. M., and Misra, J. *Parallel Program Design.* Addison-Wesley, 1988.

[10] Chapin, N. Flowcharting with the ansi standard: a tutorial. *ACM Computer Surveys 2(2)* (November 1970).

[11] Gehani, N. H. *Broadcasting Sequential Processes (BSP). International Computer Science Series*, T.J. Press, 1988, pp. 234–255.

[12] Glinert, E. P., and Tanimoto, S. L. Pict: an interactive graphical programming environment. *IEEE Computer 17*, 11 (November 1984), 7–25.

[13] Hoare, C. Communicating sequential processes. *Communications of the ACM* (August 1978), 666–677.

[14] Hoare, C. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[15] Jacob, R. J. K. A state transition diagram language for visual programming. *IEEE Computer 18*, 8 (August 1985), 51–59.

[16] Kerridge, J. *Occam Programming: A Practical Approach.* Blackwell Scientific Publications, 1987.

[17] Lieberman, H. *Concurrent Object-Oriented Programming in ACT 1. Computer Systems Series*, The MIT Press, 1987, pp. 9–36.

[18] Lusk, E. L., and Overbeek, R. A. *A Minimalist Approach to Portable, Parallel Programming. Scientific Computation Series*, The MIT Press, 1987, pp. 351–362.

[19] Melamed, B., and Morris, R. J. T. Visual simulation: the performance analysis workstation. *IEEE Computer 18*, 8 (August 1985), 87–94.

[20] Milner, R. A calculus of communicating systems. In *Lecture Notes in Computer Science No. 92*, Springer-Verlag, 1980. Lecture Notes in Computer Science No. 279.

[21] Muehle, E. Frobs user guide. March 1988. Utah PASS Project OpNote 87-05.

[22] Nelson, P. A., and Snyder, L. *Programming Paradigms for Nonshared Memory Parallel Computers. Scientific Computation Series*, The MIT Press, 1987, pp. 3–20.

[23] INMOS Limited. *Occam Programming Manual.* 1984.

[24] Raeder, G. A survey of current graphical programming techniques. *IEEE Computer 18*, 8 (August 1985), 11–25.

[25] Reiss, S. P. Working in the garden environment for conceptual programming. *IEEE Software 6*, 6 (November 1987), 16–27.

[26] BBN Advanced Computers Inc. *Butterfly Scheme Reference.* 1988.

[27] Shibayama, E., and Yonezawa, A. *Distributed Computing in ABCL/1. Computer Systems Series*, The MIT Press, 1987, pp. 91–128.

[28] Shu, N. C. *Visual Programming.* Van Nostrand Reinhold Company, 1988.

[29] Spencer, T. M. A visual interactive programming environment for the construction of block diagrams. In *University of Utah* (June 1989).

[30] Swanson, M. R., and Kessler, R. Domains-efficient mechanisms for specifying mutual exclusion and disciplined data sharing in a concurrent scheme. October 1988. Utah PASS Project OpNote 88-09.

[31] Yokote, Y., and Tokoro, M. *Concurrent Programming in Concurrent Smalltalk. Computer Systems Series*, The MIT Press, 1987, pp. 129–158.

[32] Yonezawa, A., and Tokoro, M. *Object-Oriented Concureent Programming.* The MIT Press, 1987.