An Abstract Implementation for A Generalized Data Flow Language

by

Kung-Seng Weng

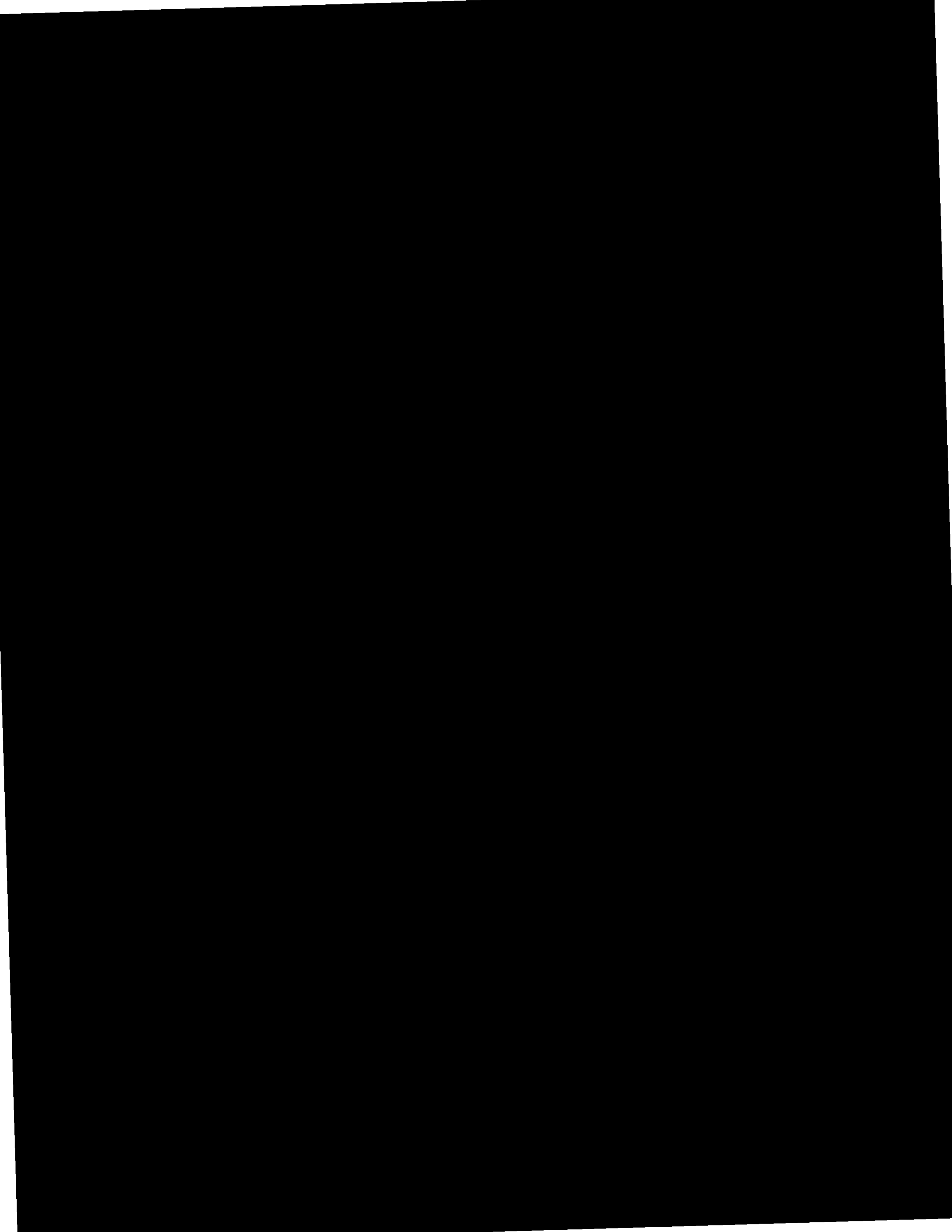
© Massachusetts Institute of Technology

February 1980

This research was supported by the National Science Foundation
under grant MCS-75-[illegible] A01

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge **Massachusetts 02139**



This empty page was substituted for a blank page in the original document.

This empty page was substituted for a blank page in the original document.

Acknowledgments

I am gratefully indebted to my thesis supervisor, Professor Jack Dennis, for his guidance when most needed and for the freedom of research throughout this work. Without his encouragement and patience, this document would not be possible. Thanks are due to the assistance provided by the thesis readers: Professor Peter Elias in the preparation of the thesis, and Professor Stephen Ward for many interesting discussions and suggestions for improvements.

The friendly environment of the Computation Structures Group has bred many ideas and provided warmth when the going is rough. I thank Clement Leung for being a critique on many drafts of the thesis and a friend throughout the years, and his wife, Enid, for innumerable warm meals. Sheldon and Sandy Borkin have generously offered me invaluable friendship in many ways. I wish to thank Dean Brock for providing me insights into nondeterminate computations, and for invitations to the comfortable home that his wife Ruth has made. I feel a sense of indebtedness to David Misunas with whom many nights were spent on discussions. Randy Bryant and Andy Boughton have been patient when we discussed technical subjects. Bill Ackerman has opened my interest in the Packet Memory.

I thank my wife, Michelle Hoshi, for her understanding during the difficult years of being a student's wife. I owe much to my brother Sam Weng for being a responsible son to my parents who have supported me both financially and emotionally.

Table of Contents

Abstract 2
Acknowledgments 3
Table of Contents 4

Chapter 1. Introduction 7
1.1. Concurrent systems 9
1.2. Concurrent programming languages 11
1.3. Data flow concept 13
1.4. Scope of the thesis 17
1.5. Synopsis 21

Chapter 2. Data Flow Schemas 23
2.1. Recursive data flow schemas 23
2.2. Well formed data flow schemas 29
2.3. Apply actors 29
2.4. Data structures 34
2.5. Discussion 43

Chapter 3. A Textual Language 45
3.1. A value-oriented language 45
3.2. Correspondence between the language and data flow schemas 51
3.3. Discussion 59

Chapter 4. Implementation of data flow schemas in a data flow processor 63
4.1. Data flow processor 63
4.2. Procedure structures and activation records 66
4.3. Procedure activations 77
4.4. Tail procedure applications 81
4.5. Discussion 85

Chapter 5. Streams, Nondeterminacy, and Forall 87
5.1. Streams 87
5.2. Implementation of streams 90
5.3. Forall 105
5.4. Nondeterminate merge of streams 114
5.5. Discussion 117

Chapter 6. Supporting Data Structures and Activation Records 123
6.1. Packet Memory 123
6.2. Activation records and holes 132
6.3. Remarks 136

Chapter 7. Conclusion 139

Bibliography 145
Appendix A 151
Appendix B 153

This page intentionally left blank.

Chapter 1. Introduction

In this thesis we are concerned with issues arising from the need to achieve concurrency of operation within a computation on a large scale. Several factors contribute toward increasing interest in systems capable of exploiting the concurrency of computation. Concurrency provides the potential for performance improvement through concurrent operation of hardware components such as processors and memory modules. This results in better utilization of total resources and in faster response if a computation has a high level of concurrency. The dramatic progress of technology has made concurrent systems more attractive as an alternative for high performance systems. In particular, systems that have many replicated hardware modules can take advantage of the projected potential of the processing capability of a single chip device which can be very economically produced. Such systems may further offer better fault-tolerance capability and extendability of system performance.

So far, concurrent programming has not been adequately dealt with in conventional programming languages. It is our belief that future systems must depart from the prevalent view of sequential computation both at the programming language level and at the machine organization level if a substantial progress is to be made toward practical large concurrent systems.

The goal of this thesis is to demonstrate that an adequate computation model can provide a basis both for a good programming language and for an architecture that can fully exploit the inherent concurrency in algorithms expressed in the language. To this end, we show how a value-oriented language can be implemented based on a model of concurrent computation known as *data flow schemas* [DenFo73] and how this implementation can guide the design of an architecture that achieves a high level of concurrent operations.

The model of computation is based on the notion of data driven computation, in the

sense that an operation in a computation is executed as soon as all of the required operands become available. Thus, there is no notion of sequential control of execution. Data flow schemas allow many concurrent subcomputations to take place without creating side-effects. The lack of side-effects is essential for several reasons. First, the existence of side-effects among concurrent processes may cause the outcome of the computation to be dependent on the order in which the processes are executed -- that is, the computation is nondeterminate. In most applications, it is desirable to achieve concurrent operation while preserving the uniqueness of the result of the computation. From the semantic point of view, a language that is free of side-effects is easily formalized using denotational semantics [Stoy77]. Furthermore, when a computation is expressed in a side-effect-free language, concurrency in the computation is easily recognized as subcomputations which do not depend on results of other subcomputations -- and this data dependency is manifest in the program structure.

We introduce a simple value-oriented language that has two important features: streams which are sequences of values communicated between computations, and <u>forall</u> constructs in which one can express concurrent operations on components of data structures. A computation expressed in this language is guaranteed determinate unless explicit forms of nondeterminacy are used. In this thesis, we consider a limited form of nondeterminacy that merges two sequences of values in a nondeterminate manner. We discuss limitations of the language in Section 1.4.

The architecture presented in this thesis is based on a form of data flow processor proposed by Dennis and Misunas [DenMi75, Misun78]. We show how the language can be effectively implemented on this architecture such that concurrency of a computation can be exploited. The main extension includes suggestions for the design of the storage of a large number of activations of procedures and data structures such that contentions in accessing data structures can be alleviated.

In the next two sections, we give a brief discussion of computer systems designed for achieving highly concurrent operations and programming language for expressing concurrent computations. Section 1.3 explains the data flow concept.

1.1 Concurrent Systems

Many computing systems [Kuck77, YauFu77, Enslo77] have departed from conventional computer organizations to improve the capability for concurrent execution. A class of such processors belong to the category of SIMD (Single Instruction Multiple Data) machines [Flynn72]. For instance, there are array processors represented by the ILLIAC IV [Barne68], associative processors like the STARAN [Batch74], and vector processors such as the CDC STAR 100 [Hintz72]. These processors perform well only when the computation can be expressed in program and data structures which are easily mapped onto the particular machine structures. Array processors require that data structures be mapped onto a fixed structure imposed by the physical arrangement of the processors, such as a two dimensional array. Associative processors require that data structures be linear lists of words so that associative operations on parts of these words can be efficient. For vector processors, data structures must be in the form of one-dimensional arrays to allow pipelining of operations on successive array elements. Furthermore, programs must exhibit a high degree of locality of reference such that a significant amount of data structure movement is not necessary during the execution. This dependence on locality of reference arises because the performance is achieved by short instruction execution delays and by special pipelined execution units or by many tightly synchronized independent execution units.

Unfortunately, the class of computations having these properties is rather limited; hence, much effort has been devoted to transforming programs -- either by the application programmer or by compilers -- so that efficient execution can be achieved [Lampo74, Kuck77].

In fact, even in the limited domain of numerical computations for which these processors are designed (or intended), there is a high degree of irregularity in computations so that these processors can not easily achieve their potential performance.[1]

The strong dependence on locality of reference and special features such as vector instructions inevitably tempts the programmers to be explicitly aware of the hardware features of the processors. This awareness often leads to programming errors due to concern with the optimization of programs. In this sense, these processors share the common problems that the programming issues are neglected and that the performance can neither be readily extended by introducing more execution units nor by moving from a processor of one configuration to another without a substantial amount of effort in program conversion.[2]

There are concurrent processors that belong to category of MIMD (Multiple Instruction Multiple Data) machines. A typical realization of this form of machines is based on multiple processor and shared multiple memory organization. Examples of such processors are Pluribus [Orns*75], C.mmp [WulBe72], and CM* [SwFuS77].[3] The predominant problem of these processors is that the system performance is based on the assumption of locality of reference achieved by programmers' explicit partitioning of a computation. Furthermore, because the semantics of the languages supported by these systems are based on the notion of sequential execution and operations which have side-effects, concurrency is achieved through careful analysis of programs to prevent possible deadlocks and bottlenecks in memory references.

---

1. We refer the reader to [KisRu75] for an example of how program mixtures have affected the performance of one of these processors. It is interesting to note that the CRAY computer [RamLi77] is designed with more recognition of this fact than previous vector computers by improving operations on vectors of short length.
2. Note, however, that the difficulty of transporting software among different systems is a pervasive problem of existing systems as well.
3. We refer readers to [Enslo77] for a more detailed discussion on machines based on multiple processor organizations.

1.2 Concurrent Programming Languages

Yet, what is a good concurrent programming language? There are two essential properties of a program: correctness and performance. The motivation behind *structured programming* is a consequence of the concern over the difficulty of establishing correctness of programs and of improving the productivity of the programming task. The task of concurrent programming, however, is much more difficult than that of sequential programming because the existence of concurrency makes any interaction between concurrent processes nontrivial. It should, therefore, be an essential design objective of a concurrent programming language to have the property that unnecessary programming difficulty is not introduced to improve the concurrency exhibited by programs.

There are several concepts which are unique to concurrent computations. In the execution of many concurrent processes, it is possible that the order in which the operations are performed affects the outcome of the computation. Such computations are said to be *nondeterminate.* Conversely, a computation whose result are guaranteed to be the same when the set of concurrent subprocesses are executed in any allowable order is said to be *determinate.*[1] Since many concurrently running processes may depend on the results of or synchronization by other processes, it is possible that a set of processes may become simultaneously dependent on the results of each other. If none of the processes can proceed further, then the set of processes are said to be in *deadlock.* Deadlocks occur in many forms depending on the possible situations which can arise to prevent a process from being able to proceed. The purest form of deadlock is that the computation itself can run into deadlock even if the amount of computational resources is infinite. In this case, what causes deadlocks is the semantics of the computation

1. A computation which contains nondeterminate subcomputations may itself be determinate. Thus, the class of computations expressible with operations which cannot introduce nondeterminacy is strictly contained by the class of determinate computations.

rather than the manner in which resources are allocated.

We now give a historical perspective of the problems of various approaches to concurrent programming, then outline in Section 1.3 an approach we feel may alleviate these problems and is followed in this thesis.

A natural development of concurrent programming has been to extend the existing semantic basis to include explicit process control primitives. An example is the introduction of call and wait primitives of PL/I which provide explicit control over the creation and resumption of processes. The coordination of concurrent processes is achieved by additional control primitives which interrupt and resume the control of an process with explicitly specified *signals* and with *conditions* which dictate when the control of a process may be influenced by signals from other processes. Another approach uses mechanisms such as *semaphores* and P and V primitives to coordinate these processes [Dijks68].

These forms of concurrent programming are at too low a level of abstraction to be good programming constructs in several ways.

It is often the case that a given computation when expressed in different sets of primitives results in quite different program structures. These differences arise not from the conceptual scheme of the computation but rather from the explicit control mechanisms that must be used.

Another consequence is that programmers tend to become very aware of the efficiency of the mechanisms. For instance, the cost of creating and controlling a process is often prohibitively high due to the inherent complexity of the semantics of these programming languages. The programming task is, therefore, further impeded because users often create processes with explicit concerns over resource management. (This, in a sense, is analogous to the situation when programmers had to be explicitly aware of the memory management in writing large programs before use of automatic memory management and because common

practice.)

In many situations, one finds that the computation is inherently determinate, but the program expressed in these forms is non-determinate in the presence of programming errors. Thus, there is no way to ensure determinacy when it is desirable. Tests or proofs for the program behavior are, therefore, unnecessarily complex, since the possible outcome of a computation is a set whose size depends greatly on the number of interacting processes. Furthermore, even in the presence of desired nondeterminacy, none of the individual subprograms can be validated independently. This deficiency for independent validation is attributable not only to the semantics of these primitives but also to the use of global variables that many concurrent processes can access and modify.

More recent approaches for concurrent programming emphasize the ease of validation of correctness for concurrent programs. Examples of language constructs using these approaches are monitors [Hoare74], path expressions [LauCa75], and guarded commands [Dijks75]. Note that these constructs are defined in conjunction with restricted use of variables and the flow of control. These represent steps toward a more structured and higher level of concurrent programming. A common feature of these approaches, however, is that concurrency is created explicitly with constructs such as the cobegin block or the guarded command blocks. Thus, the concurrency expressed is at the level of processes rather than at the level of operations where a substantial amount of concurrency also exists.

1.3 Data Flow Concept

Developments in the theory of parallel computation have motivated a computation model called *data flow schemas* [DenFo73]. This model is one of many models [Fosse72, Kosin73, ArvGo77] based on the data flow concept. The model represents a computation only in terms of data dependencies between instructions, and reveals inherent parallelism without unnecessary

constraints on instruction sequencing imposed by the conventional machine level representations.

1.3.1 Data Flow Languages

Because the data flow model is graphical in nature, numerous studies [Denns74, ArGoP77, Rumba75, Kosin73, Weng75] have attempted to define textual programming languages based on these models. While it is possible to define an algorithm that transforms programs written in existing sequential programming languages into data flow schemas, such an algorithm is complex because of the semantics of the sequential programming languages. Furthermore, the inherent concurrency of a computation is often hidden from the translator because there are additional constraints that are builtin in the expressiveness of sequential programming languages.[1] We believe that high level data flow programming languages will allow algorithms for concurrent computation to be easily expressible.

Programming languages based on the data flow concept are sufficiently expressive to encompass conventional programming language constructs such as iterations, while-loops, conditionals, procedures, and data types such as data structures and procedure values. These constructs, however, are embedded in a semantics which is free of both side-effects and the sequential control of execution. The distinctive lack of control transfer primitives such as GOTO's and operations which introduce side-effects allows compilers to easily detect data dependencies between operations in a program. Languages with these characteristics have been shown to have simple denotational formal semantics [Stoy74, Brock78].

---

1. This phenomenon is a well known fact among researchers working on optimizing compilers both for sequential processors and concurrent processors. For example, use of array indexing and common variables in large Fortran programs makes many optimizations difficult if not impossible.

Additional features such as forall constructs, primitives for stream values, and constructs for nondeterminate computations are found to be natural extensions to these languages. The forall constructs allow programmers to specify concurrent operations on all components of a data structure. The notion of stream provides an alternative to the use of coroutines and synchronization primitives for expressing computations passing sequences of values among their component modules.

A very important characteristic of these languages is that the determinacy of a computation is guaranteed when the computation is expressed not using primitives or features explicitly provided for situations where non-determinacy is required. In conventional languages, nondeterminate computations are expressed using semaphore primitives, call and wait primitives, and monitors. The semantics of these primitives, however, are not consistent with the semantics of data flow languages. Because there are significant applications where nondeterminacy is necessary, the formal semantics of languages with non-determinate primitives is still an active area of research [Plotk76, Milne79, Kelle77]. In this thesis, we have chosen a very primitive form of nondeterminacy which seems essential as a basis for higher level constructs for nondeterminate computations.

1.3.2 Data Flow Processors

Data flow schemas are not only a suitable vehicle for representing concurrent computations but also provide a simple operational semantics which has suggested several new computer architecture designs. Another characteristic of the model which is not often cited is that data flow graphs are very flexible bases for machine level representations. These representations are applicable to a wide class of computer architectures, including architectures extended from conventional processor and memory organizations.

The common characteristic of all data flow processors is the use of some machine level

representation of data flow graphs. Assuming that a data flow program already resides in a processor, its execution requires mechanisms for

(1) detection of conditions for an instruction to be executable,

(2) execution of the instruction, and

(3) transmission of the result to the instructions requiring it as an operand.

The processor proposed by Dennis and Misunas [Dennis75, Misun75] consists of five sections: Instruction Memory, Arbitration Memory, Functional Unit, Distribution Network, and Packet Memory. The Instruction Memory stores the machine level representation of a data flow graph so that enabled instructions can be independently detected and sent to the Arbitration Network as operation packets. The Arbitration and Distribution Networks are packet switching networks. The Functional Unit processes operation packets in a pipelined fashion. The Packet Memory performs data structure operations and memory management. The most distinguishing characteristic of the processor is that its performance is not derived from any assumption about the locality of allocations of the programs; thus, the program execution is not dependent on where each instruction resides. Different assumptions about the locality of computations result in great differences in the architectures of concurrent processors. While it is often the case that a computation exhibits locality of reference,[1] it has not been demonstrated that concurrent processors taking advantages of this fact are not subject to significant performance degradation when this assumption is violated by parts of a computation.

1. In particular, Swan has observed that references to the codes of procedures represent a large portion of memory references and exhibit high degree of locality of reference [Swan78].

1.4 Scope of the thesis

In this thesis we present an implementation of a programming language on an extended form of the Dennis-Misunas architecture. The extension includes storage of procedure activations, stream values, and data structures in the Packet Memory and we suggest a way to perform memory management for copies of data structures.

We chose a well defined programming language as the basis for extending the capability of the processor. This language has features which allow concurrency to be expressed in two forms and still guarantees that the computation is determinate and deadlock-free regardless of programming errors. The first form is based on procedure activations which automatically create concurrently executable procedure instances: this is the most familiar form of concurrency. The second form is based on the notion of stream computations (or, pipeline computations in some sense): this form of concurrency is frequently seen in large software such as compilers or in many operating system functions such as input/output which are often expressed either in the form of coroutines or in the form of coordinated processes [Conwa63, McIlr68, Hoare78].

Based on the notion of stream computations we provide a primitive for expressing nondeterminate computations by merging two streams of values in a first-in-first-out manner. Though this is a very primitive language construct, we feel it is an essential low level primitive for implementing other forms of nondeterminate constructs.

There are two ways in which some reader may consider our language limited. The language does not provide any construct for expressing a set of concurrent processes whose communication path forms a cyclic structure. This limitation is due to the general belief that the deadlock property for such a set of processes is not decidable in general at compile time. The second limitation is that we have not included procedures as values. This is because we have not found a satisfactory solution to its implementation. The data structures in the

language do not include any cyclic data structures such as doubly linked lists or cyclic graphs. The extension of the language to include such structures can be based on the notion of immutable objects which contain cyclic structures such that the semantics of the data structures is free of side-effects [Hende75]. This is an interesting issue of both practical and theoretical importance that we have not been able to scrutinize in depth in this thesis.

1.4.1 Related Work

The model of data flow computation proposed by Arvind and Gostelow [ArvGo77] is based on an interpreter that is quite different in philosophy from data flow schemas. The model does not introduce the notion of arcs of finite buffering on which data flow schemas are based, and results in an architecture different from the Dennis-Misunas architecture. Other data flow research include the DDMI model by Davis [Davis78], the model by Kosinski [Kosin73], the graph model introduced at UCLA [BaBoE70], the LAU system [SyCoH77], Gurd and Watson [GurWa77], and Treleaven [Trele77]. (This list is by no means complete.)

More recently, many workers have begun to interpret the meaning of the data flow concept within languages which have side-effect-free semantics such as Reduction language [Berkl75] and FFP systems [Backu78], and LISP based systems [FriWi76, KePaL78].[1] These languages have a different approach toward concurrent programming and may have interpreters whose operations are highly concurrent due to the side-effect-free nature of the languages.

In analyzing the structures of data flow processors, we can define two classes of processor organizations in the broad spectrum of possible structures that have been proposed.

---

1. The Actor semantics [Hewit76] based on the message passing style of programming also provide an basis for concurrent computation.

The first class consists of processors that have a large set of homogeneous modules connected by a network. Each module has a functional unit and a local memory, and all executable instructions are performed within the module. Processors of the second class do not have uniformly identical modules, and each module is specialized to perform a particular function, such as detection of executable instructions, execution of scalar operations, or switching of packets containing instructions or data. The types of networks for both classes range from simple bus structures to routing networks for handling packets of varying lengths. These networks are not intended for performing communications over a very long distance and therefore may not directly imply that the processors naturally extend to geographically distributed systems.[1]

The processors proposed by Davis, Syre, and Arvind and Gostelow can be considered to belong in the first class, and the second class is represented by the processor proposed by Dennis and Misunas.

Davis has proposed a hierarchical processor structure similar to a tree in which each processor module is allowed to communicate with its parent and a fixed number of child modules. Each module is capable of storing large segments of data flow programs and of partitioning a segment into subsegments which are sent to child processors as concurrently executable subcomputations. Because of its tree-like structure, this processor has the potential problem of unbalanced utilization of modules. The partitioning of a computation can also result in communication problems, since communication between child modules is made through parent modules. It has been proposed that these difficulties may be overcome by additional connections between leaf nodes of the tree.

---

1. The problems of detection and recovery from faulty communication links or processors, and those of resource managements are but some of the issues that are highly emphasized in distributed systems.

Syre proposes a bus-oriented network connecting a set of modules, each of which has a special control mechanism for detecting executable instructions. The allocation of processes to the modules is partly performed by the compiler that preprocesses each procedure by dividing it into segments for easy allocation of resources at run time. Some information needed for compile time allocation are supplied by the programmer in the high level language program. The bus network is adequate for connecting a limited number of modules, but is not extendable to a much larger numbers of modules.

Arvind and Gostelow prepose a ring network containing a number of ring interfaces each of which connects to a set of modules through a bus. Each set of these modules also share a memory controller which provides accesses and movements of data between a module and the large memory.

The main differences between our processor and these processors are the Packet Memory which is needed for general purpose computations and the assumption about the requirements of the networks. It is not clear how streams and forall's can be effectively implemented on these processors.

1.4.2 Hard problems

It remains untested whether programming languages based on the notion of data flow or the notion of side-effect-free semantics are applicable to designing counterparts of conventional operating system functions and to various techniques of heuristic programming found in the area of artificial intelligence.

For any system that is capable of creating a large number of concurrent activities, there are several inherently hard problems that need be solved. The most critical problem is the resource management which must not only be efficient for allocating a process but also provide mechanisms for controlling concurrent activities so the system is not overwhelmed by an

excessive number of activities. For systems that intend to support a wide range of applications, it may be necessary to provide mechanisms for aborting a computation which might never terminate or whose results are known not to be needed. For sequential computers, these mechanisms are supported by controlling the process states in the process queue of the system; but, for a highly concurrent system where activities may spread over a large number of hardware resources, it is not clear how these functions can be supported without degrading the performance promised by concurrency. For programming languages which can express computations that may result in deadlocks due to mutual data dependencies among processes, it is also necessary to have the above mentioned mechanisms.

It is important to realize that the limited nature of the scope of this thesis is due to our lack of understanding of the above problems and lack of simple solutions to them. It should be of great interest to readers to examine various proposed systems which exhibit a high degree of concurrency, whether they are based on the data flow concept or not, with awareness of these problems.

1.5 Synopsis

In Chapter 2, we present data flow schemas for completeness. This chapter also includes a short introduction to data structures. We have excluded data flow schemas which correspond to language constructs such as while-loops, and instead, we use recursions as an equivalent form of such constructs.

Chapter 3 introduces a simple programming language which is value-oriented. This language demonstrates that a clean programming language can be defined and translated into data flow schemas presented in Chapter 2. The procedure names in the language are globally defined. We include a discussion on issues related to extending the language for defining local procedures and handling procedure values.

Chapter 4 shows how an encoding of data flow schemas can be defined. We give a short introduction to the structure of the data flow processor and how the representation of encoded data flow schemas can be used to implement procedure activations.

Chapter 5 introduces the data flow operators that allow expression of stream computations in a natural manner. The straightforward implementation of streams based on these operators is very inefficient, therefore, we show an implementation of streams that is based on the notion of *holes*. We also introduce a primitive that nondeterminately merges two streams. We describes how several limited forms of forall constructs can be translated into recursive forms which exploit the concurrency in a natural manner.

Chapter 6 shows how resources for storing procedure activations can be allocated and supported. We show how simultaneous accesses to data structures can be handled in a multi-port and multi-cache memory organization while implementing *reference count* memory management.

Concluding remarks and directions of further research are in Chapter 7.

Chapter 2. Data Flow Schemas

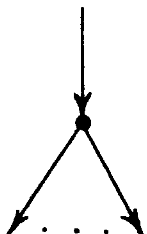
In this chapter we introduce an operational model for concurrent computation that has evolved from many similar graphical operational models used for studying the properties of concurrent computation. The earliest models were pioneered by Adams, Karp and Miller, and Rodriguez [Adams68, KarMi66, Rodri69]. These models were intended for investigating the decidability of properties of concurrent computations such as deadlocks, nondeterminacy, equivalences of program graphs, and comparative power for expressing concurrent computation.

Later works [Denns74, Kosin73, ArvGo77] are more oriented toward defining operational models as a basis for programming concurrent computations, and as a basis for investigating the degree of concurrency achievable. We are interested in the Data Flow Schema proposed by Dennis and Fosseen [DenFo73], because this model has evolved to the point that we are able to express naturally most language features found in existing high level programming languages. Furthermore, this model guarantees that computations expressed in the model are determinate while exhibiting a high degree of concurrency. We present a slightly modified version of the data flow schemas that does not have cyclic schemas and allows recursions.

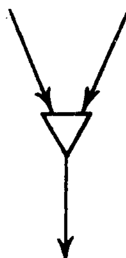
2.1 Recursive data flow schemas

The data flow schema is an operational model of computation and consists of a graph representation and an interpreter which operates on the representation. A data flow schema is a directed graph whose nodes are *actors* connected by directed arcs. An arc pointing to an actor is called an input arc of the actor, and an output arc is an arc emanating from the actor. Each actor has an ordered set of input arcs and output arcs. There are five types of actors: link, operator, switch, merge and sink. The five types of actors are shown in Figure 2.1. An (m, n)

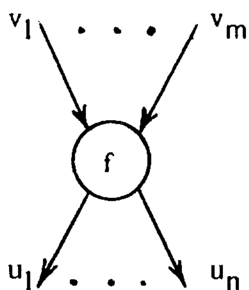
(1) link



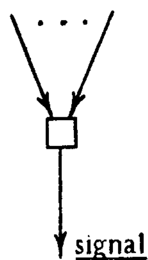
(4) merge



(2) operator



(5) sink



(3) switch

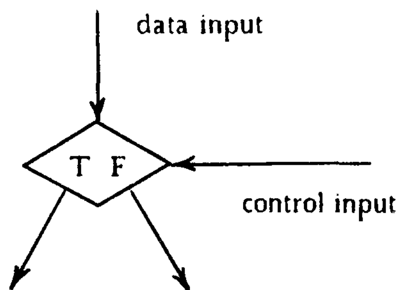


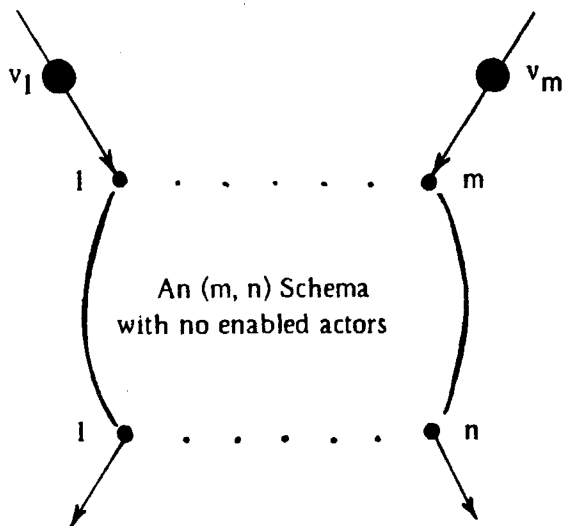
Figure 2.1. Data Flow Actors

data flow schema must have m link's which do not have input arcs, and n link's not having output arcs. These link's are respectively called internal input link's and internal output link's of the (m, n) schema. Further, we require that the schema must be proper in the sense that all other actors must have the arcs required of each type and each arc must be connected at both ends.

Description of the operational semantics of data flow schemas requires additional concepts: availability of data at the inputs and firing rules that define how a computation proceeds. A *configuration* of a data flow schema is the graph of the schema together with an assignment of labeled *tokens* to some arcs of the graph. An assignment of a token to an arc is represented by the presence of a solid disk on an arc. The label denotes the value carried by the token and may be omitted when the value is irrelevant to our presentation. Informally, the presence of a token on an arc means that a value is made available to the actor to which the arc points. In this thesis, we shall assume that the tokens carry values of types integer, real, boolean, or structure.

To describe a computation of an application of an (m, n) schema to some input values, we introduce the notion of *snapshots*: a snapshot consists of a configuration connected to a set of input and output arcs as shown in Figure 2.2. The computation of a data flow schema when applied to a set of input values is described by a sequence of snapshots. The initial snapshot consists of the graph shown in Figure 2.2 and an initial configuration which only has tokens on the input arcs as inputs to the computation. The computation advances from one snapshot to the next through the *firing* of some actor that is *enabled* in the previous snapshot. The condition under which an actor is enabled is depicted in Figure 2.3. It should be noted that a necessary condition for any actor to be enabled is that each output arc does not hold a token.

(a) Initial Snapshot



(b) Final Snapshot

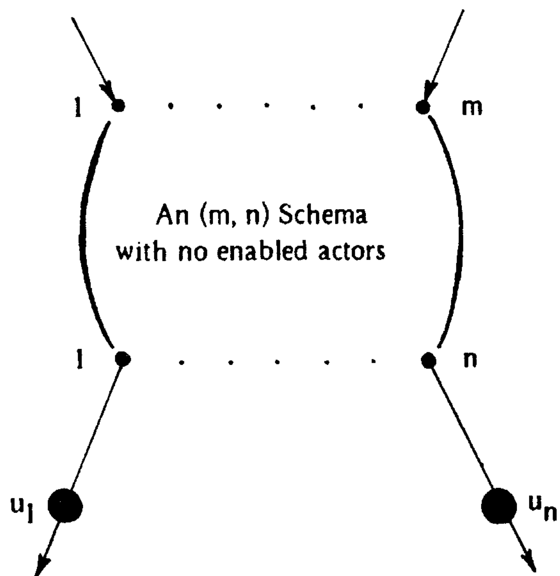
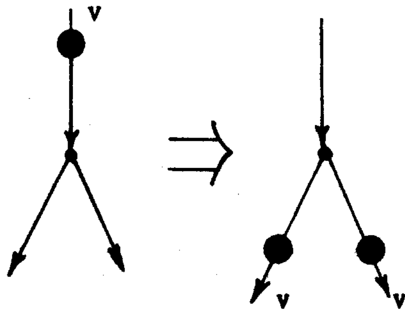
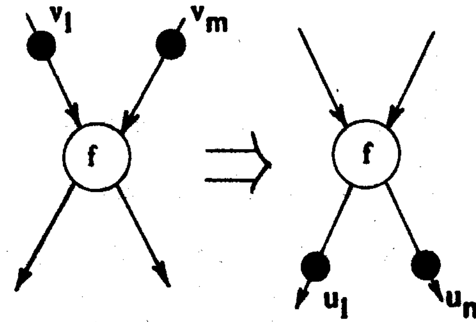
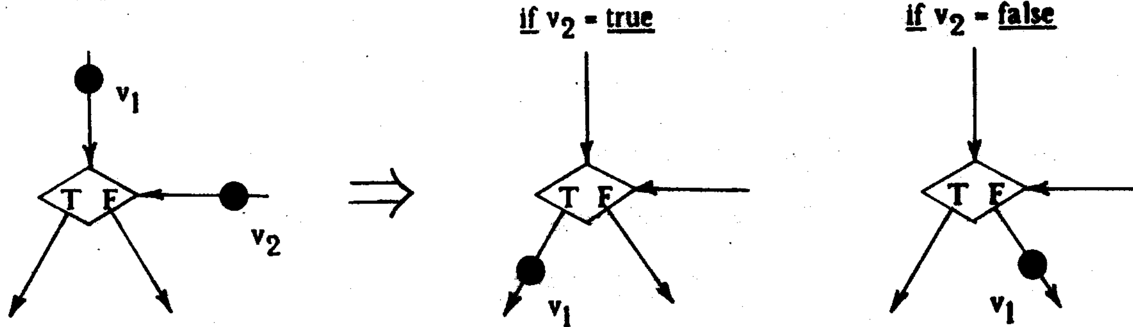
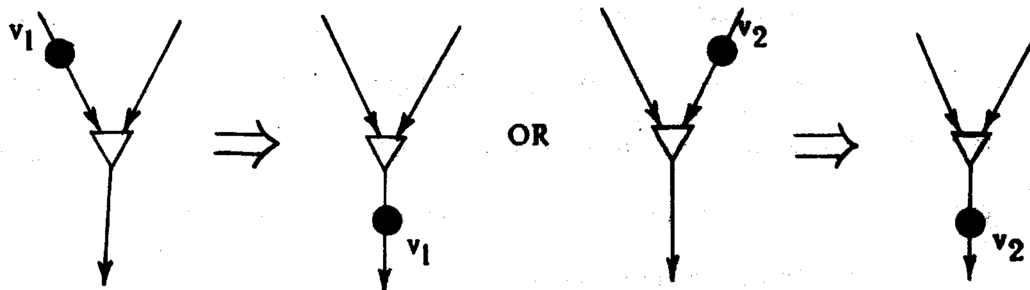


Figure 2.2. Snapshots









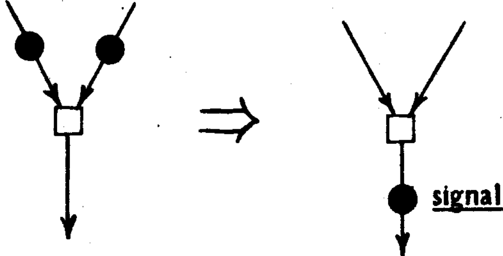


Figure 2.3. Firing Rules

Firing Rules

A typical actor is enabled by presence of a token on each input arc - with the exception of a merge. The firing of an actor absorbs tokens from its input arcs and places a token on each of the selected output arcs. The values of the output tokens are functionally related to the values of the input tokens. A link simply replicates the value received and distributes it to the destination actors - actors to which an output arc is connected. The effect of the firing of an operator is to apply to the inputs V1,...,Vm the function associated with the operation name written inside the operator to yield the outputs U1,...,Un. We generally require that labels be used to identify the type of the values carried by each arc, but will omit them when their types are clear from the context. The switch and merge are used for controlling the flow of tokens. A switch requires a data input and a control input of boolean value from the set {true, false}. The firing of a switch replicates the input token on one of the output arcs according to the boolean control value. The arrival of a token on either input arc enables a merge, and upon firing a token of the same value is placed upon the output. The behavior of a merge is inherently nondeterminate when two input tokens reside on the input arcs; neither token is lost, but the firing rule does not specify in which order the output tokens will be generated.[1] A sink absorbs the input tokens upon firing and places a special token signal on the output arc. The purpose of a sink actor is to absorb unwanted values; the signal output token is necessary for the implementation of schema application is described in Chapter 4. .

The set of functions commonly associated with an operator actor includes the scalar arithmetic operations and constant functions.

---

1. We choose the merge instead of the determinate merge of [DenFo73], because in recursive data flow schemas the chosen nondeterminate merge can safely replace the determinate merge and its use results in less complicated graphs.

2.2 Well formed data flow schemas

Unrestricted use of switch and merge is undesirable since arbitrary connection of these actors may form schemas which deadlock or are nondeterminate. Because these properties are undesirable for reliable programming, we choose a subclass of such schemas which will satisfy the needs of programming.

An (m, n) *well formed* data flow schema is an (m, n) data flow schema formed by any acyclic composition of component data flow schemas, where each component is either a link, a sink, an operator, or a *conditional subschema*. The structure of a conditional subschema is shown in Figure 2.4, where the heavily darkened arcs are labeled by letters denoting the number of arcs they represent. If P is an (m, n) subschema, Q is an (m, n) subschema and D is an (k, 1) subschema whose output is of type boolean, then the conditional subschema is an (m, n) subschema. Constructing a conditional schema from subschemas of different arity can be done by patching sink actors within each subschema.

2.3 Apply actors

The class of well formed data flow schemas as defined cannot express program features such as procedures, procedure applications, and iterations. We introduce an operator apply whose symbol is shown in Figure 2.5. The first input to an apply actor is a token carrying a name uniquely associated with an (m, n) well formed data flow schema which may also contain apply actors. An apply actor is enabled when a token resides on each input arc.[1] The effect of firing an apply operator is to modify the snapshot by replacing the actor with the (m, n) schema

1. This enabling condition is actually a very restricted form of procedure application, and does not satisfy some requirements of models which have the property of referential transparency [Stoy77]. Furthermore, this form of firing rule reduces asynchrony of the computation. We will discuss this in greater detail in Section 2.5.

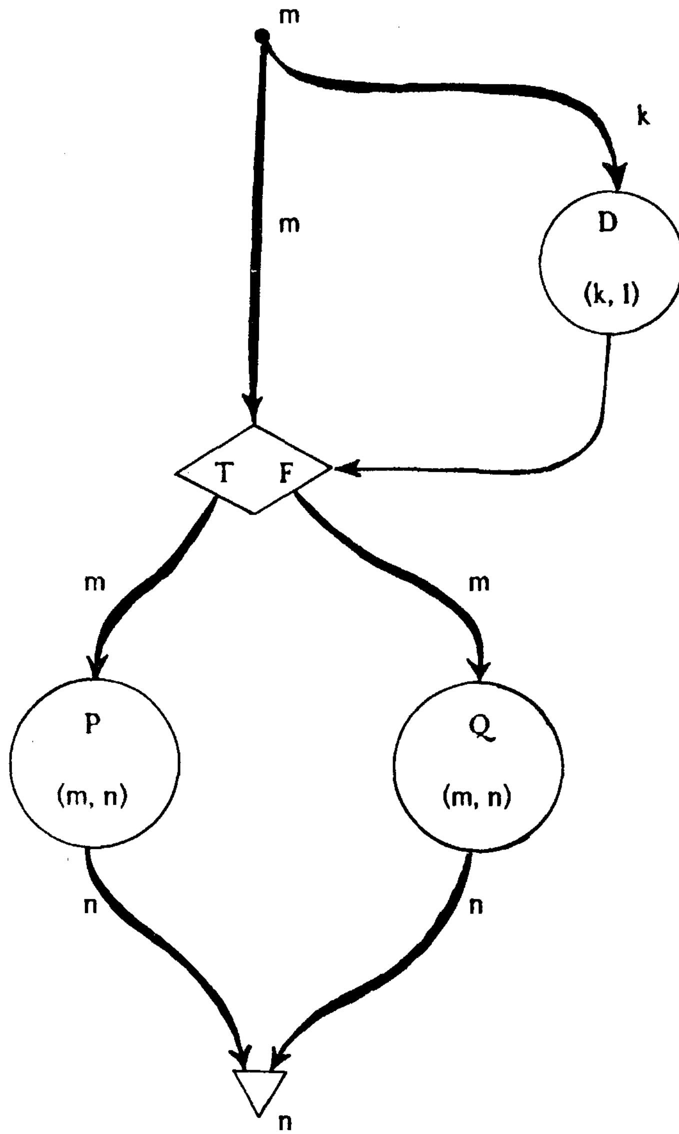
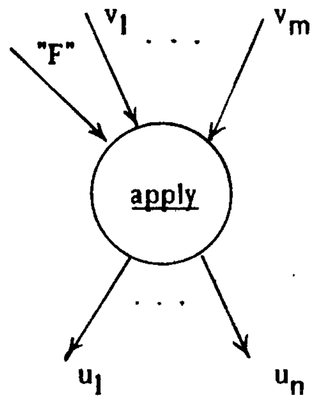
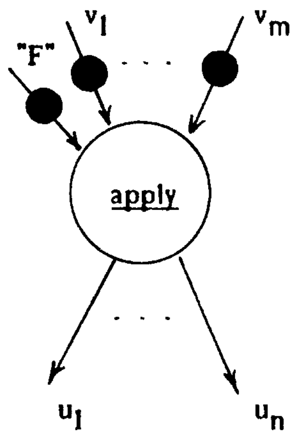


Figure 2.4. A Conditional Schema

(a) Notation for apply



(b) Firing Rule



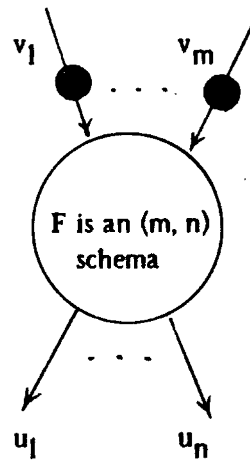


Figure 2.5. The Apply Actor

designated by the name as shown in Figure 2.5. This replacement connects the input arcs carrying values $V_1, \ldots, V_m$ to the m input link's of the schema and the n output link's to the output arcs $U_1, \ldots, U_n$ of the apply actor. Notice that the symbol of an apply operator allows one to define a data flow schema which involves recursive applications of the same schema by naming each data flow schemas. In this model, then, there is a global name space in which all schemas are defined a unique name.

An example of the use of apply actors is shown in Figure 2.6. It is a (3, 2) schema that is recursively defined, and computes the factorial of an integer greater than one. The first link actor labeled trigger is an input link whose function is to trigger constant actors for generating constants. The second link labeled f is for carrying the name of the procedure to the first input of the apply actor that uses the name to create another instance of the same schema. The merge actor labeled signal is to allow a proper construction of a conditional schema that may contain subschemas which uses sink actors. (Notice that the apply actor has a special output arc which carries a signal value. This is a convention that we have adopted and can be optimized in many situations.)

We have not included the class of data flow schemas which corresponds to language constructs such as while loops in Algol 60 or Do statements in Fortran or PL/I. Such data flow schemas [DenFo73] are constructed by cyclic connections of data flow actors, thus, the firing rule of actors that require the output arcs to be empty for their enabling must be observed. To implement this firing rule faithfully would require each actor to receive an acknowledge signal from each of its destination actors in addition to input tokens.[1] In addition, the merge actor must be a determinate merge actor [DenFo73] which requires a control input to determine which input tokens to be passed to the output arc. The use of acknowledge signals, however, can be

1. We refer the reader to [Misun75] for an example illustrating this point.

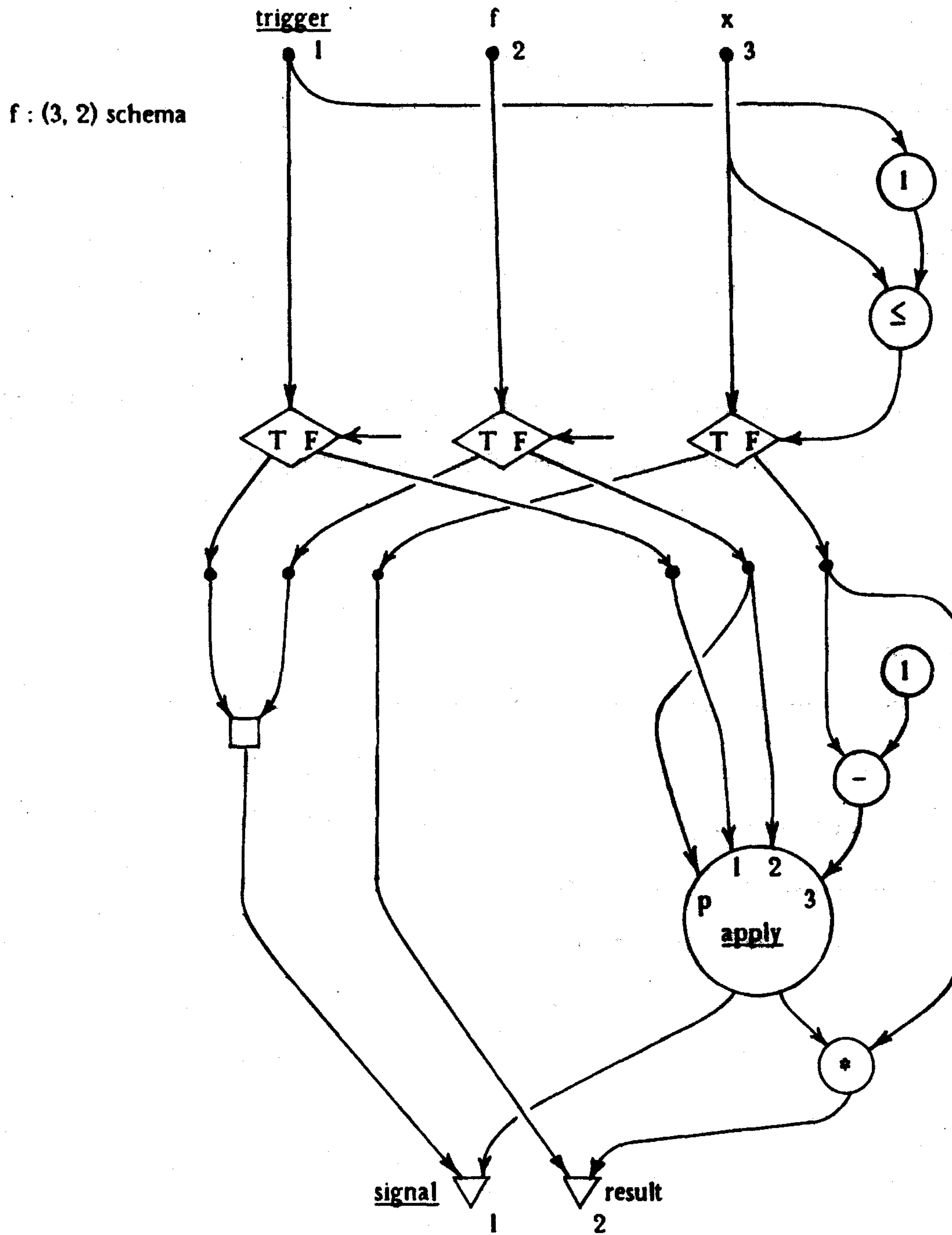


Figure 2.6. Recursive data flow schema for factorial
f(x) = if x =< 1 then x else x * f(x - 1) end

eliminated when the schemas are free of cyclic connections. This has the advantage that the firing of each actor is not delayed by waiting for acknowledge signals from its destinations. Furthermore, there is no need to encode into instructions the information required for returning acknowledge signals. This leads to a simpler mechanism for implementing procedure activations if the class of data flow schema is restricted only to acyclic schemas.

For these reasons, we choose to implement these language features in their equivalent form of self recursive application of data flow schemas. This has the desirable property that, without any compile time analysis, the mechanism for procedure activation allows simultaneous execution of different instances of the data flow schema which correspond to different iterations of a While-loop.

2.4 Data structures

In this thesis, we are interested in an interpretation of data flow schemas which requires the types integer, real, boolean, character_string[1] and structure. We will assume that the set of operations defined on the data types other than data structures is well understood. We now define notation for data structures and the set of allowable operations. (The material presented here is based on [Denns72, Ellis74].)

The strict definition of the semantics of data structures must include all data flow actors which have at least one input or output arc for carrying data structures. Thus, the set of actors would include link, switch, merge, sink, and operator. The function of switch and merge

---

1. We restrict ourselves to character_string of bounded length which can be treated as a scalar value. For character_string of variable length, the implementation will be quite different. Furthermore, if selector names of a data structure operation is of variable length, implementation of data structure operations depend on how variable length character_string is implemented.

is purely for controlling the flow of values and is naturally extendable to data structures. The function of create, append, select, link and sink determines the number of the instances of data structure values that exists in the system. These actors, therefore, are related to the function of resource management of storage for data structures. Semantically, the function of the link and sink actors are the same as defined previously. The primary type of actors that we define here will be the class of operators which perform operations on data structures.

A data structure can be either a nil structure which has no component or a structure having n component structures d1,...,dn whose *selector* names are respectively s1,...,sn as shown in Figure 2.7(a). The selectors are either character strings or integers and each selector name must be different from all others in the same data structure. Furthermore, these selectors are assumed to be ordered lexicographically. An alternative linear notation for the structure is

(s1 : d1, . . . , sn : dn).

The set of data structure operations are defined below, where d and d' are data structures, s is a selector name, and c is an object of any type:

(1) create ( )

The create operation causes a nil data structure to be returned as the result. (Figure 2.7(b)(1))

(2) append (d, s, c)

The operation returns a data structure d' which is identical to d except that the s component is c regardless of whether d already has a component with the selector name s. (Figure 2.7(b)(2))

(3) delete (d, s)

The result of the operation is a data structure d' which does not have an s component. (Figure 2.7(b)(3))

(4) select (d, s)

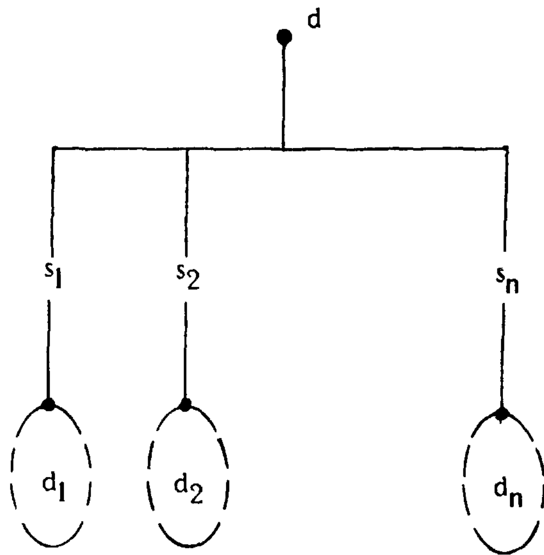
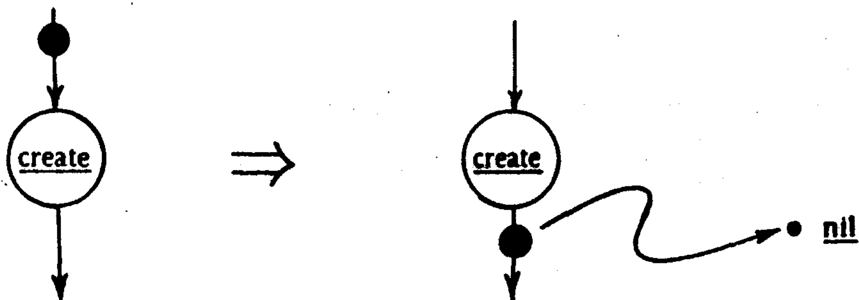
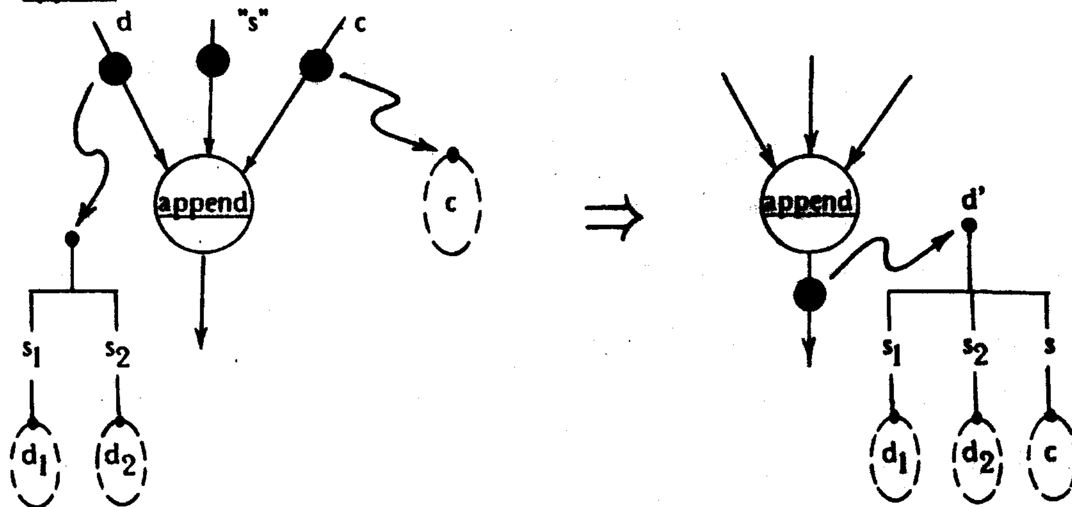


Figure 2.7(a). A Data Structure

(1) create



(2) append



(3) delete

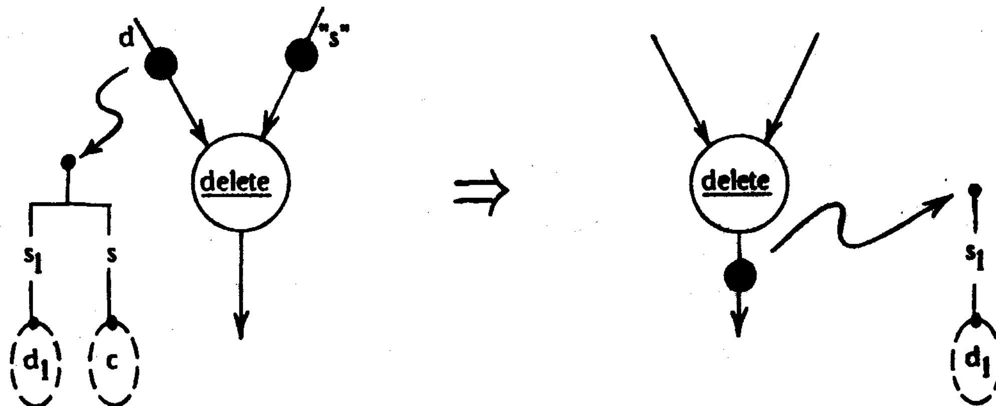


Figure 2.7(b) Effects of data structure operations

If d has an s component, the result is the object c associated with that component. Otherwise, the result is the value undefined. (Figure 2.7(b)(4))

(5) nil_structure (d)

This is a predicate whose value is true if d is nil, otherwise its value is false.

Examples of the effect of these operations are illustrated in Figure 2.7(c). Notice that the effects of

delete (d, s), and

append (d, s, nil)

are different, since the the delete operation would remove the tuple (s, d') while the append operation would replace it with (s, nil). In general, it is possible to distinguish between these two data structures using the select operation, since it returns the nil structure for one while returning undefined for the other. It should be mentioned that an array is simply a data structure whose selector names are all integers.

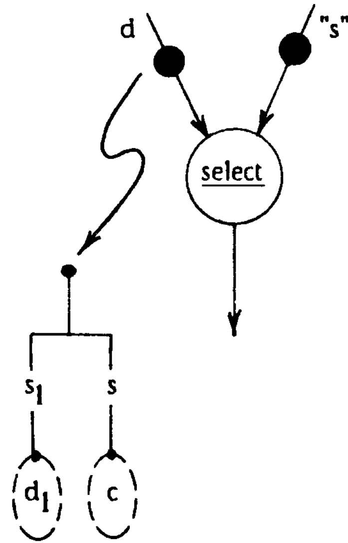
The set of operations together with the link actors and sink actors provides a complete set of operations on data structures.[1] These operations allows one to create dynamic data structures of arbitrary size as opposed to data structures which are declared to be of fixed structure and mapped into linear representations at the compile time. The function of storage allocation for the data structure operations is implicit in these operations, while conventional programming languages which allow this form of dynamic data structures often use explicit storage allocation primitives.

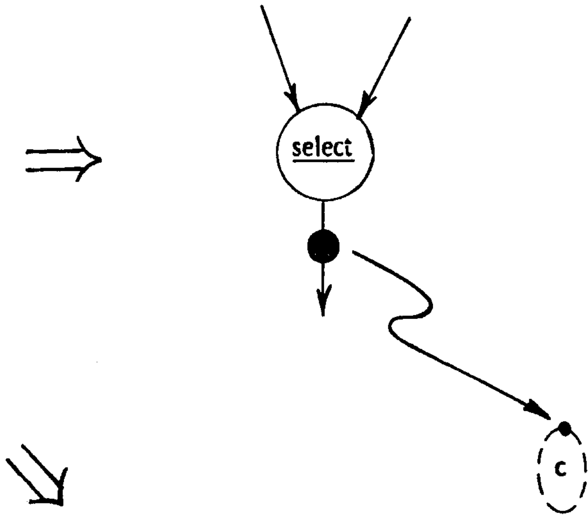
This form of data structures can represent sparse arrays in a very efficient manner. Since selector names can be character strings, it is possible to implement algorithms on data structures without having to explicitly encode the character strings into other forms such as

1. Complete in the sense that the set of data structures is closed under the operations.

(4) select

if c is a structure





if c is a scalar

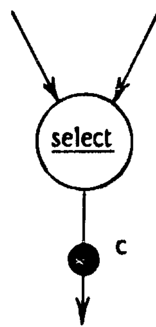


Figure 2.7(c) Effects of data structure operations

integers used as subscripts into an array. Thus, the user need not be aware of the particular structure of the internal representation. The semantics of the data structure operations defined above is free of side-effects, because a data structure operation always produces a new data structure without modifying data structures used in other parts of the schema. Thus, the computation is free of side-effects. We feel these are properties that ease programming tasks.

The implementation of data structures can be based on the notion of *items*. An item is a storage node that is associated with a unique identifier (uid) and can store a set of tuples of the form (s, c) where c is either a uid of another item or a simple scalar value and s is the selector of the component. Thus, a data structure is represented by a collection of items. In the definition of data structure operations (except the predicate nil structure), each operation on a data structure semantically creates a new data structure representing the result of the operation. In this implementation of data structures, the result of a select operation is either a scalar value or simply the uid of the selected component. The implementation of the append operation, however, must maintain the side-effect free property of these operations. The result of

append (α, s, λ)

is the uid of an item containing a new set of tuples which differs from α only in the tuple (s, λ). Using items, an efficient implementation can be defined [Denni75] since it only requires creating a new set of tuples and does not copy the entire sets of tuples of the subcomponents. Thus, the implementation allows many component structures to be shared physically while maintaining the side-effect free nature of the operations.

There are many implementation considerations that affect the efficiency of these data structure operations.

First, we must provide mechanisms for resource management. These mechanisms must allocate items and must determine when the storage can be reclaimed. The latter must be dependent on the behavior of the program and on how data structure operations may provide

additional information for the resource manager. In traditional systems using dynamic resource allocation and automatic resource management, this information is obtained by maintaining a root node from which all nodes accessible by the computation are traceable.

We choose a different approach to the garbage collection problem. This approach is possible only because the semantics of the data structure operations allows an implementation that always produces an acyclic structure of items. For each item we include a *reference count* which indicates how many references (instances of its uid in the system) to that item exist. Each data structure operation modifies the reference count of the items. The set of operations that affect the reference count must include all actors which carry tokens carrying data structure values; for example, the link actor which copies the uid of a data structure must increment the reference count of the item, and the sink actor must decrement its reference count. Thus, there is an overhead associated with each data structure operation for maintaining the reference counts.[1]

The other concern is that of the size of the node for storing the tuples. Since the allocation of a variable size node is quite difficult, we have only seen proposals that use fixed size nodes. This restriction raises the problem of how to represent a variable size node with fixed size nodes. An approach is to require that selectors have the property that each can be considered as a sequence of symbols from a fixed size alphabet. Then a variable node might be implemented as a tree of fixed size nodes such that each path from a root node to a leaf node represents a selector name. We refer readers to further readings [Rumba75, Acker77] on this

1. It has been argued that the overhead associated with reference count storage management scheme may be higher than that of garbage collection schemes on cyclic structures. This inefficiency argument against the reference count scheme is not valid when we adopt a scheme called *split reference count*: a uid to a data structure is conceptually a tuple (uid, reference_weight), a link of two output arcs that receives (α, n) fires by producing two tokens carrying (α, n_1) and (α, n_2) such that $n = n_1 + n_2$. We should mention that this is an alternative form of managing items and its feasibility needs further investigation.

subject.

Another important characteristic of the operations is that the form of data structures created using these operations is always an acyclic graph. This is quite different from conventional programming languages which allow one to create arbitrary structures constructed by manipulating pointers. We have explicitly disallowed such operations for several reasons.

The creation of cycles is a programming technique which has proved effective in sequential programming. It is not clear, however, that such techniques are suitable in a programming language which does not allow side-effects. The programming technique can indeed be *simulated* in a language not having cyclic structures by introducing procedures which interpret the acyclic counter part of the cyclic structure. It is desirable that we can provide a comparison of programming task of the two approaches. Unfortunately, we have neither seen nor found good cases against or for either approach. While we do know that semantics based on immutable cycles is a possible approach [Hende75], it remains to be shown that cycles are indeed an essential form of data structures.

The other reason for disallowing cycles is based on a resource management argument. For systems such as the LISP interpreter, the existence of cyclic data structures results in the need for garbage collection schemes which mark all of the accessible data structures and deallocate those that are left unmarked. This has the undesirable effect that a computation is interrupted during the process of garbage collection. Some recent works [Baker78, Bisho77] have reduced this effect by introducing garbage collection schemes which allow computations to be running concurrently during the garbage collection. In a system which does not create cyclic structures, the garbage collection scheme can be based on reference counts and need not resort to the elaborate schemes that have been developed.

In this thesis, we have restricted ourselves to acyclic data structures because the implementation of procedure activations and streams are orthogonal to this issue. Therefore,

we leave this as an area that can be investigated by others.

2.5 Discussion

The apply actor presented in Section 2.2 requires that all input values to be present on the input arcs to become enabled. This has two implications.

First, the language definable based on the apply actor must define its semantics based on "call by value", that is, a procedure (or, interchangeably, schema) application is well defined only for the case when the computations producing inputs to the procedure terminate. This can be contrasted with the more general form of procedure applications which allows a procedure application to take place even when the computation of some of the inputs does not terminate. The more general form of procedure application has a desirable semantic property which is often referred to as *referential transparency* or the property of *substitution* [Stoy77]. Let f and g be two procedures such that f appears as an application inside of g, and let g' be the procedure obtained from g by substituting the text of f in place of the application. In the language that is referentially transparent the specification of the functional for g will not depend on any specification of the termination property of f, thus, the functional for g and g' will be equivalent. In the language whose procedure applications must depend on the termination of the procedure f, the procedure g and g' would be of different functionals. This is because the substituted procedure allows the computation to proceed without waiting for all inputs to be available. The difficulty with designing a system which supports a referentially transparent language is that it needs mechanisms that detect when the result of a subcomputation is not required for further computation and prevent the nonterminating subcomputation from wasting computing resources.

The second implication is if the operation of the apply actor is implemented in a straightforward manner, the degree of synchrony of the computation is reduced. Because, in

most cases, there are parts of the computation that can proceed as soon as some of the input values become available and need not be constrained to wait for the arrival of other inputs. For a referentially transparent language, this asynchrony is achievable, while for the language with call-by-value semantics this asynchrony is constrained unless one knows that all computations terminate.

A consequence of side-effect-free data structure operations is that some operations which seems rather simple to perform in existing languages become more complicated. Consider a data structure A from which a data structure A' is to be constructed which is identical to A except for the component

select(select(A, "a"), "c").

To construct A', we need the following operations when no language level sugaring is provided:

append(A, "a", (

append(select(A, "a")), "c", C')).

Thus, from the criteria of ease of expression, some additional higher level operations need to be defined.

There are many issues that require further study to understand fully the implication of the side-effect free semantics. We have already touched briefly on the issues on cyclic structures. Another interesting issue relates to the computational complexity of many algorithms that have been found to be efficient but have not been shown to be equally efficient using side-effect free operations. Examples of such algorithms are heap sort and merge sort [AhHoU75]. Thus, the criteria for choosing appropriate algorithms for applications may be significantly different depending on whether modifications are allowed on existing data structures. Still another area is the semantics of nondeterminate computations.

Chapter 3. A Textual Language

In this chapter we introduce a programming language based on the model of data flow schemas described in Chapter 2. The language departs from conventional sequential languages in many ways. We have removed the notion of sequential control flow of a computation by introducing value-oriented semantics. There are no explicit language primitives for introducing parallelism. The concurrency of a computation is determined by the data dependency within the program rather than by explicit creation of concurrent processes. While it is possible that compile time analysis can be performed on sequential programs to produce an equivalent program of greater concurrency, this does not help programmers to express computations in a form which exhibits high level of concurrency. Furthermore, no compile time analysis has been able to extract the inherent concurrency from a program containing unnecessary constraints which are the result of language features based on the assumption of sequential computer organization.

The language does not have the notion of memory locations or variables commonly found in conventional sequential programming languages; instead we introduce the notion of naming for identifying a value in a computation in very much the same way mathematical notations would use names. With the value oriented semantics, we expect programs now can exhibit the inherent concurrency of an algorithm, and may even provide additional motivation for designing new algorithms of greater concurrency.

3.1 A value-oriented language

The language is value-oriented in the sense that each syntactic unit corresponds to a function whose evaluation produce a set of values. The computation associated with a syntactic unit called an *expression* does not interact with the computation of other expressions in the

program. While the purest form of value-oriented language does not use names for defining values, we introduce names for defining procedures and for convenience of programming since naming is a useful mechanism for identifying values of expressions.

In this thesis we will not be concerned about many language design issues that arise in making the syntax and the semantics of the language rich enough for a user to program in.[1] The language is intended only to demonstrate the existence of a reasonable syntax and to facilitate the discussions in later chapters. The set of data types consists of integer, real, boolean, character_string, and structure. We shall call these data types *simple data types*. The set of operations defined on integer, real, boolean, and character_string are the usual operations seen in many languages. The operations on structure are the set of data structure operations given in Chapter 2.

The syntax of the language is given in Figure 3.1. A procedure definition consists of a list of procedure definitions followed by an expression. A procedure definition is of the form:

P = procedure (a_1:T_1,..., a_m:T_m) yields (R_1,...,R_n)

... a list of procedure definitions ...

<expression>;

end P;

This defines a procedure P that requires m *input values* a_1,...,a_m of types T_1,...,T_m respectively. The names a_1,...,a_m must be distinct and can appear in <expression>. The evaluation of the

1. The language described here can be regarded as a subset of the language called VAL in development at MIT [AckDe78].

```
Notation :   { < E > }+  means  < E > | < E >, { < E > }+
             { < E > }  means  <empty> | { < E > }+ .

< program > ::=  program { < procedure def > } < expression > end
< procedure def > ::=  < name > = procedure ( < input list > )
                                  yields < output list >;
                                  { < procedure def > }; < expression >
                                  end < name >
< input list > ::=  { < type declaration > }
< type declaration > ::=  < name > : < type >
< output list > ::=  { < type > }

< expression > ::=  < primitive expression >
                 | { < expression > }+
                 | < let-block expression >
                 | < conditional expression >
                 | < application >
< let-block expression > ::=
                 let { < type declaration > }; { < name def > }; in < expression > end
< name def > ::=  { < name > } = < expression >
                 | < name def >; { < name > } = < expression >
                 | < empty >

< conditional expression > ::=
                 if < expression > then < expression > else < expression > end

< application > ::=  < name > ( < expression > )
< primitive expression > ::=
                 < expression > < primitive operation > < expression >
                 | < primitive operation > ( < expression > )
                 | < name >  | < constant >
< simple data type > ::=  integer | real | boolean | character-string | structure
< type > ::=  < simple data type > | stream of < simple data type >
```

Figure 3.1. Syntax of a value-oriented language

procedure yields an ordered set of *output values* of types $R_1,...,R_n$ resulting from the evaluation of <expression>. While each procedure in the list of procedure definitions may itself contain procedure definitions, we adopt for simplicity the scope rule that all procedure names are globally defined - that is, no two procedures can have the same name in the entire program.

An expression has several attributes: arity and ordering. Each expression yields an ordered sequence of values. The arity of an expression is the size of the sequence of values it yields. We give a recursive definition of the arity, A(E), of each of the six types of expressions as follows:

A(<primitive expression>) = 1,
A(<exp_1>, . . . ,<exp_k>) = A(<exp_1>) + ... + A(<exp_k>),
A(<let-block expression>) = A(let <definitions> in <exp> end)
= A(<exp>),
A(<conditional expression>) = A(if <exp> then <exp_t> else <exp_f> end)
= A(<exp_t>) (and must equal A(<exp_f>)),
A(<application>) = the number of results listed in the yields clause of the procedure definition.

For a procedure to be well defined the arity of the expression of a procedure must match the number of result types declared in the yields clause. Names appearing in an expression must be defined either in the input list of the procedure or be procedure names.

In many situations it is convenient to introduce a name for an expression because it is a common subexpression of a larger expression or because it is necessary to build a new expression whose values are permutations of another. The let-block expression is used for introducing names each standing for an expression of arity one. A let-block expression is of the form:

let { <type declaration> }

<name-list$_1$> = <exp$_1$>;

.....

<name-list$_k$> = <exp$_k$>;

in <exp> end;

Where the names in <type declaration> of a let block are temporary names meaningful only within the block, and any reference to these names outside of the block is not defined. These names must be distinct from each other and may appear in the expressions <exp$_1$>,...,<exp$_k$>, and <exp>. Since they may conflict with names for inputs of the procedure or names defined in outer let-blocks the scope rule is that innermost definitions take precedence over the outer definitions. Type declaration of names is in the form:

name$_1$: type$_1$, . . . , name$_k$: type$_k$;

where type$_1$,..., type$_k$ are one of the allowable types.

We require that the number of names in a name-list be equal to the arity of the expression on the right side of the equality sign. The *value* of a name in a name-list is the value of the corresponding expression appearing on the right hand side of the equal sign, and the value must be of the type specified by the type declaration of the name. The value of a let-block expression is the value of <exp> enclosed by in and end.

A conditional expression is of the form:

if <exp$_1$> then <exp$_2$> else <exp$_3$> end;

The expression <exp$_1$> must be a boolean value of arity one. The expressions <exp$_2$> and <exp$_3$> must have the same arity and the corresponding value in each expression must be of the same type. The value of a conditional expression is: <exp$_2$> if <exp$_1$> evaluates to the

boolean value true; <exp$_3$> if <exp$_1$> evaluates to false; otherwise, undefined.[1]

A procedure application expression is of the form:

P(<exp>);

where the arity of the expression <exp> is the number of input values required by procedure P and the type of each value must match that of the input specification. The result of the procedure application is a sequence of values of size and types specified by the yield clause of the procedure heading.

A primitive expression is an expression that uses the set of primitive operations defined on the data types. For historical reasons we introduce two forms of primitive expressions: *infix* and *prefix*. An infix expression is of a form:

<exp$_1$> operation <exp$_2$>;

where the operation must be a binary operation, and <exp$_1$> and <exp$_2$> must be of arity one and of compatible type with the operation. A prefix primitive expression is of the form:

operation (<exp>);

where the expression must be of arity and type compatible with the operation.

1. We will assume that, most data flow actors produce the value undefined, if some required input value is undefined.

An example

We give a procedure that defines a parallel factorial computation below:

```
Factorial = procedure ( n : integer ) yields integer;

    Product = procedure ( n1 : integer, n2 : integer ) yields integer;
                  if n1 >= n2 then n1
                  else let middle : integer;
                           middle = ( n1 + n2 ) quotient 2;
                           /* this is an integer division */
                           in Product( n1, middle ) * Product( middle+1, n2 ) end
                  end
              end Product;

      if n < 0 then error else Product(1, n) end;
    end Factorial;
```

3.2 Correspondence between the language and data flow schemas

A procedure of m inputs and n outputs corresponds to an (m+1, n+1) data flow schema. The m input links corresponds to the m inputs of the procedure. The data flow schema has an additional input link called the *trigger* link whose purpose is to send trigger values to constant actors in the schema. The additional output link is for passing *signal* values from sink actors. As a convention, we require a *trigger* input link and the *signal* output link be there whether constant actors and *sink* actors are used in the procedure or not. Internal actors of the data flow schema evaluate the expression of the procedure.

The translation of a program in the language into data flow schemas is quite simple due to the value-oriented semantics of the language. We give an informal and recursive translation procedure below. In this translation procedure each expression is translated into an (m, n) schema S whose input links are labeled by names. We shall use the notation In(S) to

denote the set of names used as labels for the input links of the schema S. The notation Size(α) defines the number of distinct names in the set α; $(\alpha \cup \beta)$ defines the union of two sets α and β; $(\alpha - \beta)$ defines the set that contains the elements in α which are not in β.

Given a procedure P, it contains a set of procedure definitions $\{ P_i \}$ and an expression E.

(1) Translate each procedure P_i into an (m_i, n_i) schema, and add the name P_i to the global name space of the program. Since procedure names are uniquely defined , there is no conflict of names in the name space.

(2) The translation of an expression is defined by cases according to the syntactic structure of the expression.

(a) E = < primitive expression >

If E is a name, then it is translated into a single link actor labeled by that name. If E is a constant, it is translated into a constant actor whose input arc is connected from a link actor labeled trigger and whose output arc is connected to a link. If E is a primitive expression,

<primitive operator> (E_i),

then the resulting schema S for E is an (m, n) schema where $m = m_1$ assuming E_1 is translated into an (m_1, n_1) schema S_1. The connection between the input arcs of the primitive operator and the output links of schema S_1 is implicitly defined by the ordering of the expression E_1 as shown in Figure 3.2(a). The input links of S_1 become the input links of S. The output arcs of the primitive operator are connected to the output links of S and an extra output link is created and labeled signal if the schema S_1 contains an output link labeled signal. Thus, n is either equal to the output arity of the primitive operator or is larger than it by one.

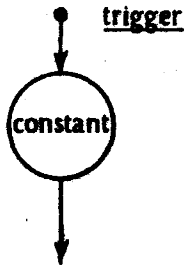
(b) $E = E_1, \dots, E_k$

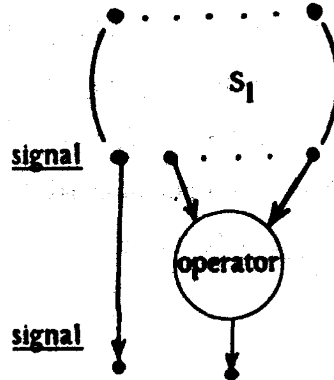
Translate $E_1, \dots, E_k$ into $S_1, \dots, S_k$, where each S_i is an (m_i, n_i) schema. The schema

(a) E = < primitive expression >

E = < name >　　E = < constant >　　E = < primitive operation > (E_1)

• name





(b) E = $E_1, \ldots, E_k$

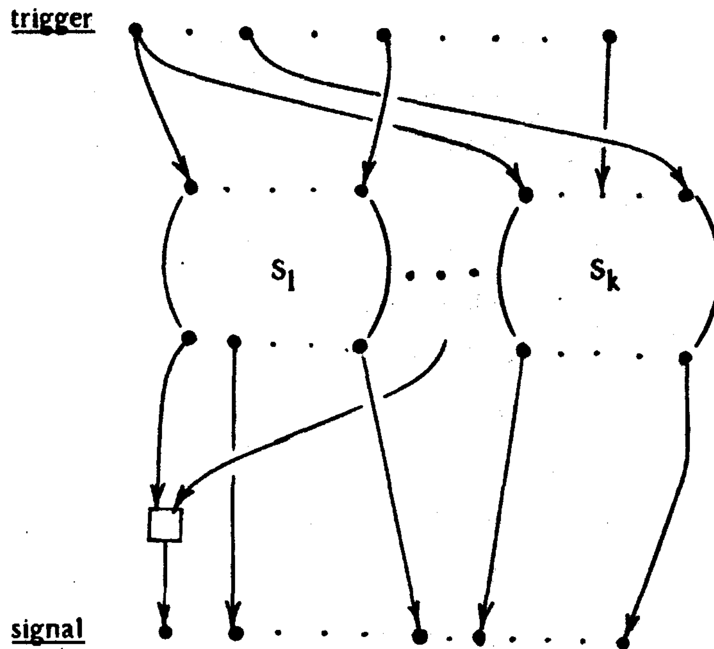


Figure 3.2(a), (b) Translation Rules

S is an (m, n) schema such that

$m = \mathrm{Size}(\ In(S_1) \cup \ldots \cup In(S_k)\ )$

$n = n_0$ = the sum of n_i, for $i = 1, \ldots, k$, if none of the output links of S_i are labeled signal; otherwise

$n = 1 + n_0$ - (the number of output links labeled signal).

The construction of S from S_i's is by connecting the set of m input links to the input links of each S_i according to the labels of their input links and by connecting all output links of $S_1, \ldots, S_k$ to the n output links in the order defined by the expression such that all output links labeled signal are connected to the only output link of S labeled signal. (Refer to Figure 3.2(b).)

(c) E = let T, N result E_0 end

The type definition T only provides information for compile time type checking; N is the list of name definitions containing k names; and E_0 is an expression. The translation of expressions in N yields an (m_1, n_1) schema S_1 where $n_1 = k$ or $k+1$ depending on the existence of an output link labeled signal. These k output links are labeled with names in N according to the definition. The translation of E_0 yields an (m_0, n_0) schema S_0.

The (m, n) schema S is constructed by cascading S_1 and S_0 such that the set of input links in S_0 labeled with the names in N are connected to the output links of S_1. The set of m input links are labeled with names in the set $(In(S_1) \cup (In(S_0) - N))$ and are connected to input links of S_1 and S_0 according to the labels. The output links of S includes the n_0 output links of S_0 and may contain an output link labeled signal if one of the following three conditions is true:

(i) S_0 or S_1 contains an output link labeled signal. In this case simply connects all such output links to that of S.

(ii) The set $(N - In(S_0))$ is not empty. This implies that the set of names defined in N are not all used in the expression E_0; and, therefore, must be discarded using sink actors which are then connected to the output arc labeled signal.

The resulting schema is shown in Figure 3.2(c).

(d) E = if E_1 then E_2 else E_3 end

Let S_1, S_2, and S_3 be (m_1, n_1), (m_2, n_2), and (m_3, n_3) schemas translated from E_1, E_2, and E_3 respectively. For a well formed conditional statement, note that n_2 differs from n_3 at most by one. The S is an (m, n) conditional schema such that m = Size($In(S_1) \cup In(S_2) \cup In(S_3)$). This conditional schema contains m' switch actors, where m' = Size($In(S_2) \cup In(S_3)$). (Notice that m' may be less than m_3 becuase some inputs are used only in the predicate of the conditional schema.) It contains n_3 merge actors, where n_3 = maximum(n_1, n_2). The true branch of the conditional schema is obtained by modifying S_2 by adding additional sink actors if m' > m_2; the false branch is similarly constructed. This construction results in a schema S shown in Figure 3.2(d).

(e) E = < procedure application > = P (E_1)

Let P be the name of a procedure which is defined to have m input values and yields n output values. The translation of the expression E_1 produces an (m_1, n_1) schema. The schema S for E is constructed using a constant actor of value "P" and an apply actor of m+2 inputs and n+1 outputs as shown in Figure 3.2(e). The apply actor requires m+2 inputs because the first input is for the name of the procedure and the m+1 inputs and n+1 outputs are for the (m+1, n+1) schema translated from the procedure P.

(3) The application of the translation rule to the expression E yields an (m'+k, n') schema S, where m' = m, or m+1 and n' = n, or n+1, if the procedure P is defined to have m input values and n output values. The extra k input arcs are due to the procedure names used in the expression E, and m' and n' depend on whether a trigger input link and a signal output link is produced during the translation. We obtain the final (m+1, n+1) schema for P by adding constant actors whose values are k procedure names and by adding a trigger

(c) E = let T, N in E_0 end

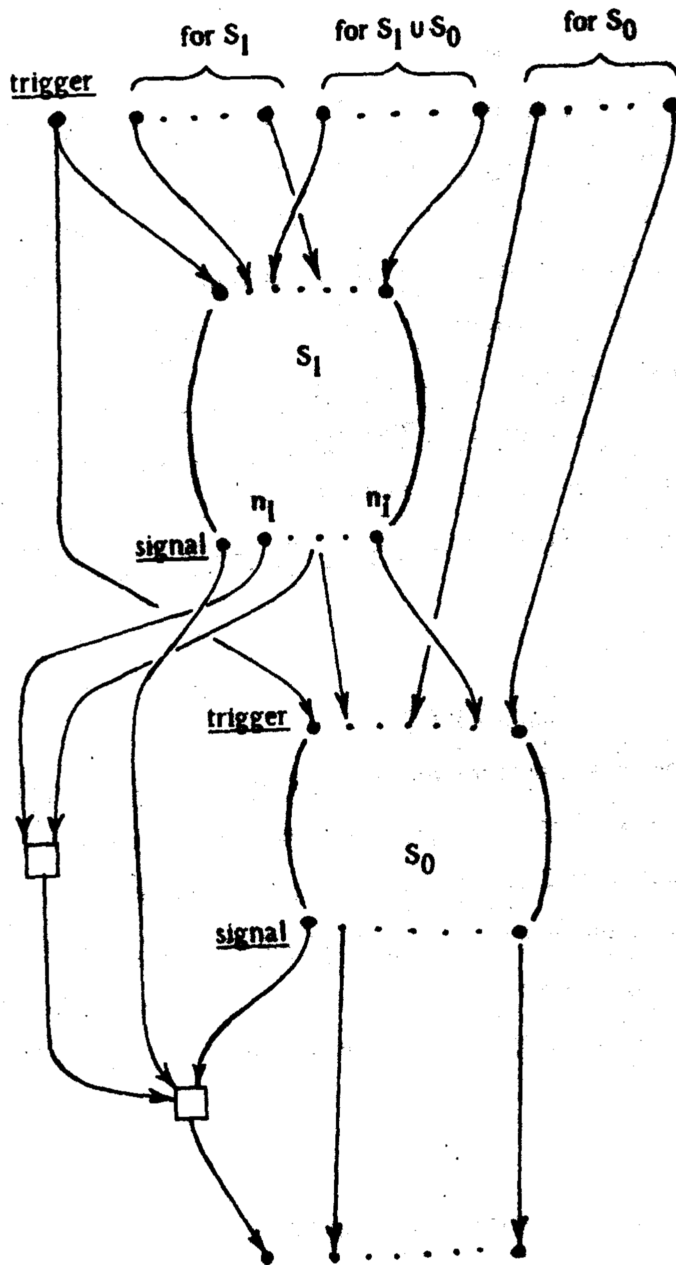


Figure 3.2(c) Translation rules

(d) $E = \underline{\text{if}}\ E_1\ \underline{\text{then}}\ E_2\ \underline{\text{else}}\ E_3\ \underline{\text{end}}$

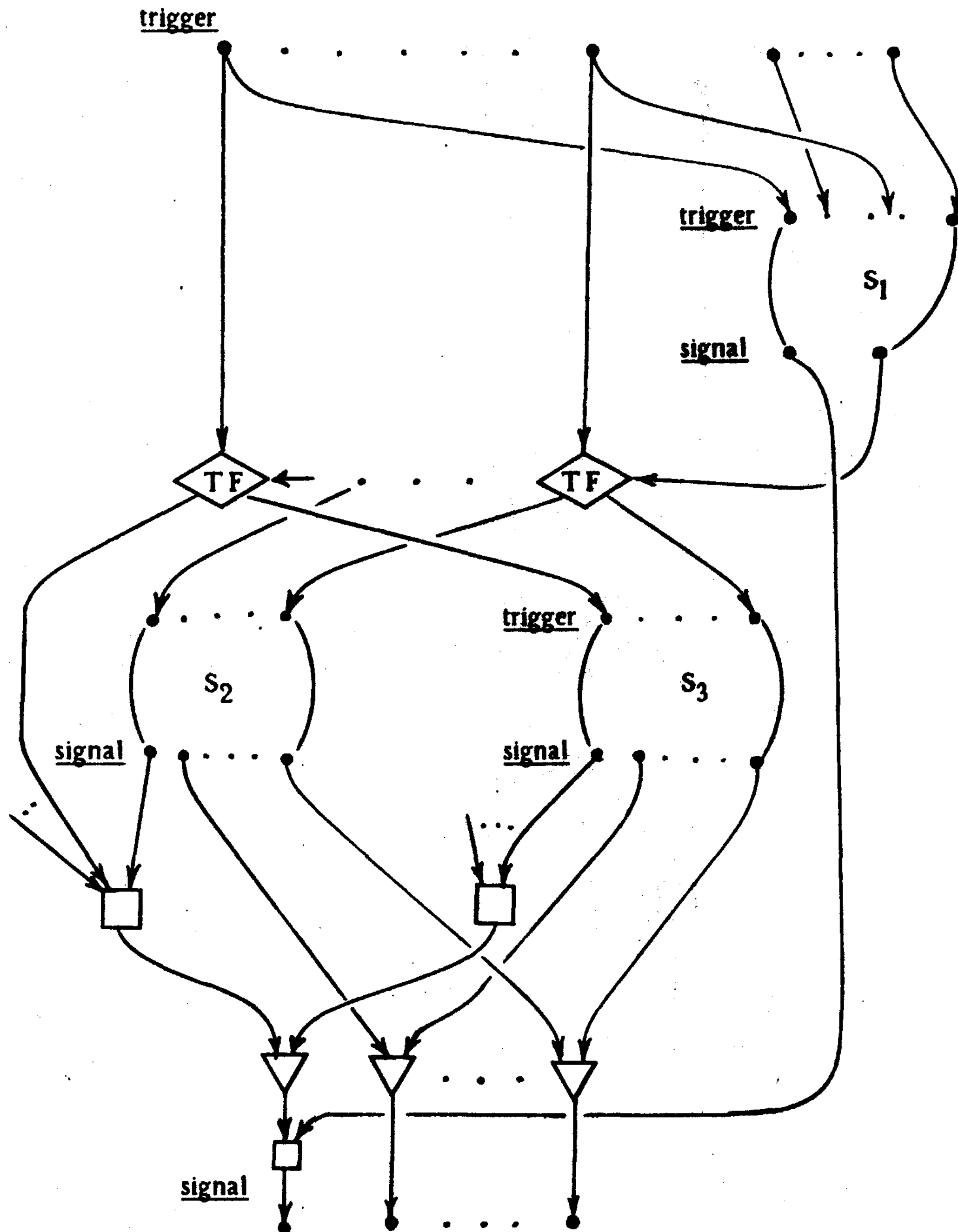


Figure 3.2 (d) Translation rules

(e) $E = P(E_1)$

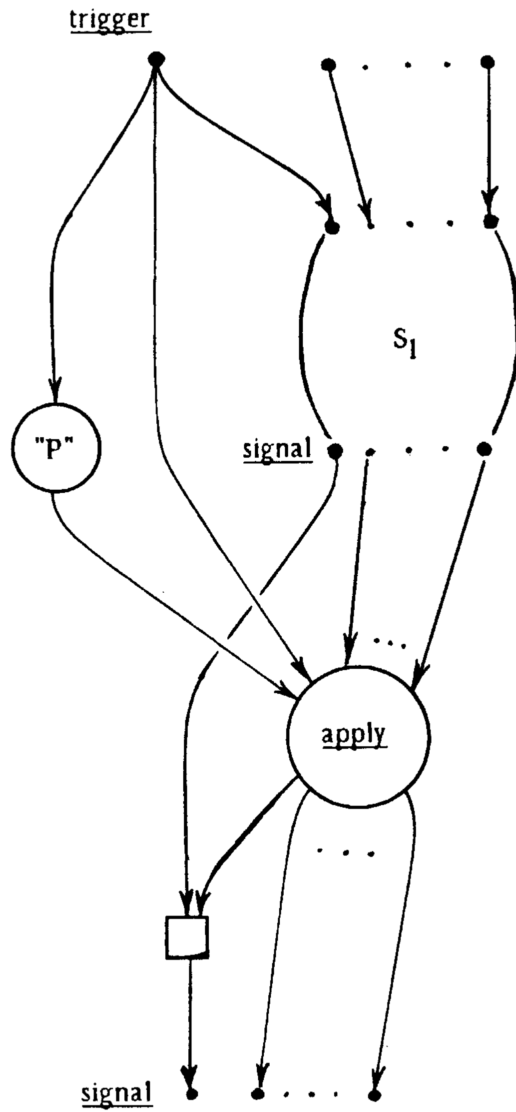


Figure 3.2(e) Translation rules

and a signal link if necessary.

This concludes the translation procedure. The result of the translation procedure on the procedure definition for computing the factorial of an integer is shown in Figure 3.3.

3.3 Discussion

We have not introduced data type declarations for arrays or records. It is desirable to introduce additional declaration mechanisms for defining data structures of specific forms such as array, record and union types, because such declarations provide effective compile time checking which would otherwise be costly at execution time of a program. These are regarded as extensions not of our primary concern in this thesis.

The implementation of procedures as values (or, procedure-values) is a very subtle issue that involves both the representation of procedures and the manner in which procedural values are used. In this simple language, we have only allowed application of procedures that are defined at compile time. The use of a global name space for procedure names is overly restrictive in that there are many situations where definitions of local procedures are desirable without regard to use of names. The use of a global name space also violates principles of programming methodology which emphasize the importance of modular program structures and language structures which guard against the propagation of unintended or malicious side-effects.

In a more general programming language, we would like to be able to dynamically create procedures by compiling a procedure definition or by combining existing procedures to yield another procedure whose function is the result of composition of others. To implement these operations on procedure-values in an operation model that is free of side-effects presents several problems.

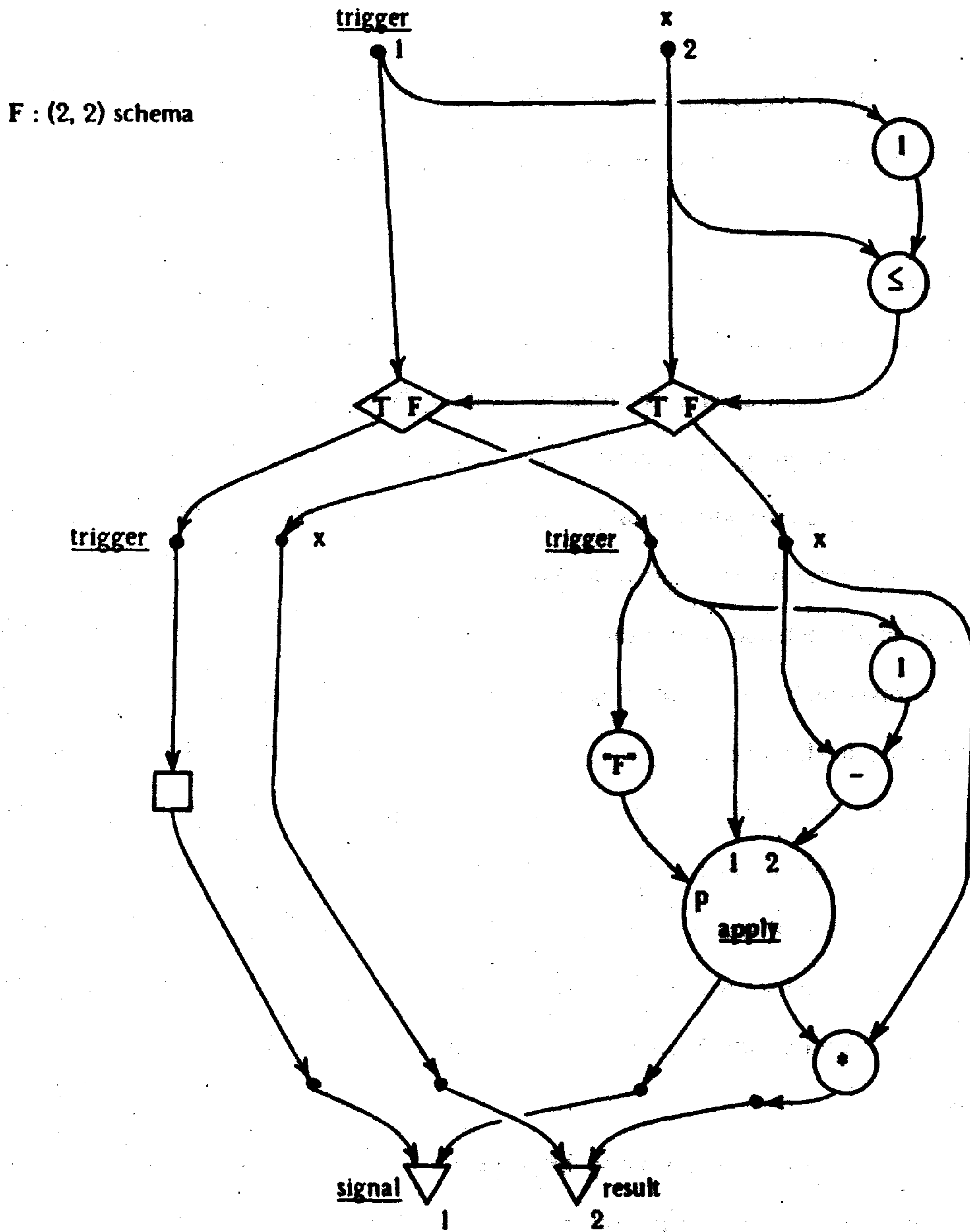


Figure 3.3. A example of translation rules on the procedure F:
F = procedure (x : integer) yields integer
if x =< 1 then x else x * F(x - 1) end
end F

The creation of procedures cannot simply cause updates to the global name space, since this would create side-effects for the processes having references to it. Another problem relates to the construction of recursive procedure definitions. In Henderson's binding model [Hende75], the construction of recursive procedures is cast in an operational model that allows data structures containing cycles. In the language presented here, we have been able to allow recursive definition of procedures by introducing a global name space such that no cycles are created. While it is possible to extend this scheme for constructing recursive procedures dynamically, it seems premature to define any implementation of procedural values without further conclusions regarding the desirability of data structures containing cycles.

This page intentionally left blank

Chapter 4. Implementation of Data Flow Schemas in a Data Flow Processor

The data flow schema model presented in Chapter 2 is based on the graphical representation and a data flow interpreter that implements its operational semantics. In this chapter we present the structure of the data flow processor and an implementation of the interpreter. Section 4.1 introduces the structure of the data flow processor, and the remaining sections describes the representation of a schema as a data structure and that of an activation of a schema. In Section 4.3, we present additional modifications on data flow schemas for implementing the semantics of procedure activations.

4.1 Data Flow Processor

The structure of a data flow processor for supporting the execution of recursive data flow schemas is shown Figure 4.1. It consists of six subsystems: Functional Units, Structure Controller, Execution Controller, the Arbitration and Distribution Networks and the Packet Memory. The processor is based on a packet communication design principle that has been advocated by Dennis [Denns75]. The arcs between subsystems represent channels through which packets of the specified types are sent. Two major subsystems of interest to us are the Packet Memory and the Execution Controller.

The Packet Memory holds data structures as collections of storage nodes, called *items*, each of which represents a tuple of a one-level data structure. An item may have scalar values and unique identifiers of other items as its components each identified by its selector. Thus, a collection of items can represent an acyclic directed graph where each arc corresponds to a unique identifier component of the item representing its origin node. The Packet Memory maintains a reference count for each item and reclaims physical storage space when items

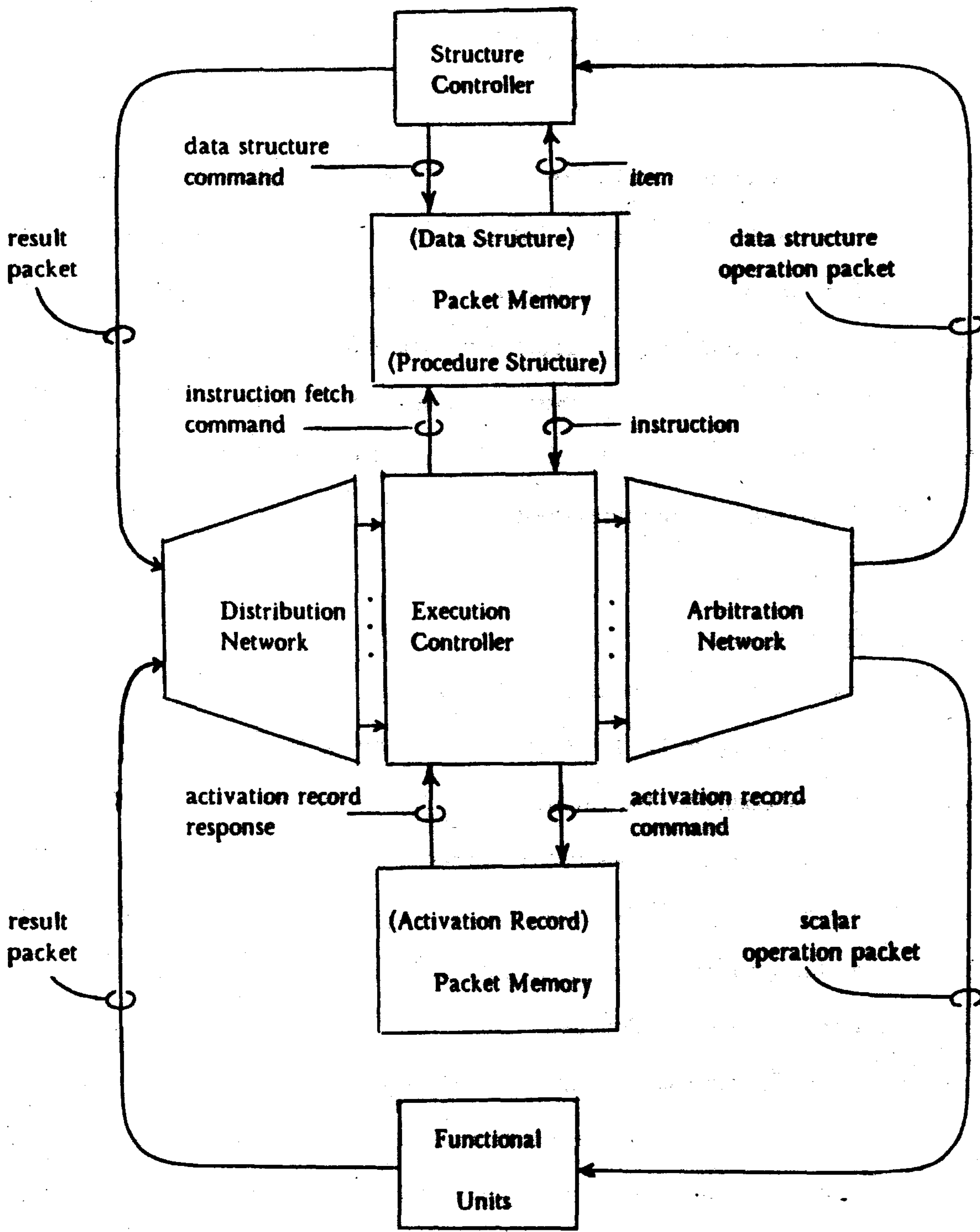


Figure 4.1. Data Flow Processor

become inaccessible.

Structures held in the Packet Memory have three roles in the execution of data flow schemas:

(1) as operands for the data structure operations implemented by the Structure Controller;

(2) as *procedure structures* that represent data flow graphs and have as components *instructions* of a data flow procedure which are encodings of actors and their output arcs in a data flow schema; and

(3) as *activation records* which hold operand values, i.e. tokens arrived at an actor, for each actor instance while waiting for their enabling condition to be satisfied.

The concept of a Packet Memory System was introduced by Dennis, and the design issues for these systems and the Structure Controller have been studied [Denns75, Acker76]. In Chapter 6, we discuss in greater detail the properties of the Packet Memory that must be satisfied to support these structures effectively.

The Execution Controller fetches instructions from a procedure structure and operands from an activation record that are stored in the Packet Memory and forms them into operation packets. Each operation packet is passed to the Arbitration Network for transmission to an appropriate Functional Unit if a scalar operation is called for, or to the Structure Controller for the data structure operations. Instruction execution in the Structure Controller and Functional Units generates result packets which are sent through the Distribution Network to the Execution Controller where they will join with other operands to activate their target instruction.

The Arbitration and Distribution networks are both store and forward networks and

can forward a packet from any one of the input ports to any one of the output ports.[1] It is important to realize that the delay of packet traversal through the networks is subject to variations due to the resolution of contention for buffers among packets in the networks. Thus, the Execution Controller has to store the result packets as operands and detect the enabling configuration of an actor regardless of the order of arrival of these packets. That this can be implemented correctly will be seen later when we give detailed representations of procedure structures and activation records.

Although the Execution Controller, Structure Controller and the Packet Memory are shown in Figure 4.1 as single units, each is in fact a collection of many identical units. For example, the Packet Memory subsystem would consist of separate systems, each holding all items whose unique identifiers belong to a well defined partition of the address space of unique identifiers. The Execution Controller subsystem consists of identical modules each of which would serve a distinct subset of procedure activations.

4.2 Procedure Structures and Activation Records

This section presents several alternatives to the representation of procedure structures and activation records. Section 4.2.1 presents a simple representation and may incur unnecessary delays in instruction execution. Section 4.2.2 gives two other alternatives. In the rest of the thesis, however, we will assume that the simple representation presented in Section 4.2.1 is used.

---

1. We refer readers to [Bough78] for further readings on a possible approach to the design of such networks.

4.2.1 Procedure Structures and Activation Records

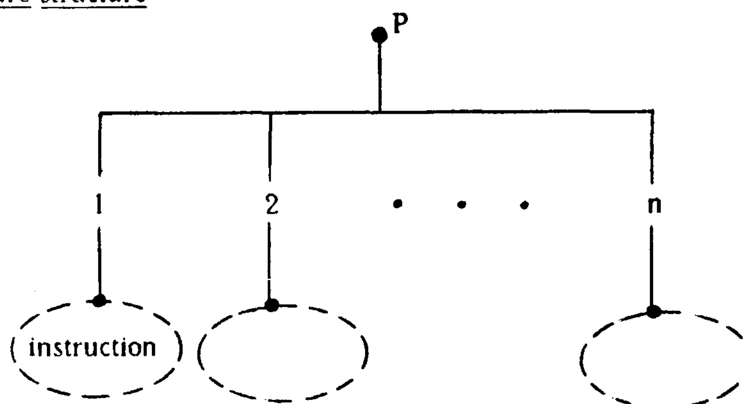
A data flow schema is represented in the machine by a kind of data structure called a *procedure structure.* A procedure structure corresponding to a data flow schema of n actors is a data structure having n components with integer selector names from 1 to n assigned to the actors. Each component, called an *instruction,* is an encoding of an actor and its output arcs.

An actor having n output arcs is encoded as a data structure shown in Figure 4.2. We shall call the components *fields* of an instruction. The *Operation* field defines the function performed by the actor, the *destination* fields D1, ..., Dn define n output arcs. Each destination field has three subcomponents: the *Inst* component is the integer selector name of the destination; the *Input-Arc* component is an integer designation of an input arc of the destination; and the *count* component is the number of result packets expected by the destination.

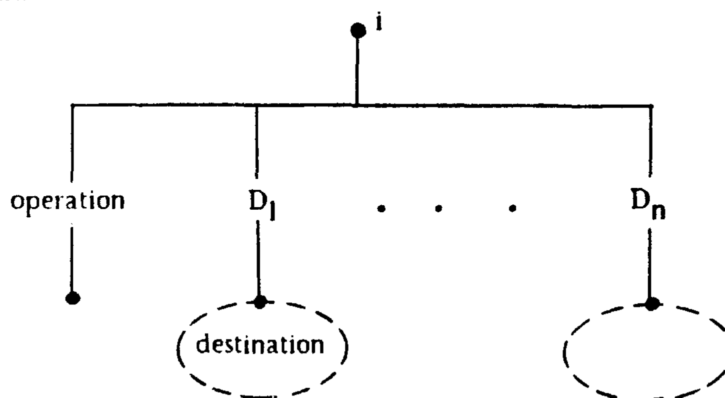
Since multiple instances of the same schema may be concurrently active in a computation, each activation (an instance of a procedure execution) is represented as a separate *activation record* whose representation is shown in Figure 4.3. Each actor in an activation is uniquely identified by the tuple (A, i), where A is a uid of the root node for the activation record and i is the integer assigned to the actor in the procedure structure. A token of value v on the k-th input arc of an actor (A, i) corresponds to a *result packet* that carries the information (A, i, k, v, count), where count indicates the number of tokens (or operands) required for the enabling of the actor.

An actor is enabled when the number of result packets having arrived at the operand record -- the i component of the activation record A -- is the same as the count in the result

(a) procedure structure



(b) instruction



(c) destination

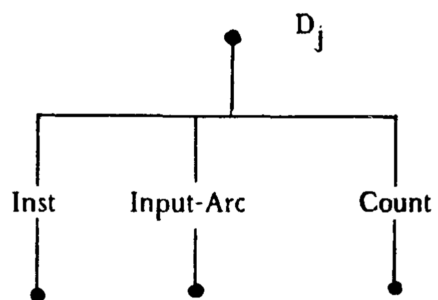
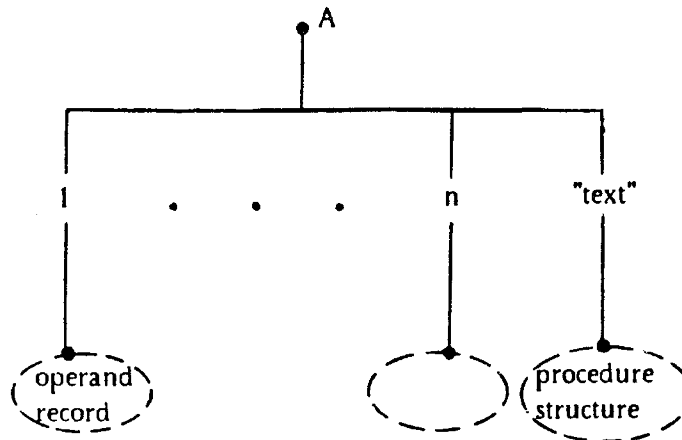


Figure 4.2. Procedure Structures

(a) activation record



(b) operand record

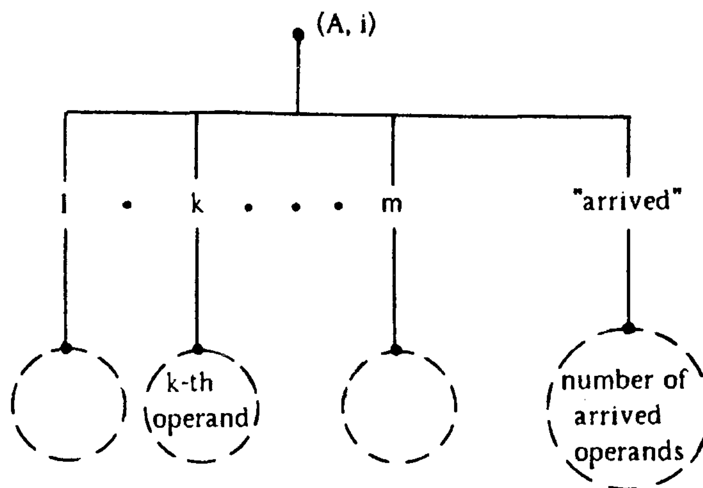


Figure 4.3. Activation Records

packet. [1] The detection of enabling is a function of the Execution Controller that processes activation records. Upon enabling of an actor instance (A, i), the instruction of the actor is fetched from the i component of the procedure structure.

An activation record shown in Figure 4.3 has components with integer selectors for operand records and an additional "text" component that is the procedure structure for the activation. (In our implementation, this component may be shared by other activations of the same schema.) An operand record may have as many integer subcomponents as input arcs of an actor, and also contains an "arrived" subcomponent indicating the number of arrived result packets. Since an activation record stores values of arrived result packets in its components, operations on an activation record modify its components. The operations on activation records are defined below:[2]

(1) create-activation(P)

This returns a new activation record with P as its "text" component and with no other components.

(2) insert(A, s, v)

The operation adds to A an s component with value v. The selector s is of the compound form i.k where k denotes the k-th input arc of the instruction i. The operation increments the i."arrived" component by one and returns the incremented value. If the i."arrived" component is undefined the value is taken as zero since it

---

1. With the exception of the merge actor, the enabling condition is easily implemented by test of equality. Under the restricted use of merge actors in well formed data flow schemas, a merge actor is enabled when it receives one input token.

2. We have treated each operand record as a structure with selector names. This should be considered an abstraction that can be implemented in an optimized form. A practical implementation of the operand record would be based on some mapping of the fields into operand records of a fixed size.

indicates that the field is non-existent.

(3) remove(A, i)

This operation deletes the i component of A; and is performed by the Execution Controller upon the delivery of the operation packet for the actor instance (A, i).

(4) free(A)

The operation deletes the entire activation record A. The section on the implementation of procedure activations gives an example of its use.

The Execution Controller consists of independent modules that provide caching of activation records. For each arriving result packet containing (A, i, k, v, count), the Execution Controller performs the operation insert(A, i.k, v) and tests the value of the "arrived" component against the count component of the result packet. If the values are equal, the instruction is fetched. Upon the arrival of the instruction packet at the Execution Controller, an operation packet containing the information (A, instruction, operands) is sent to the Arbitration Network containing the instruction and operands from the activation record. The i component of activation record A is then deleted by the Execution Controller.

The fetch command issued to Packet Memory is of the form:

< fetch, P, Inst, A >.

This packet causes the instruction structure of the Inst component of the procedure structure P to be brought into the Inst component of the activation record A.

4.2.2 Two other alternative representations

In this section we present two alternative representations of procedure structures and activation records that have some advantages over the one presented.

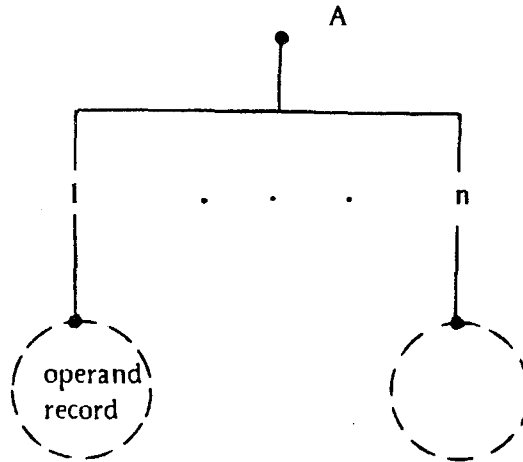
The procedure structure of the first scheme is the same as that of Figure 4.2, but the

activation records now have a "text" component for each operand record as shown in Figure 4.4. This component is supplied by result packets destined for the operand record. For each enabled instruction, the Execution Controller can, therefore, directly use the uid contained in "text" component of the operand record to fetch the instruction without having to obtain the uid of the procedure structure from the activation record presented in the previous scheme. In this scheme a result packet must, therefore, carry the information (A, i, k, v, count, P), where P is the uid of the procedure structure.

The second scheme is a further optimization of the first. This scheme eliminates the redundant information, the "count" and "text" component, carried by all result packets for each operand record. The procedure structure is shown in Figure 4.5, where the "tag" component of the destination field is a boolean value of either true or false and signifies that the values for the "count" and the "text" component of the destination operand record are to be sent if it is true, otherwise, only the operand value is contained in the result packet. The boolean value for the "tag" component of each destination structure must be assigned by the compiler such that a true tag is associated with one and only input arc of an actor. A schematic illustration of an example of this assignment is given in Figure 4.6, where the broken arc represents the destination field to which we have assigned the value true. In this figure, we have chosen the assignment rule that assigns true to the rightmost input arc of an actor. Note that a merge actor has two broken input arcs, this is because only one branch of a conditional schema is executed.

The content of a result packet is the tuple (A, i, k, v, count, P) if the tag for the destination (A, i) is true; otherwise, it is (A, i, k, v). The structure of an operand record is shown in Figure 4.7. Initially, the two components "arrived" and "count" are nil. For each result packet the "arrived" component is incremented by one and the resulting value is tested

(a) activation record



(b) operand record

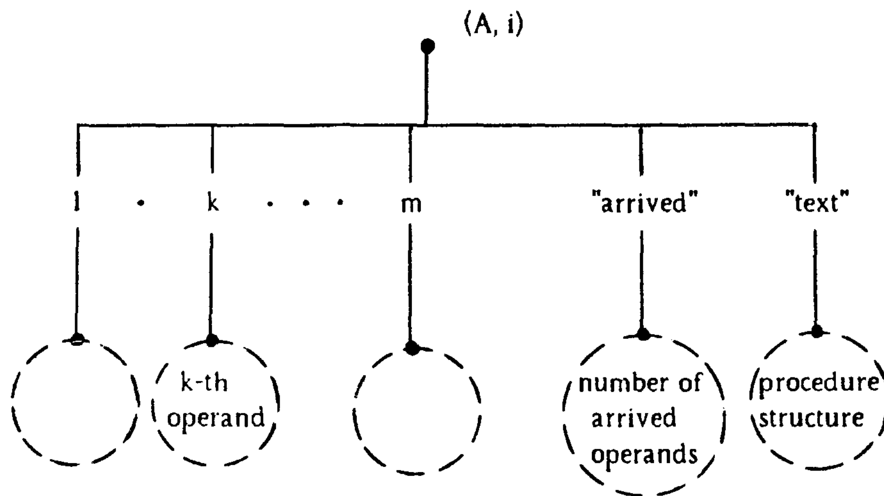
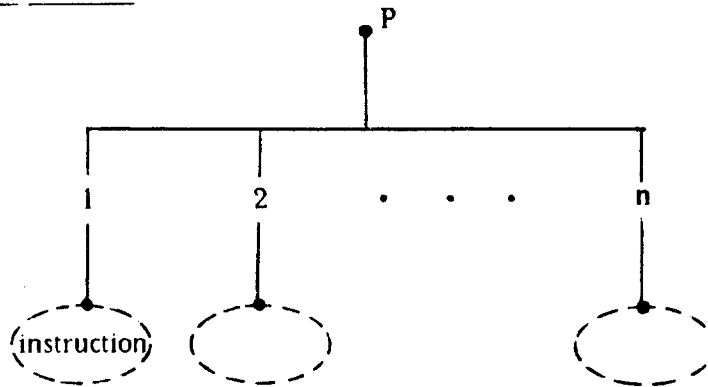
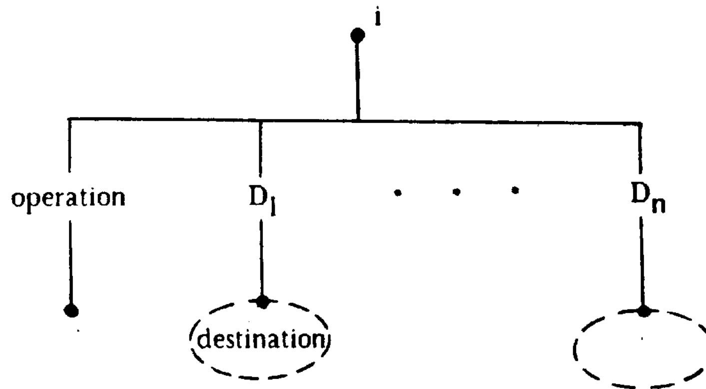


Figure 4.4. Activation Records

(a) procedure structure



(b) instruction



(c) destination

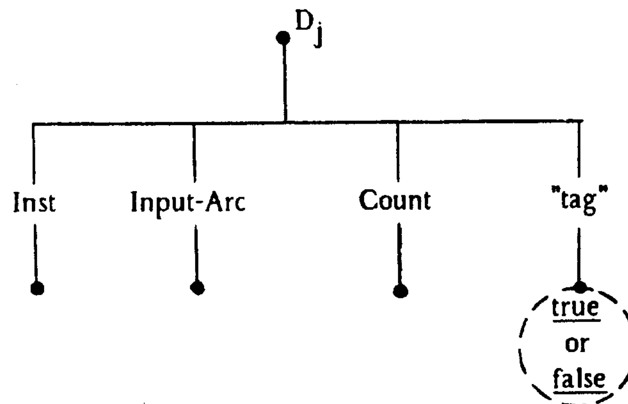


Figure 4.5. Procedure Structures

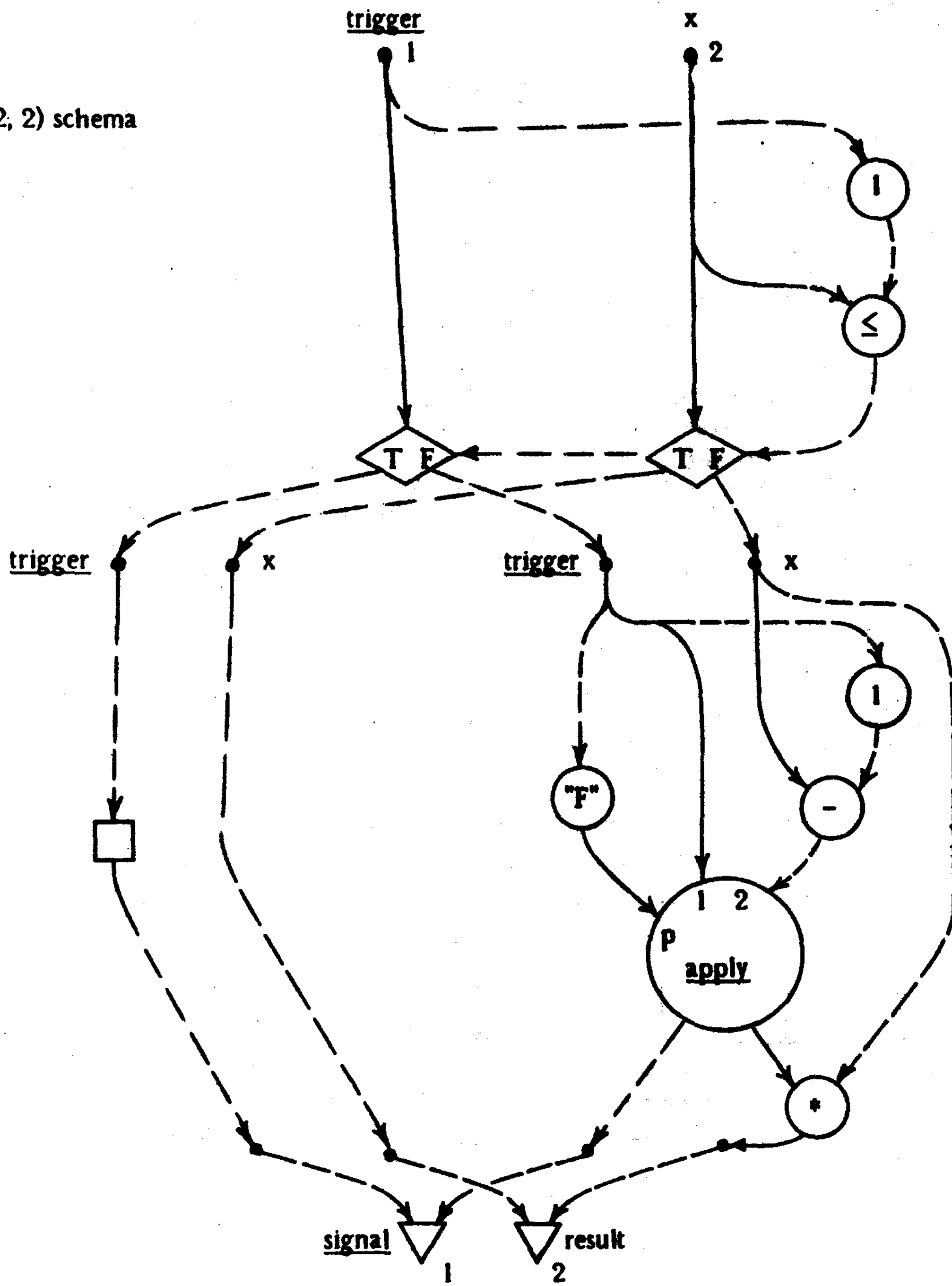
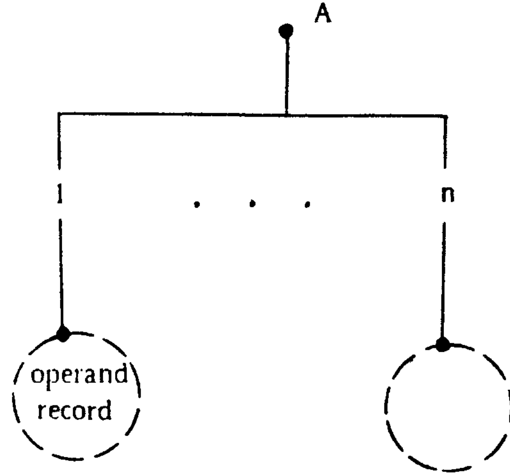


Figure 4.6. An example of tag assignments to the schema shown in Figure 3.3

(a) activation record



(b) operand record

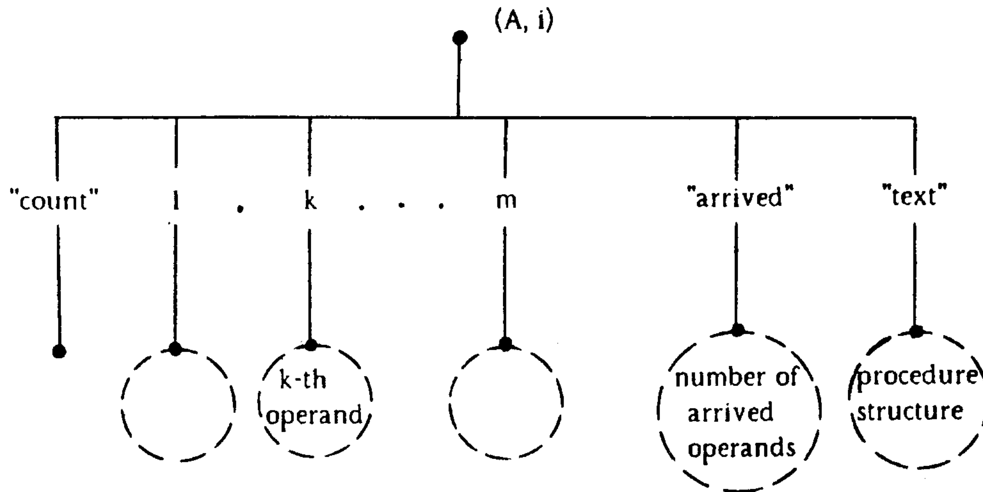


Figure 4.7. Activation Records

against the "count" component.[1] In addition, for a result packet of the form (A, i, k, v, count, P), the "count" component is written with the value count and the "text" component is written with the uid P. An instruction is enabled when the values of "count" and "arrived" are equal.

Notice that in all of the schemes presented, the instruction for an enabled actor is fetched only when it becomes enabled. Thus, there is an added delay between the enabling of an actor and the delivery of an operation packet. A further elaboration of the instruction execution scheme can be based on use of the "tag" field and can allow the instruction of an actor be to fetched before the actor becomes enabled.[2] This is achieved by requiring each subsystem that processes an operation packet to issue to the Packet Memory an instruction fetch for the destination operand record as it awaits for the arrival of other operands.[3]

These two schemes introduce additional changes to the implementation of procedure activations, since the input links and output links serve as the interface between procedure activations and must conform to the schemes described. We will not detail such changes, and will present the rest of the thesis based on the scheme described in Section 4.2.1.

4.3 Procedure activations

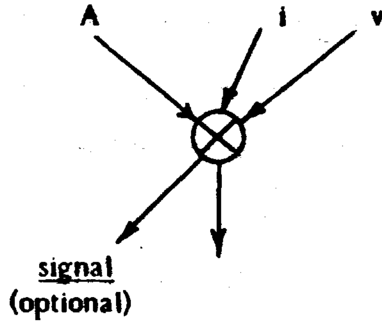
The problem of implementing procedure activations has been investigated by [Misun78, Miran77], we present here a scheme that is consistent with our representation of procedures. To implement application of schemas, we introduce four additional actors: linkage, make-ret, distribute and extract-uid. The symbols for these actors are shown in Figure 4.8. For

---

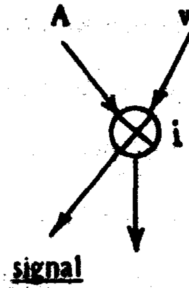
1. if the "arrived" component is nil, it is assumed to be zero.
2. This is similar to the instruction fetching schemes of lookahead processors. We mention that in this scheme the assignment of tags may be important.
3. In this case, the enabling condition can be modified such that it treats the instruction as an additional operand required for the enabling of the instruction.

(1) linkage

activation record

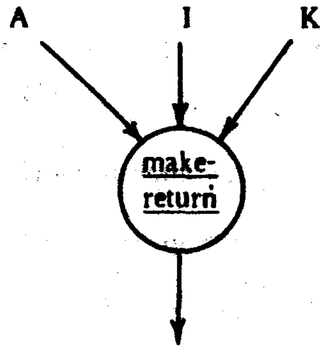


when constants are written into the actor



(2) make-return

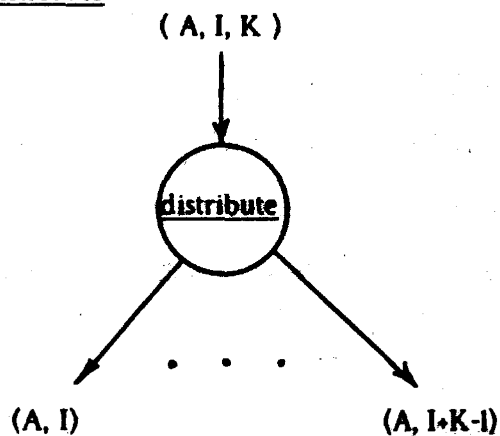
activation record base number of results



when constants are written into the actor

activation record

(3) distribute



(4) extract-uid

Figure 4.8. Actors for implementing procedure application

brevity, we illustrate the implementation with an example. The schema shown in Figure 4.9 is a translation of the schema for the factorial function shown in Figure 3.3, and embodies the additional actors. This embodiment is based on an *instruction assignment* rule that assigns integers to each actor of the augmented schema. The modification creates an (m+2, n) schema from an (m, n) schema translated from a textual program described in Section 3.2. The instruction assignment rule is the following (referring to Figure 4.9):

(1) The link actor labeled ret is assigned the integer one.
(2) The link actor labeled env is assigned the integer two.
(3) The remaining m input link actors are respectively assigned 3, ... ,and m+2.
(4) The linkage actors that supply input values to the new activation and actors that receive output values from it are respectively assigned consecutive integers. In Figure 4.9, the actors labeled I, I+1, ... ,I+3 are linkage actors supplying input values, and the link actors labeled J,J+1 receive result values from a procedure activation.
(5) The assignment rule for the remaining actors is arbitrary.

In Figure 4.9, the first input link actor labeled "ret" expects a value that encodes the destinations to which output values will be returned. The encoding consists of the uid of the activation record, the smallest integer assigned to the link actors receiving output values, and the number of output values. The distribute actor decomposes this tuple into destinations and forward them to output linkage actors of the new activation. A linkage actor communicates between two different activations and expects three inputs: a value v, an instruction number i, and a uid of another activation A. The firing of a linkage actor (A_l, i_l) in an activation A_l sends to the operand record (A, i) the result packet (A, i, l, v). In addition, this linkage actor may have a signal output arc destined for an actor within the activation A_l.

The second link actor expects the uid of the *environment structure* that contains all procedure structures with their names as selectors.

The semantics of the apply actor is implemented by using create-activation to allocate an activation record. The create-activation actor requires two inputs: a uid of a procedure

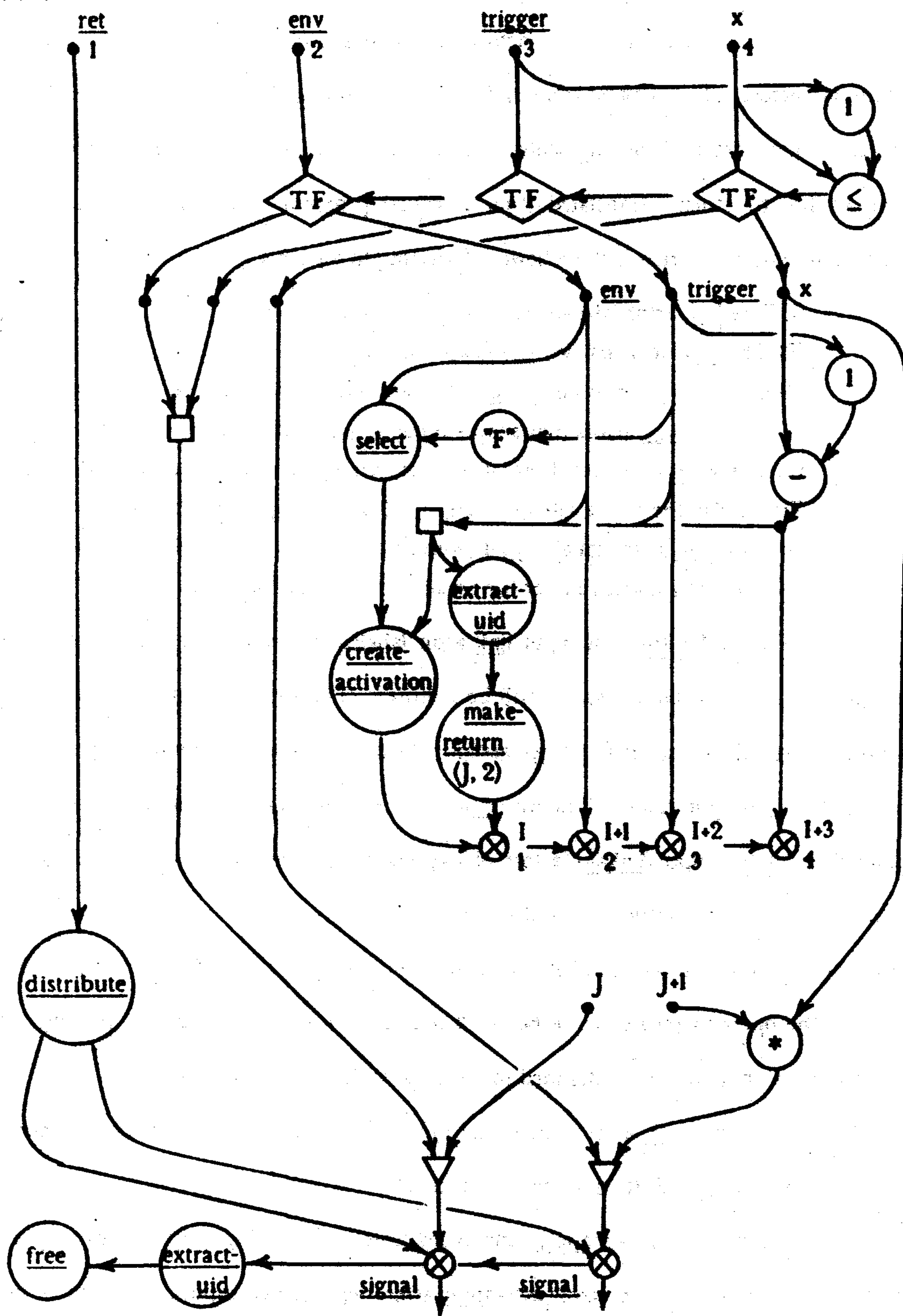


Figure 4.9. An example for the implementation of the apply actor

structure and a signal that is generated only when all input arguments for the activation have been computed; and its output is a free uid A. The uid of a procedure is selected from the environment structure using the name of the procedure. The uid of the activation record A is sent to the linkage actors I, ..., and I+3 which forward arguments to the activation. For these linkage actors the instruction number of the destinations are respectively assumed to be 1, ..., and 4. The value encoding the return destinations for the new activation is constructed by the const-ret actor using the output of the extract-uid actor which extracts from a result packet the uid of the activation; and it is sent to the first input link of the invoked activation through the linkage actor, I.

A free actor releases the activation record and is enabled only when all activities within the activation have ceased.[1] In Figure 4.9, notice that the signal output arcs of the output linkage actors on the bottom of the figure are connected to the free actor through a sink. Thus, the free actor cannot be enabled until all output linkage actors have delivered their output values. The uid of the activation record is returned to the pool of free uid's managed by the Packet Memory.

The translation of textual programs into augmented schemas is straightforward and can be based on the translation rules presented in Section 3.2, and we will omit further details of the process.

4.4 Tail procedure application

In sequential programming languages, a tail procedure application is a procedure application that occurs as the last statement in another procedure. For our value-oriented language, a tail procedure application is identifiable as a procedure application in the

---

1. This is guaranteed by the compiler that translates textual programs into data flow schemas.

expression of the body of a procedure whose output value is returned as the value of the entire procedure. For languages that have iterative constructs, the translation of an iteration loop into its equivalent recursive form of computation results in a tail recursive procedure. Often, some recursive programs can be transformed into tail recursions as well.[1] In programs with tail procedure applications, the result of a tail procedure application of P2 within P1 is simply the result of the procedure application P2. (If P1 and P2 are the same, then they form a tail recursive procedure.) Such tail procedure applications occur frequently enough that the activation record of P1 should be deallocated as soon as possible. Without such optimization, the outermost procedure activation remains until all nested procedure activations are freed.

Since the subject of compiler optimization is not within the scope of this thesis, we will simply present an example to illustrate how such optimization might be accomplished with the procedure application scheme introduced. In Figure 4.10, we give an alternative recursive program for the computation of the factorial function. In this schematic illustration, the link actor labeled ret provides the necessary information for the compiler to add actors to form the necessary linkage between the deepest nested procedure activation and the outermost procedure which invoked the factorial computation.

In Figure 4.11, we give examples of situations where tail procedure application can be optimized. While it is possible to optimize on reasonable cases of such tail procedure applications, it is not clear that the complexity introduced is desirable.

---

1. These translations are not assumed to be an important part of the task of the compiler, but such optimization may be embedded if feasible. For a user language which have iteration constructs, the translation would naturally lead to tail recursions, and thus the opportunity for this optimization should be taken advantage of.

G : (5, 2) schema

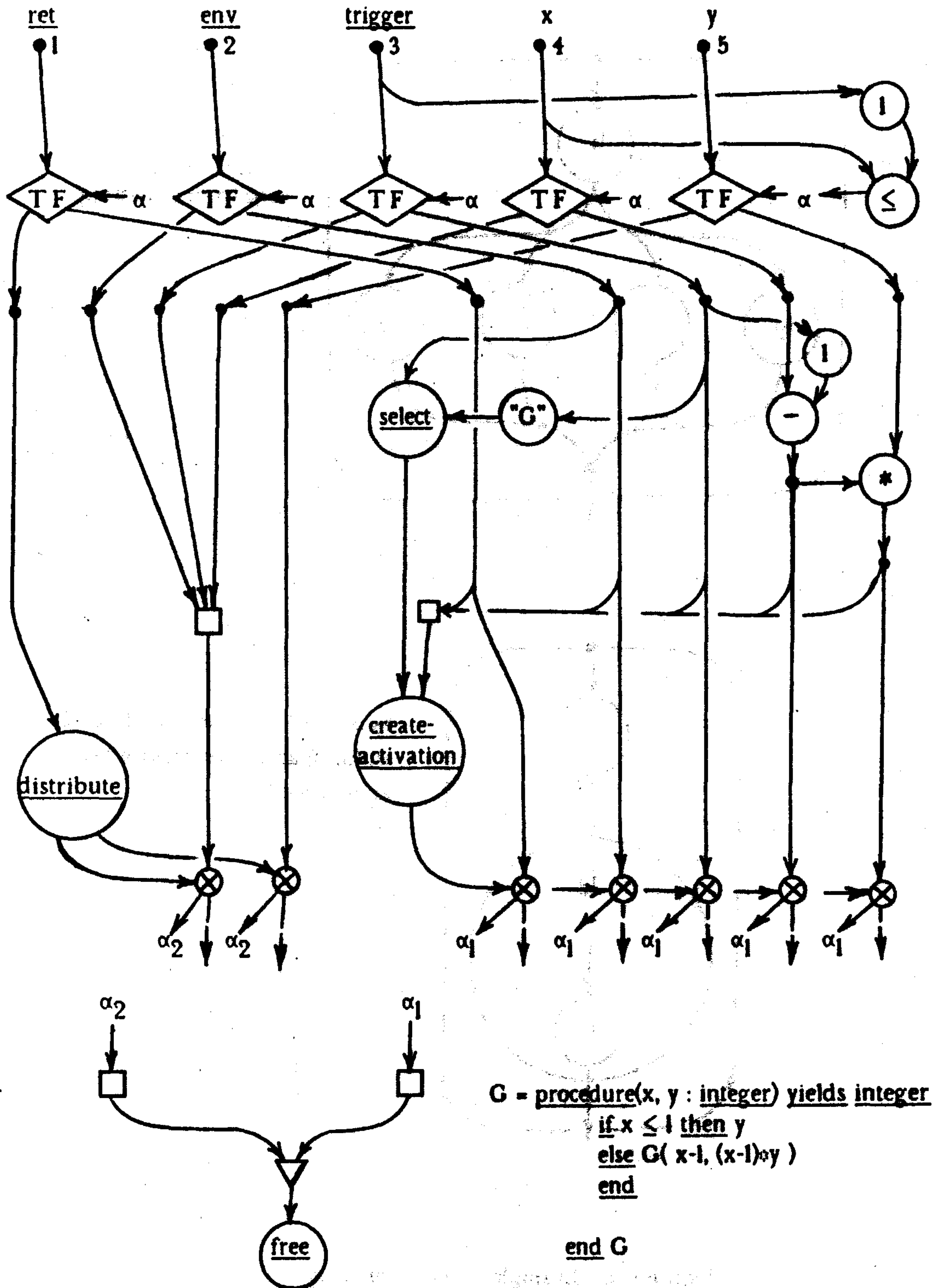
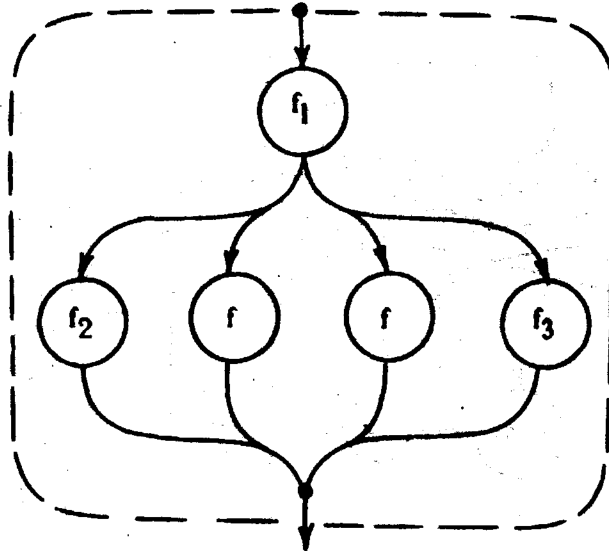


Figure 4.10. An example of a tail procedure application

(1)

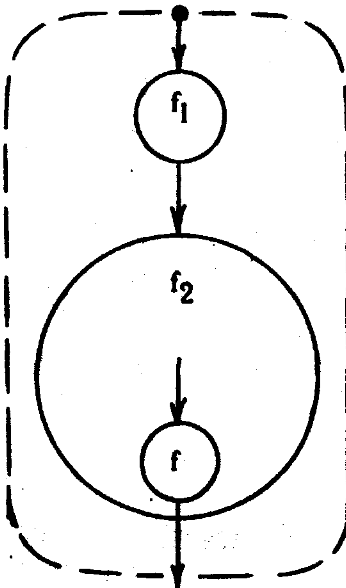
f :



f is a tail recursion in some cases of a conditaional nesting

(2)

f:



f_2 is a tail procedure application within f; and f_2 contains a tail application of f.

Figure 4.11. Examples of tail recursions

4.5 Discussion

In this chapter, we have presented a processor that is capable of supporting the semantics of the data flow schemas and the concurrency of operation. We have presented an abstract view of the operation of the processor and have discussed several alternatives of the instruction execution schemes. The choice of the execution scheme would depend on many factors that need further investigation. Some of these factors are: the delay characteristics of packet traversal through the networks, and the trade-off between the amount of storage needed to store operand records and the delay of instruction execution.

The instruction execution schemes we have presented are all called *piecewise copying* schemes, because each instruction is not fetched until the instruction is known to become an enabled instruction. Another alternative is to fetch all instructions of a procedure structure into an activation record at the time of creating the activation. This scheme would require that the instructions for actors on one branch of a conditional schema be deleted when the test outcome of the predicate for the conditional schema becomes known. This scheme also suffers from the larger storage required to store the instructions at any instance of time during the activation. Its advantage is that instructions can be fetched possibly with a single request to the Packet Memory rather than with as many requests as the piecewise copying schemes; thus, it reduces significantly the amount of packet traffic to the Packet Memory. At this level of discussion, it is not clear that this scheme offers greater advantages. To analyse this further would require further elaboration of the architecture and some understanding of the behavior of piecewise copying schemes.

The implementation of data structures and activation records by the Packet Memory has not been discussed in this chapter. We elaborate on this subject further in Chapter 6.

We have not detailed the translation from the language to the augmented schemas, but the details are straightforward and present no additional difficulties once the translation rule

presented in the Section 3.2 is understood.

Chapter 5. Stream, Nondeterminacy, and Forall

In this chapter we introduce several extensions to the language described in Chapter 3. These extensions are useful for expressing many forms of computation which are not conveniently expressible in conventional programming languages. Streams are an important abstraction for expressing computations on sequences of values. The implementation of this abstraction does not constrain the inherent concurrency of these computations and is guaranteed to be determinate when primitives for nondeterminacy are not used in the program. Another form of concurrency arises when a procedure is applied on all components of a data structure to produce new data structures or scalar values. The forall construct introduced in section 5.3 is a useful feature for expressing this form of concurrency.

Nondeterminate computations, computations that may depend on the timing of execution, can be expressed by merging two streams in a nondeterminate manner. It is important to realize that there may be computations which are not easily expressible with this extension of the language. This limitation is due to our lack of understanding of semantics for nondeterminate computations and of how such computations can be expressed in a value-oriented language.

5.1 Streams

The concept of a stream is an alternative approach for expressing computations that have conventionally been expressed as coroutines or a set of cooperating processes. For example, the organization of a compiler is often viewed as a set of coroutines each corresponding to a phase of the compiler, and we often view processes that perform input and output operations as a set of concurrent processes that coordinate using process synchronization primitives.

The significance of programming using streams has been recognized in many works on formal semantics [Landi65] and on programming languages [McIlr68, Denns69, Burge75, FreWi78].

There are many reasons for expressing computations in these forms. Large computations tend to create many large intermediate data structures that take up storage space. Coroutine mechanisms are often used to alleviate this problem by partitioning intermediate data structures into smaller units such that the total amount of storage used for intermediate data structures is reduced. The second reason is to allow these subcomputations to be concurrently executable by using explicit synchronization primitives. The third and subtler reason is that program structures expressed in these forms are more *modular* in the following sense: program modules can be expressed as a function over streams and their overall behavior can be characterized as compositions of these functions using denotational semantics [Kahn74].

Writing programs for applications that lead naturally to these forms of computations, however, has been difficult in sequential programming languages that have explicit coroutine mechanisms and synchronization primitives. Because these primitives require explicit initialization of either control sequences or common synchronization variables, the correctness of these programs is more often than not difficult to establish and programming errors may result in deadlocks or unwanted nondeterminacy.

Since many of these computations are inherently determinate, it is desirable to be able to express them in a more structured manner and without these undesirable properties. Using streams as presented here, one can express computations of these forms such that the inherent concurrency is not lost and the result of the computation is determinate and free of deadlocks.

5.1.1 Stream operations

A stream is a sequence of values, all of the same type, that are passed in succession, one-at-a-time between program modules. The operations on values of type stream of T are defined below where s and s' are streams, and c is a value of type T.

(1) []

The result is the empty stream which is the sequence of length zero.

(2) cons (c, s)

The result is a stream s' whose first element is c and whose remaining elements are the stream s.

(3) first (s)

The result is the value c which is the first element of s. If s = [], the result is undefined.

(4) rest (s) The result is the stream left after removing the first element of s. If s = [], the result is undefined.

(5) empty (s)

The result is true if s = [], and is false otherwise.

For a non-empty stream s, the following property is satisfied:

s = cons(first(s), rest(s)).

We shall use [1, 2, 3] to denote a constant of type stream of integer whose stream elements are the integers 1, 2, and 3. Using the notation we give examples of operations on stream values below:

Let x = [1, 2, 3] and y = 5, then

first(x) = 1,

rest(x) = [2, 3],

cons(y, x) = [5, 1, 2, 3],

empty(x) = false, and

empty([ ]) = true.

5.1.2 An example program

The problem of generating all prime numbers less than a given integer n is a good computation for illustrating how our data flow execution scheme can express highly concurrent computation using streams. The sieve of Erastosthenes [Knuth69] expressed in our textual language is presented in Figure 5.1.

The procedure "generate" produces the sequence of integers beginning with 2 which is processed by "sieve" to remove nonprime elements. Procedure "sieve" operates by taking the first element of its input as a prime and using which all multiples are removed by "delete" before applying "sieve" recursively to the remaining elements of its input stream.

In Figure 5.2, we show a snapshot of the execution of the program prime_generator. It can be seen that a substantial amount of concurrency exists in the computation if each activation of the procedure "sieve" can be executed as soon as the first element in the input stream is available. Section 5.2 shows how this concurrency can be achieved.

5.2 Implementation of streams

In this section we first present a correct and efficient implementation of streams, and then discuss why another alternative scheme is not adequate. The alternative scheme is presented here because it is a natural consequence of thinking in terms of tokens in the data flow model of computation, but it neither correctly nor efficiently implement the semantics of the language.

```
prime_generator = procedure ( n : integer ) yields stream of integer;

    generate = procedure ( i, n : integer ) yields stream of integer;
            if i < n then [ ]
                else cons ( i, generate( i+1, n ) )
            end;
        end generate;

    sieve = procedure ( s : stream of integer ) yields stream of integer;
            if empty ( s ) then [ ]
                else let    x : integer, s2, s3 : stream of integer;
                            x, s2 = first ( s ), rest ( s );
                            s3 = delete ( x, s2 );
                     in  cons ( x, sieve( s3 ) )
                     end;
            end;
        end sieve;

    delete = procedure ( x : integer, s : stream of integer ) yields stream of integer;
            if empty ( s ) then [ ]
                else let    y : integer, s2, s3 : stream of integer;
                            y, s2 = first ( s ), rest ( s );
                            s3 = delete ( x, s2 );
                     in     if divide ( x, y ) then s3
                                else cons ( y, s3 )
                            end;
                     end;
            end;
        end delete;

    sieve (  generate ( 2, n ) );

    end prime_generator;
```

Figure 5.1 A prime number generator using streams

(1) prime_generator

(2)

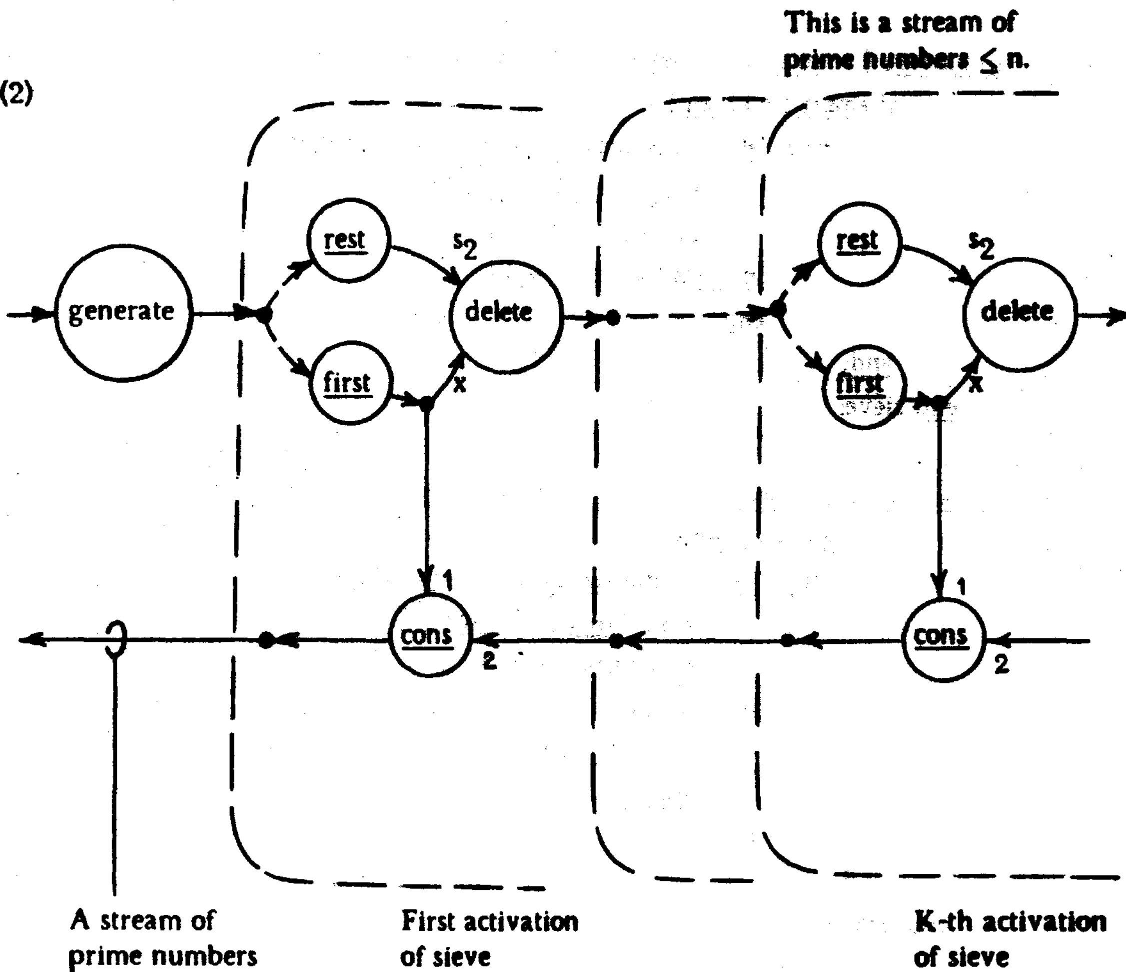


Figure 5.2. A snapshot for the prime number computation

5.2.1 Implementation of stream operations

The implementation presented here is based on translation of each stream operation into one or more data structure operators that include operations on "holes". The notion of holes used here originated in the work of Henderson [Hende75] who used the term "tokens"; and it differs from the notion of suspensions discussed by Friedman and Wise [FreWi78].[1] In this implementation, an empty stream is represented by the nil structure, and a stream s is represented by a data structure whose "first" component is first(s) and whose "rest" component is the data structure representation of rest(s).

The implementation of the stream operations (except cons) is shown in Figure 5.3, and is simply a replacement of a stream operation by a simple data structure operation. The cons actor is implemented by the actors shown in Figure 5.4, where the actors create-hole and write-hole are special data structure operators defined as follows:

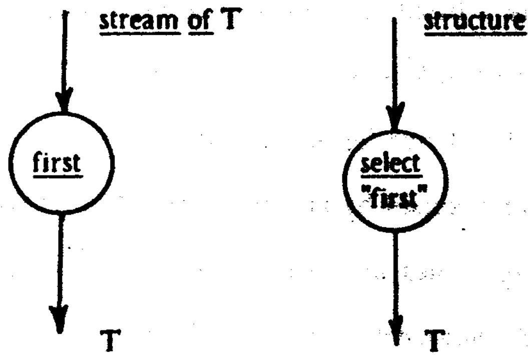
> The output of a create-hole actor is a *unfilled* hole H which is a uid and a *tag* in {filled, unfilled}. The tag of a hole represents its state and affects operations on it: in the unfilled state, all data structure operations on the hole are simply pooled - except the write-hole operation. Upon the completion of the write-hole(H, v) operation, the hole H changes its state to filled and contains the value v; and all previously pooled and subsequent operations are processed without further queuing.[2]

---

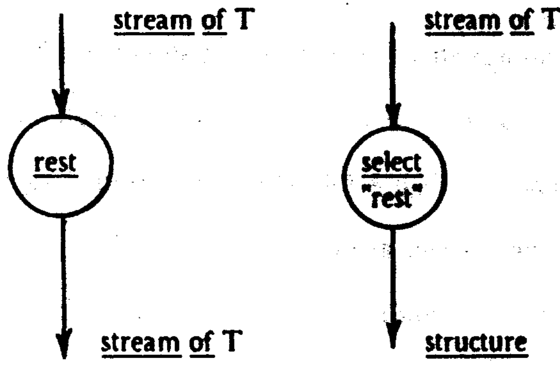
1. The notion of suspension allows one to force the evaluation of some values which is *promised*; and a promised value does not necessarily evaluated as soon as possilbe.

2. The implementation of the write-hole operation must, in addition to writing the value in the address, allow the operations pooled for the hole to proceed. It should be mentioned that the operations on holes are used in a restricted context such that only one write-hole operation is performed on each hole; thus, there is no possibility of race between several write-hole operations on the same hole. A simple way to implement the pooling of operations is to queue them as a list the uid of its head is stored in the hole.

(a) first



(b) rest



(c) empty

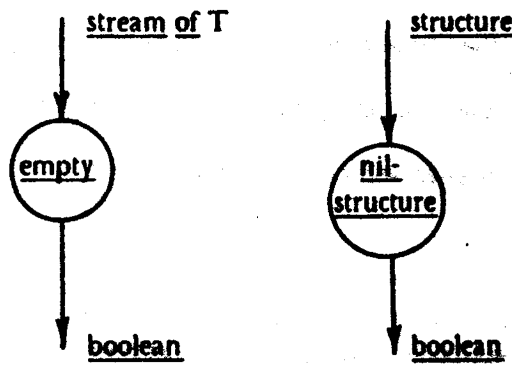


Figure 5.3. Stream Actors (except cons)

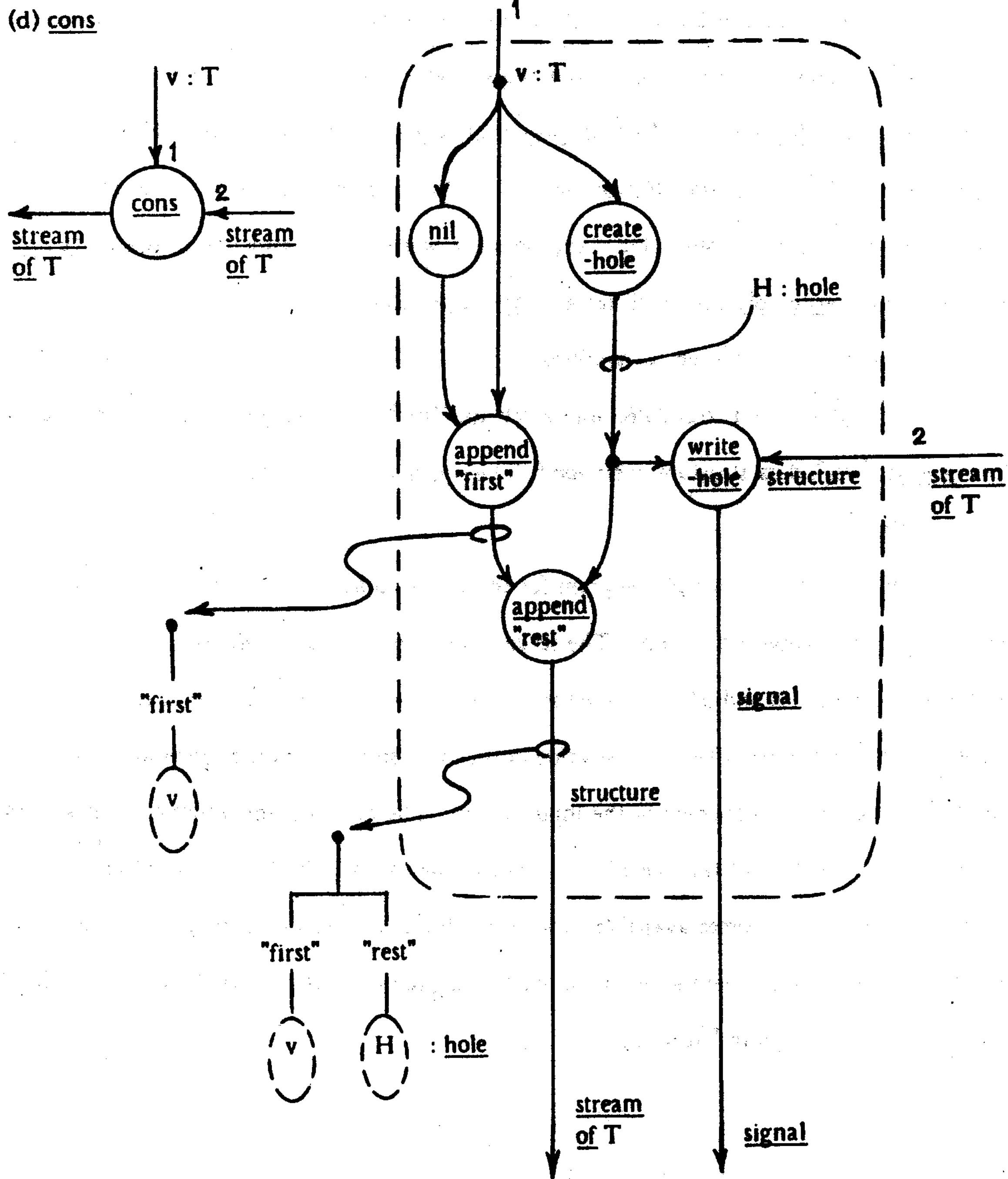


Figure 5.4. Cons

Referring to Figure 5.4, the effect of the cons actor is to construct a data structure whose "first" component is the value v and whose "rest" component is the hole H from the output of the create-hole actor. The write-hole actor receives as inputs the hole H and a data structure representing a stream. Notice that the implementation of the cons actor creates an output after receiving the input value v and does not wait for the completion of the write-hole operation.[1] The write-hole operation has a signal output used for ensuring that the activation is not deleted before its operation is completed.

The first(s) actor is translated into a select(s, "first"), and the rest(s) actor is translated into a select(s, "rest") data structure operation.[2] The empty actor is translated into the predicate nil-structure(s).

Using the earlier example program for the prime number generation, we illustrate the concurrency of operations on streams. The schemas for the two procedures "sieve" and "delete" are shown in Figure 5.5 and 5.6. From the schema for "sieve", it can be seen that the output of the cons actor is generated after the first value in the input stream from the "generator" is made available as the "first" component of the input stream. The second prime number is produced by the second activation of the "sieve" and is not available until the first value of the output stream of the "delete" becomes available. Figure 5.7 shows how various activations of schemas may relate to each other, where we used the notation D_{ij} to denote the j-th activation of "delete" within the i-th activation of "sieve" S_i.

---

1. By making the "first" component a stream, the language could be extended to include stream of < stream type >.
2. Without going further into the details of the implementation of data structures, we simply state the requirement that operations on data structures with holes as components have the property that once the holes are filled, they behave as normal data structures.

```
sieve = procedure ( s : stream of integer )  yields  stream of integer;
            if empty ( s ) then [ ]
             else   let x : integer, s2, s3 : stream of integer;
                        x, s2 = first ( s ), rest ( s );
                        s3 = delete ( x, s2 );
                        in  cons ( x, sieve( s3 ) ) end; end;
      end sieve;
```

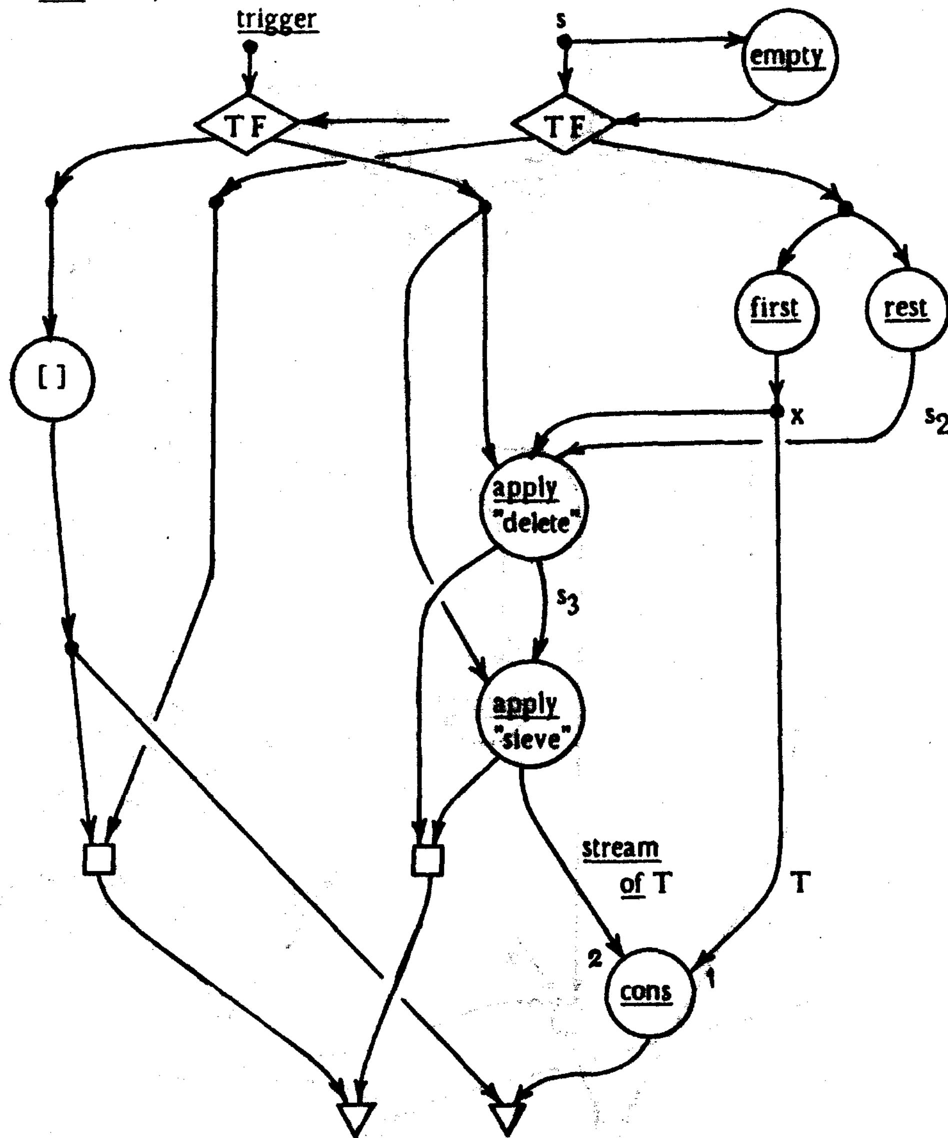


Figure 5.5. Data Flow Schema for "sieve"

```
delete = procedure ( x : integer, s : stream of integer ) yields stream of integer;
        if empty ( s ) then [ ]
            else let  y : integer, s2, s3 : stream of integer;
                      y, s2 = first ( s ), rest ( s );
                      s3 = delete ( x, s2 );
                 in   if divide ( x, y ) then s3 else cons ( y, s3 ) end;
                 end;
        end
    end delete;
```

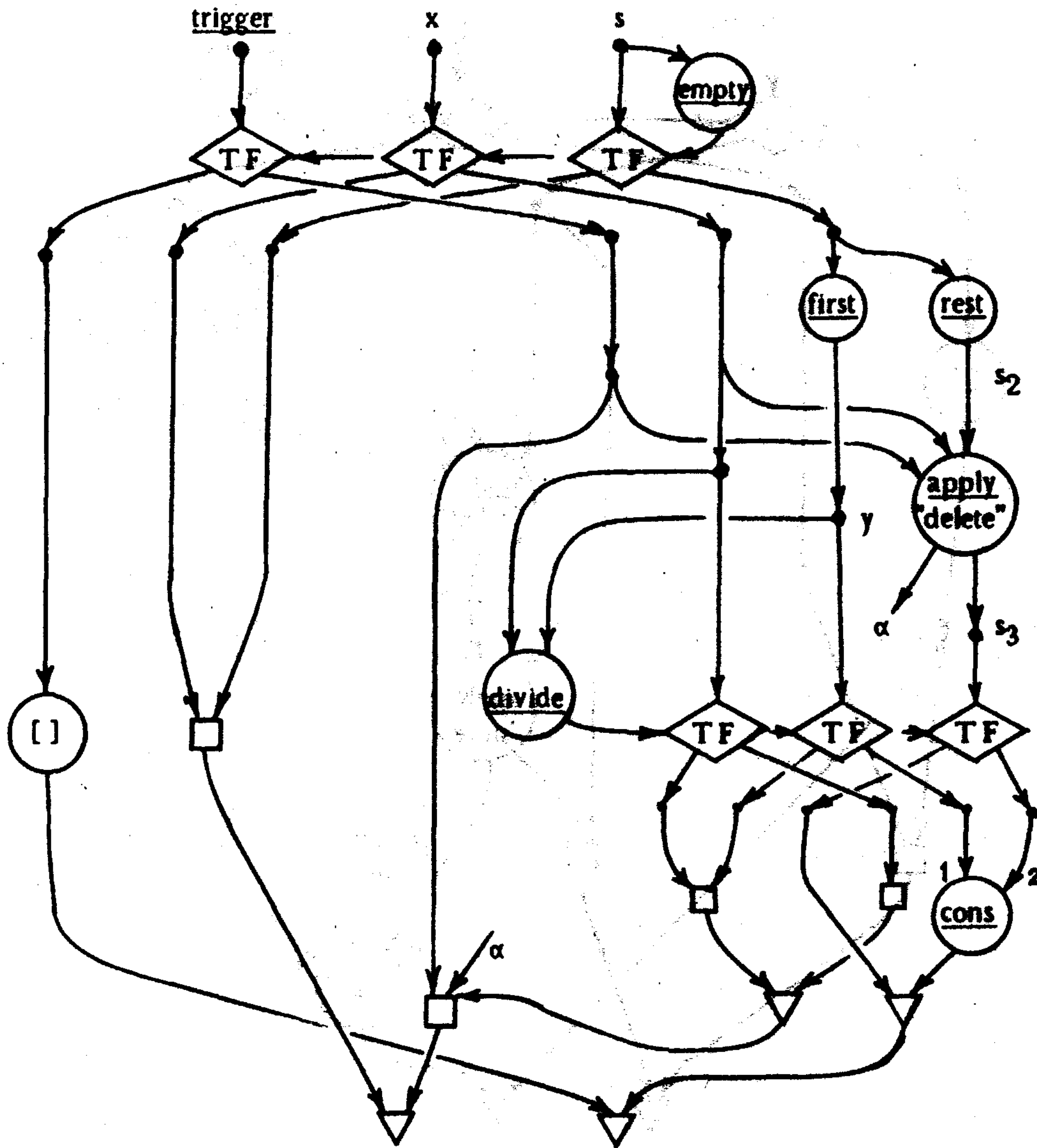


Figure 5.6. Data Flow Schema for "delete"

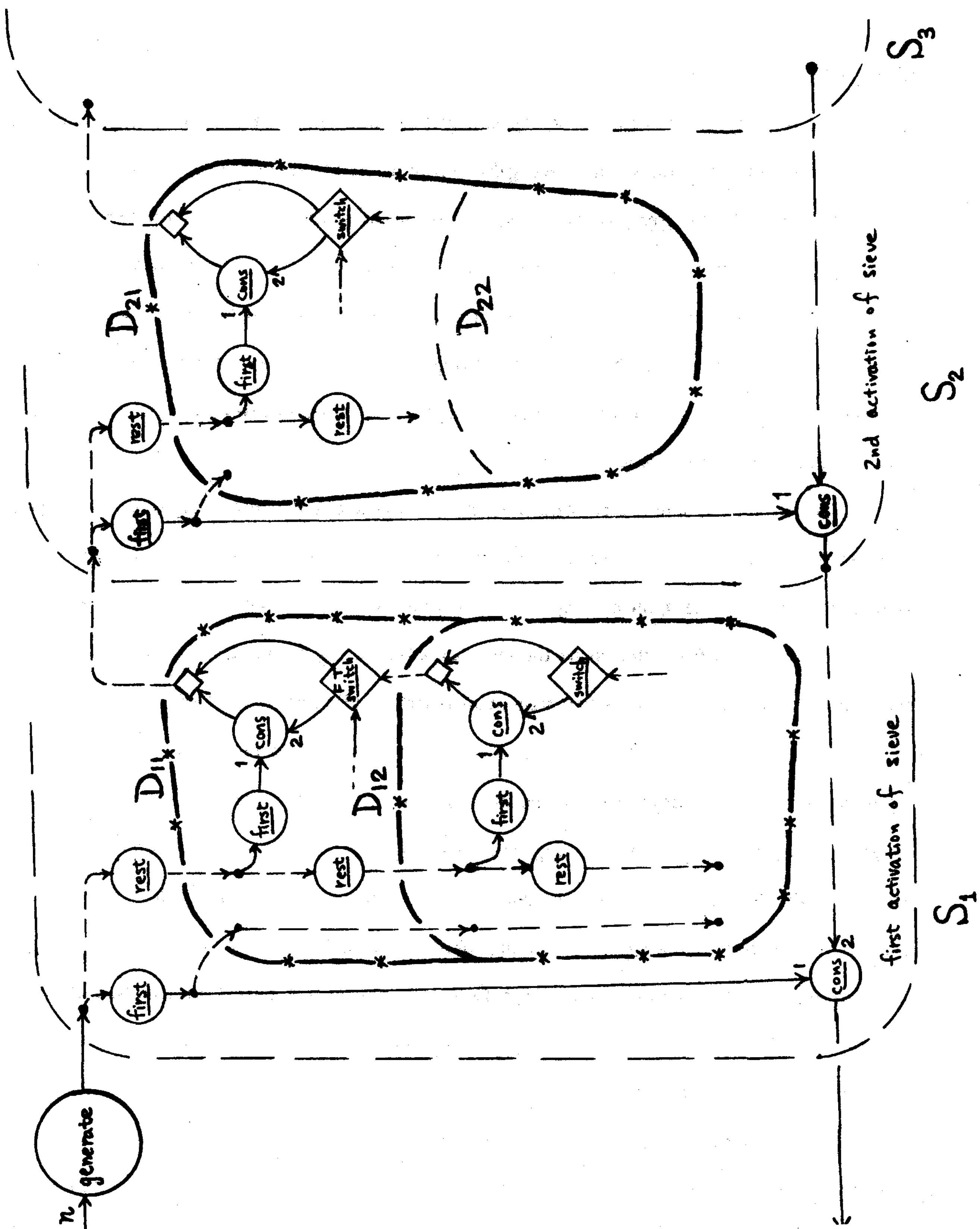


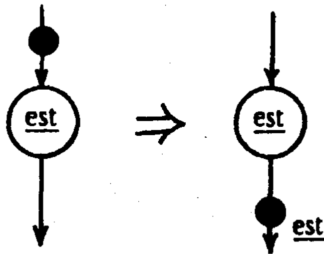
Figure 6.7. A snapshot for the prime number computation

5.2.2 A token passing scheme

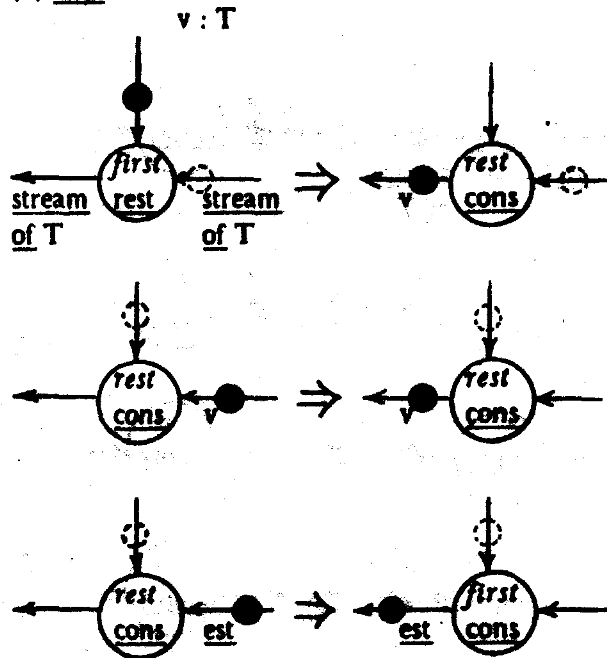
To illustrate the difficulty of implementing streams using "token passing", we introduce a set of data flow actors for streams [Weng75]. These actors are defined over streams in the sense that an arc typed stream carries a sequence of tokens of the same type terminated by a special end_of_stream (or, est) token - hence, the term "token passing". The notation and the operational semantics of data flow actors for stream values are shown in Figure 5.8, where the behavior of each actor is described by a set of firing rules based on the configuration of tokens and the state of the actor. Each actor, except est and st-link actors, has two states *first* and *rest*, and is initially in the *first* state.

An est actor is simply a constant function which generates the special est token. A cons actor enters the *rest* state after placing a token from the first input arc on the output arc, and returns to the *first* state upon passing from the second input arc all tokens ending with an est on its output arc. A first actor enters the *rest* state after placing a token from its input arc on its output arc, and returns to the *first* state upon absorbing all remaining tokens in the stream. A rest actor enters the *rest* state after absorbing the first token, and returns to the *first* state upon passing all remaining tokens in the input stream. An empty actor tests if an stream is empty. In the *first* state, if the arriving token is an est token, the output is true and the actor returns to the *first* state; otherwise, the output is false and it enters the *rest* state. The actor returns to the *first* state after the remaining tokens are absorbed. An st-switch actor takes a boolean input and a stream input, tokens forming a stream are passed to the output arc according to the boolean value. An st-merge simply passes the stream to the output from one of the input arcs. We restrict the use of st-switch and st-merge actors only to the construction of conditional schemas corresponding to the restriction imposed on switch and merge actors presented in Chapter 2. An st-link actor replicates a stream by copying each arriving token and by distributing them to the output arcs. An st-sink is a sink actor for stream values and

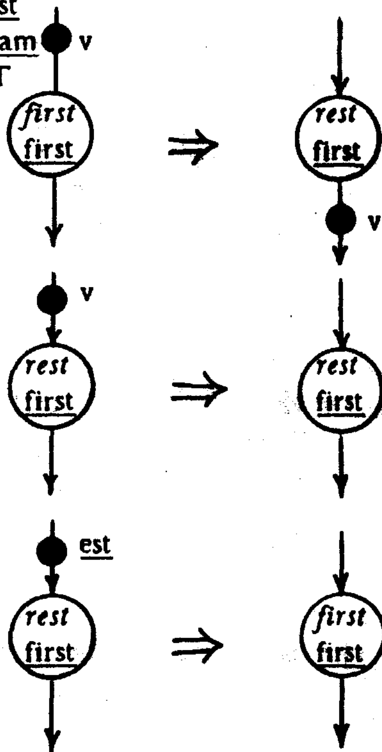
(i) est



(ii) cons



(iii) first
stream
of T



(iv) rest
stream
of T

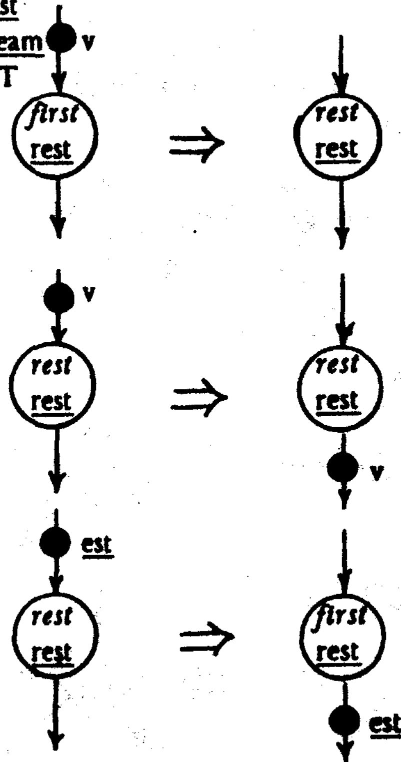
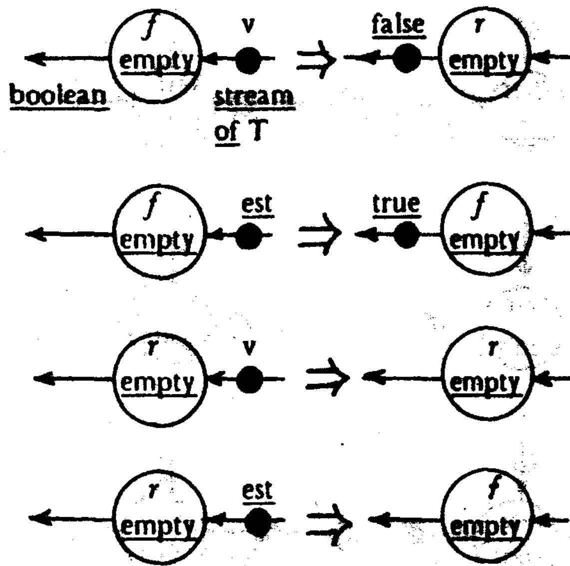
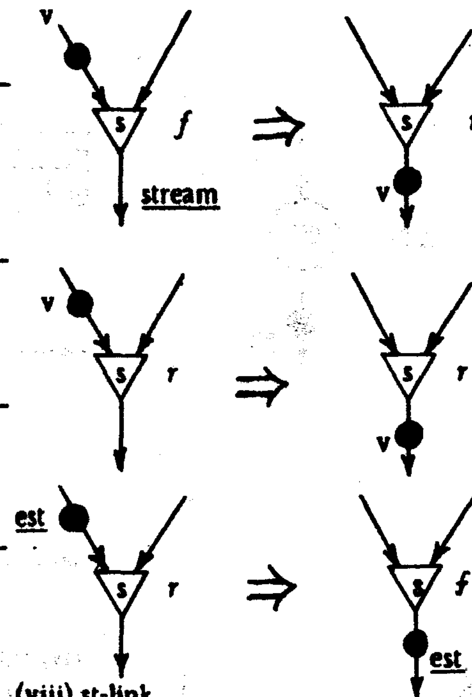


Figure 5.8(a). est, cons, first, and rest

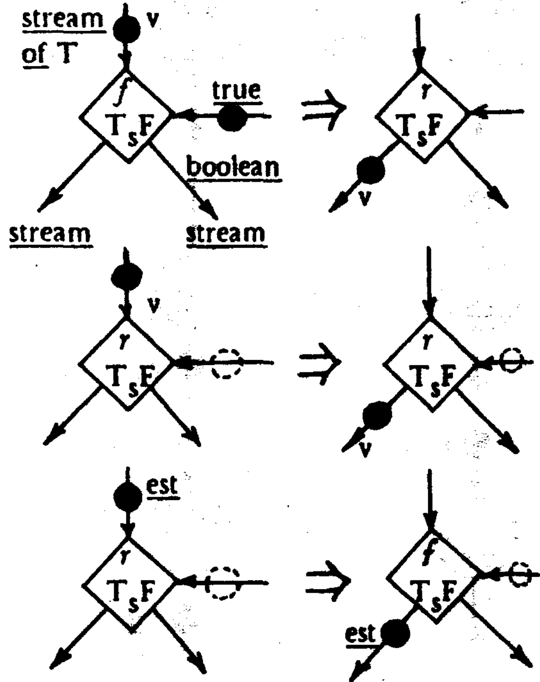
(v) empty



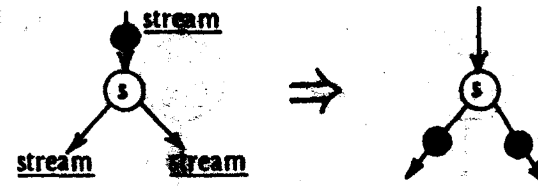
(vii) st-merge



(vi) st-switch (Complementary action occurs when the boolean value is false.)



(viii) st-link



(ix) st-signal

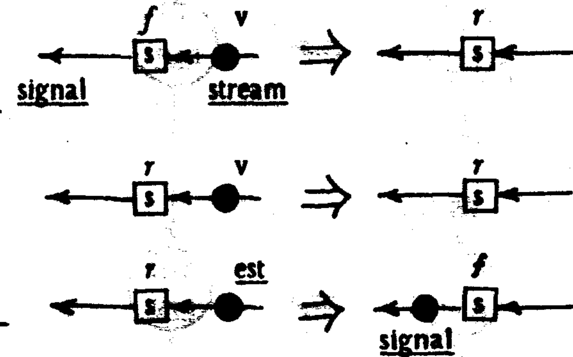


Figure 5.8(b). empty, st-switch, st-merge, st-link, and st-signal

produces a signal to the output arc when an est token is absorbed.

Deadlocks

The set of actors presented above do not implement stream operations correctly, because the substitution of these stream actors for stream operations results in a schema that may deadlock when the predicate of a conditional subschema is an arbitrary expression on streams. This deadlock situation is best illustrated by an example.

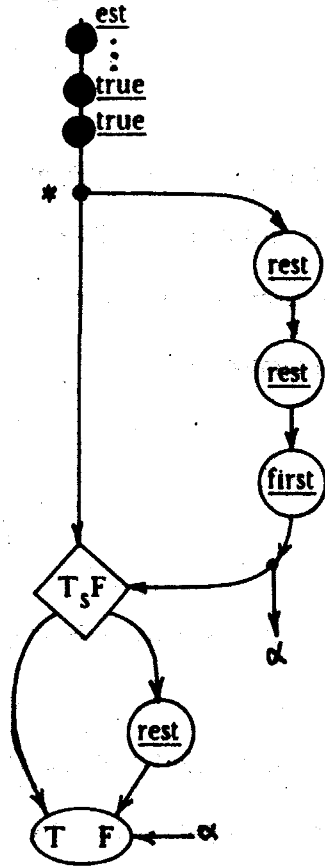
Consider the conditional expression, C:

```
if first( rest( rest( s ) ) )
then  s  else  rest( s )  end;
      where s has the type stream of boolean.
```

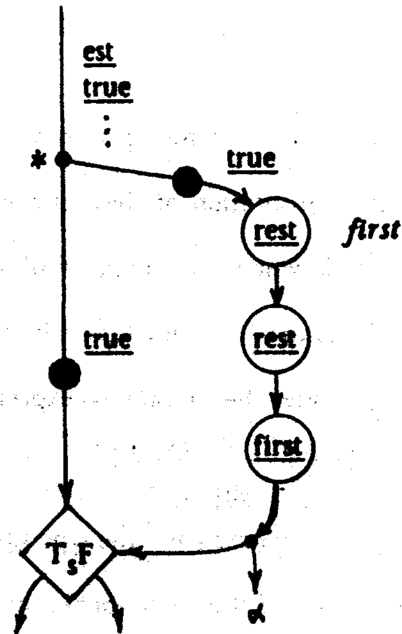
The translation of the conditional statement C yields a conditional schema S shown in Figure 5.9. The predicate of the schema S consists of a chain of stream actors. Execution of the schema for an input stream s = [true, true, ... , true] would deadlock because the input link marked with the symbol ⋆ is prevented from firing by the left output arc holding a token. This situation arises when the predicate controlling the st-switch actor requires an arbitrary number of input tokens to produce the decision outcome. Most predicates, however, can be analyzed at the compile time so that additional link actors are added between the input st-link actor and the st-switch actors to avoid deadlocks.

This example illustrates a very important property: the arcs of the data flow schema are finite buffers. In a computation model that allows infinitely buffered arcs, it can be shown that the history of tokens passing through each arc agrees with the history obtained by the mathematical characterization proposed by Kahn [Kahn74]. For computation models based on arcs of bounded size buffers, the history observed is a prefix of that observable if arcs are unbounded buffers. No mathematical treatment has been found which shows how to derive the exact history for models with finitely buffered arcs. This property of data flow schemas is

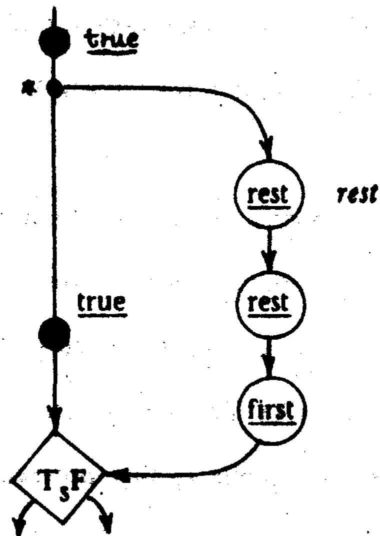
(i) Initial configuration



(ii) After one firing of *



(iii) After the firing of the enabled rest



There is no actor enabled;
notice that the st-link labeled *
is not enabled because the output arc
to the st-switch is occupied by
a token

Figure 5.9. An example of a deadlock situation

undesirable, since the output history would depend on the amount of buffering provided by the number of link actors in a data flow path and cannot be characterized in a clean formal semantics.

Inefficiency

We use the prime number computation presented in Figure 5.2 to illustrate the inefficiency of implementing streams as a sequence of tokens passed along an arc. Referring to Figure 5.2, if we regard each stream operation as a token passing actor, the computation is inefficient, because stream actors form a chain that all tokens in a stream must travel through during the computation. For example, the prime number that is generated by S_i must travel through the chain of (i-1) cons actors to reach the output of the S_1. In fact, the number of firings of a data flow actor to process a stream of length n is proportional to n, and for a chain of n actors it is proportional to n^2 in the worst case.

The rate at which streams are generated or consumed, however, is not necessarily reduced due to this traversal because all tokens can be traversing a chain of stream actors simultaneously if the execution time of all stream operations does not have a large variation. The execution delay caused by the traversal would be much larger if some stream actors in a chain are delayed such that sections of the pipeline containing the stream actors are void of stream elements.

5.3 Forall

In many applications, operations on components of a data structure can be performed concurrently. We present a construct for expressing concurrent computations on arrays. First, we define a data type array of <simple data type>. The form of a forall expression is:

<forall expression> ::= forall <range clause> <eval clause> end;

<range clause> ::= <name> in [<expression$_1$>, <expression$_2$>]

<eval clause> ::= { eval operation <expression> }$^+$

| let {<type decl>}; {<name def>} in <eval clause>;

It is required that <expression$_1$> and <expression$_2$> are of arity one and of type integer. Furthermore, the values lb = <expression$_1$> and ub = <expression$_2$> must satisfy $lb \leq ub$. The expressions in the eval clause can contain references to N, the <name> of the range clause, and must be of arity one. The result of the forall expression is an expression of arity k, where k = the number of eval's in the eval clause. Its j-th value is equivalent to the result of the following expression:

$$E_j(N = lb)\ O_j\ E_j(N = lb+1)\ O_j \ldots O_j\ E_j(N=ub),$$

where O_j and E_j denote the operation and the expression in the j-th eval clause, and the notation $E_j(N = i)$ denotes the j-th expression evaluated using the free variable N with the value i. For the above expression to be well defined, we further require that the operations O_j are binary (requiring two operands) and associative.

Consider the following example:

```
forall i = [5, 100]
eval + A[ i ],
eval * (A[ i ] + B[ i-3 ] - i);
end;
```

The resulting expression is of arity two: the first value is simply the sum of all values A[5],...,A[100] of the array A, and the second value is the product of the expressions A[i]+B[i-3]-i, for i ranging from 5 to 100.

The construct can be easily translated into a recursive procedure as follows:

```
P = procedure(lb, ub, <free-list>) yield R1, . . . ,Rk;
    if ub < lb then undefined, . . . , undefined
    else
            if ub ≤ lb
            then  E1( N=lb ), . . . , Ek( N=lb )
            else let middle : integer,
                    x1 : R1,..., xk : Rk,
                    y1 : R1,..., yk : Rk;
                    middle = ( lb + ub ) / 2;
                    x1,..., xk = P( lb, middle, <free-list>);
                    y1,..., yk = P( middle+1, ub, <free-list>);
                    in  x1 O1 y1, . . . , xk Ok yk;
                    end;
            end;
    end
  end P;
```

where the <free-list> is the list of identifiers (other than the identifier N appearing in the range clause) that are free in each expressions E_j.

It should be noted that the recursive procedure as defined is not the only translation possible, since each recursion can create any fixed number of activations. The translation is only intended to show that the construct can be supported within the framework of our architecture without additional special functional units for dynamic creation of concurrent computations on arrays. It is interesting to observe that similar types of forall expressions cannot be easily defined on data structures that are not arrays. The problem is that we do not have any information about the selector names of a data structure.

5.3.1 Constructing data structures

It is possible to devise a mechanism for defining a more general form of forall expressions on data structures provided that the implementation of data structures is known. As we have mentioned in Section 2.4, a data structure is represented by a collection of items each containing a set of tuples of the form (s, c), where c is either a scalar value or a uid of another item. While it is possible to implement items capable of storing a variable number of tuples, an efficient implementation can be based on storage nodes that can contain only a fixed number of tuples - we shall call these nodes primitive items. In the latter scheme, an item may be represented by more than one primitive item. An example of the representation of an array with primitive items is shown in Figure 5.10. Each primitive item (pitem) consists of two tuples:

{ ("0" : C_1) ("1" : C_2) },

where C_1 and C_2 are either scalar values, uid's of other pitem, or nil's. The example is an array A such that:

A[4] = 2,
A[2] = 3, and
A[i] = nil, for all other i from 0 to 6.

In this representation, the traversal from the root node A to a leaf node defines an ordering from less to more significant bits of the binary representation of the index to the array A. Using this representation, we show two ways of constructing an array using the forall construct.

We define an associative operation construct for constructing an array from two arrays. This operation is defined only when indices of non-nil elements of the two arrays are disjoint. This is satisfied when construct is used within the forall construct in a fashion such that the condition for disjoint indices can be determined at compile time. The construct operation is defined recursively as:

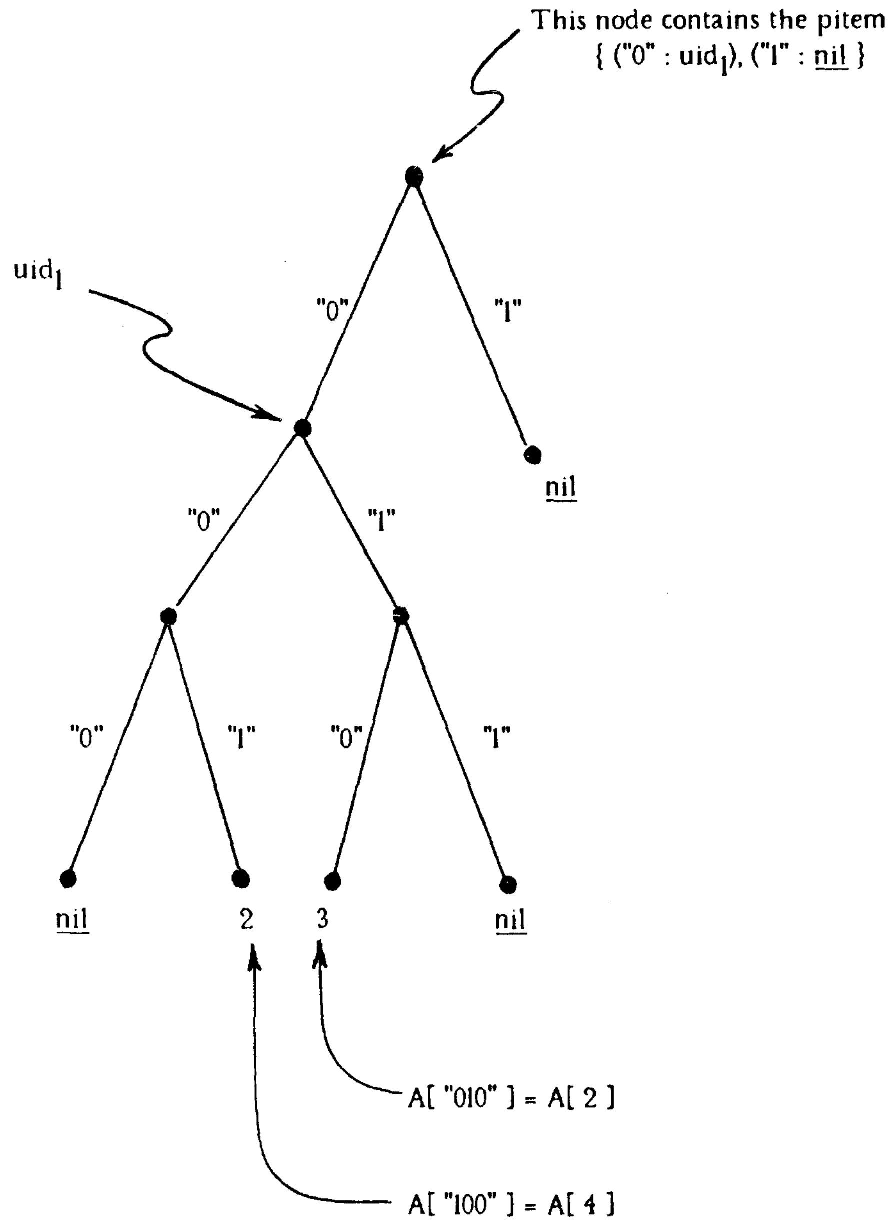


Figure 5.10. An array representation

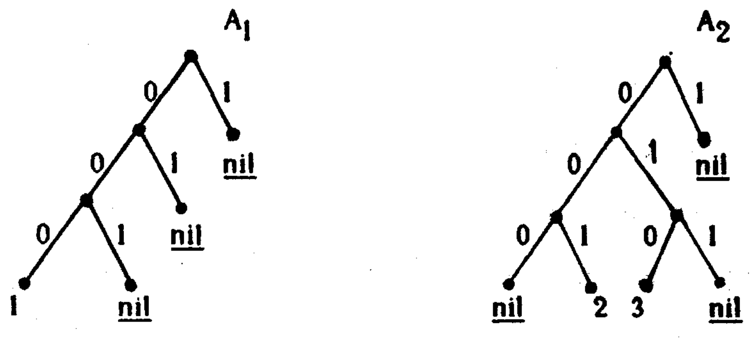
```
construct = procedure( A, B ) yields structure;
        if nil( A ) then B
        else if nil( B ) then A
             else if scalar( A ) and scalar( B ) then error
                  else if scalar( A ) then migrate( A, B)
                       else if scalar( B ) then migrate( B, A)
                            else let  C0 = construct( A•"0", B•"0" );
                                      C1 = construct( A•"1", B•"1" );
                                 in make-hole( make-pitem( C0, C1 ) )
                                 end;
        end; end; end; end; end;
    end construct;
```

The operation • is defined such that the result of A•"0" returns s, where A is a pitem containing {("0" : s), ("1" : t)}. The result of make-pitem(C_1, C_2) is a pitem {("0" : C_1), ("1" : C_2)}. The function of the make-hole(x) operation is to create a hole H which is returned as the output of the construct and which is later filled with the item x. The procedure "migrate"(A, B) takes a scalar value A and stores it into the leftmost available component of B whose selector is formed by a sequence of bits "0". Figure 5.11 illustrates the manner in which the result of construct(A_1, A_2) is created.

An example of the use of construct in a forall construct is:

```
forall i in [5, 100]
eval + A[ i ] + B[ i ],
eval construct  append( nil, i+1,
                          if i = 5 then A[ i ] + A[ i+1 ]
                             else if i = 100 then A[ i-1 ] + A[ i ]
                             else A[ i-1 ] + A[ i ] + A[ i+1 ]
                          end; end;)
end;
```

Notice in this example that the resulting array contains indices in the range [6, 101]. In general, the expression for the selector inside the append must be restricted to simple expressions to



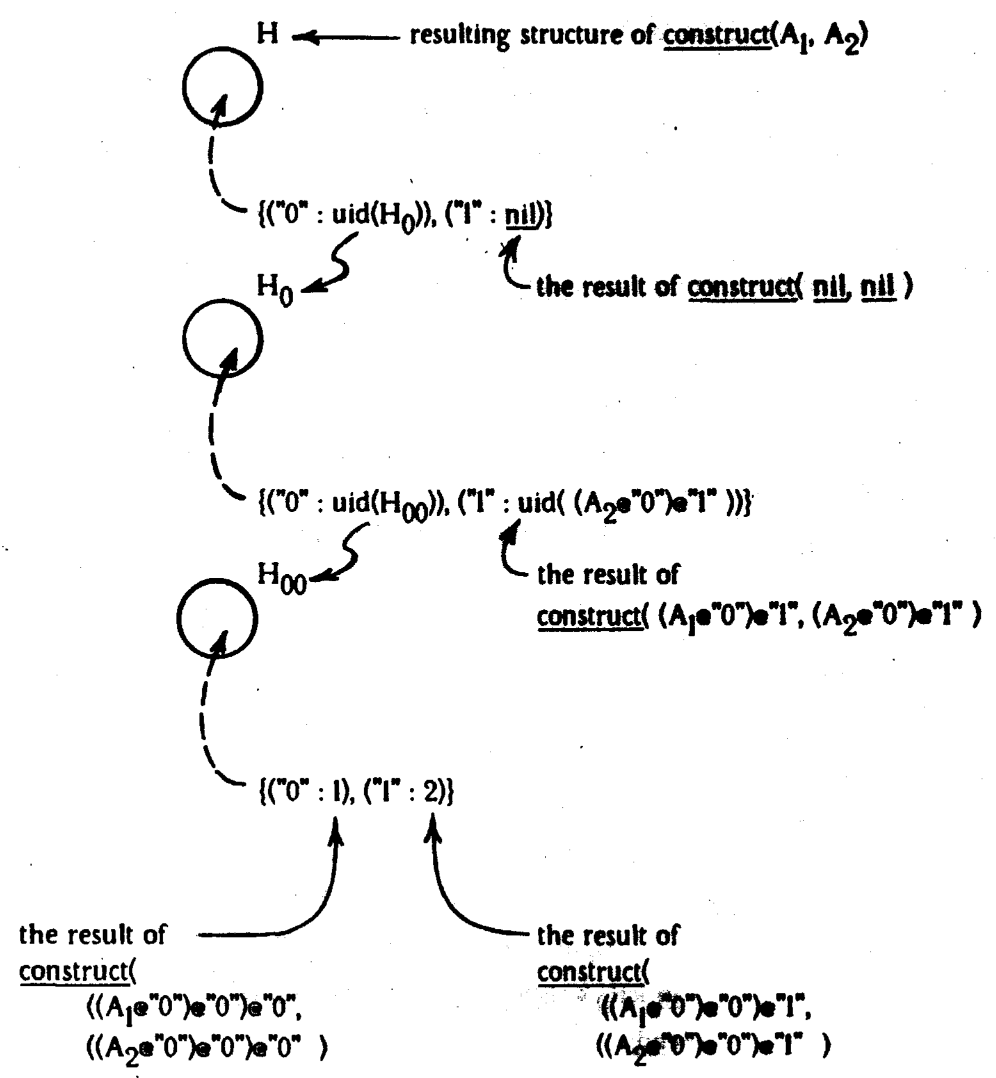


Figure 5.11. An example for the working of construct

guarantee the disjointness of the indeces;[1] and we leave this as an issue for language design.

In the forall construct as presented, the range clause may only be integers. This is undesirable in cases where the range is much larger than the number of data elements in the array, because the number of activations created would be much larger than the number of elements in the array. We introduce another form of specifications of the range clause:

<name> in range of A,

where <name> is an identifier that can range through all one level indices of the array A; thus, the range clause is not usable for specifying compound selectors as the indices [i, j] of a two dimensional array.

An example of its use in a forall is:

```
forall i in range of A
eval * A[ i ] + A[ i+1 ],
eval construct append( nil, i, A[ i ] + B[ i ] )
end;
```

The above forall expression can be translated into the following call to the recursive procedure P:

```
P(A, A, B, nil);
where
      P = procedure( a, A, B, i) yields integer, array;
            if nil( a ) then 1, nil
                  else if scalar( a ) then A[ i ] + A[ i+1 ], A[ i ] + B[ i ]
                  else let     left0, left1 = P( a•"0", A, B, "0"•i );
                               right0, right1 = P( a•"1", A, B, "1"•i )
                        in left0 + right0, make-pitem( left1, right1 )
                        end;
```

---

1. If the expression is an arbitrary function on i, then there is no simple compile time check for this condition. One must define the semantics of data structures very carefully, if any expression is allowed.

end;

end P;

The result of the expression "0"*i is a concatenation of two bit strings such that, if i="001", then the result is "0001".[1] The procedure P works by "tracing" down the array A for each primitive item a and by creating recursive procedures for the components a@"0" and a@"1" of the primitive item. The construction of the resulting array C by using the make-pitem is possible because the selector in the append expression is of the simple form i; if not, the expression

make-pitem($left_1$, $right_1$)

must be replaced by

construct($left_1$, $right_1$),

and the expression

A[i] + B[i]

must be replaced by

append(nil, exp, A[i] + B[i]).

The reader can verify that the number of procedure activations created is the number of the leaf nodes of an array representation. A further step for optimization is possible for the above example: notice that the value of the expression A[i] equal a when the predicate scalar(a) is true. Thus, there is a significant amount of compile time analysis involved for translating the forall construct into the procedure P. We note that the above translation together with the optimization can result in significantly efficient programs.

The two forall translation schemes presented provide more expressiveness for the language but are dependent on the representation of data structures. Further extensions for allowing the range clause to include data structures in general can be envisioned. In particular

---

1. We will assume that the representation of such bit strings is not difficult.

the latter form of range clause can be readily extended to data structures.

5.4 Nondeterminate merge of streams

In this section we introduce a primitive that can be used to produce a stream by nondeterminately merging two streams. We believe this primitive may be used successfully in building well structured programs. Often, nondeterminacy in a computation can be expressed using arbitration among streams of values, and procedures that operates on the resulting streams. (It is not clear that there are not form of nondeterminate computation that have only awkward realization in terms of streams, and this is an area for further research.) The particular implementation of the nondeterminate merge of streams is in terms of a recursive procedure and is reasonably efficient.

A primitive nondeterminate merge actor, n-merge actor shown in Figure 5.12 has two inputs I_1 and I_2, three outputs O_1, O_2 and O_3, and has two states *first* and *second*. In the *first* state, an n-merge actor can fire as soon as an input token arrives at either one of the input arcs I_1 or I_2. Upon firing, it places the input token on O_1; and, on the second output arc O_2, it places an integer i if I_i is the input arc having received the token. After the firing, it enters the state *second* to expect another token. In this state, the second token is simply absorbed and a signal is placed on O_3; and the actor returns to the *first* state. If two tokens arrive simultaneously, then: one token is selected and placed on O_1; an integer indicating this selection is placed on O_2; a signal is placed on O_3; and the discriminated token is simply absorbed. We show a correct implementation of the n-merge in Appendix A.[1]

---

1. Since the firing rule depends on the timing of the arrival of input tokens, an Execution Controller must implement this critical region correctly. Furthermore, the n-merge actor requires two firings, and the implementation must be consistent with the instruction execution scheme described in section 4.2.

(a) Firing rules for n-merge

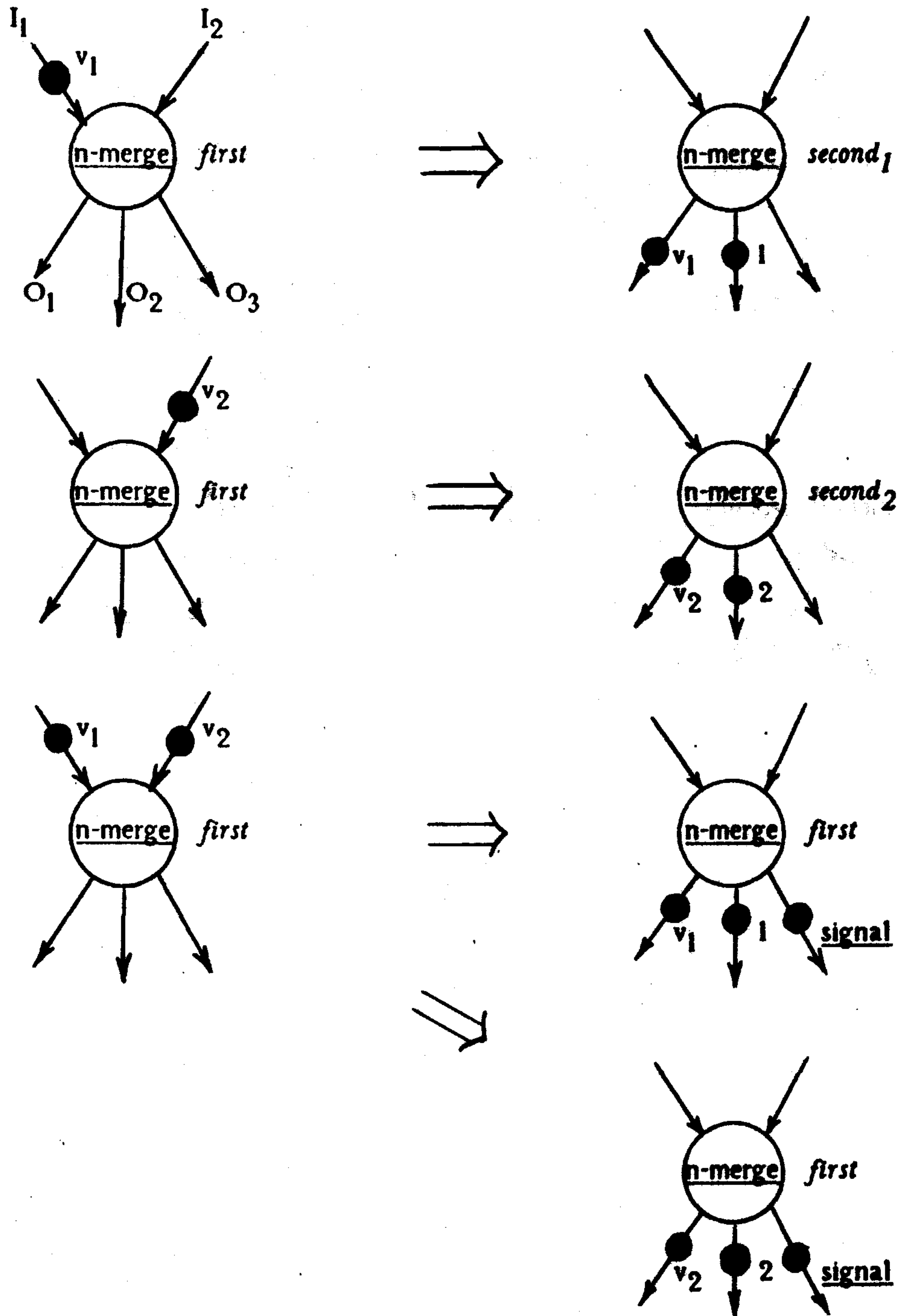


Figure 5.12(a) Firing rules for n-merge

(b) Firing rules for n-merge

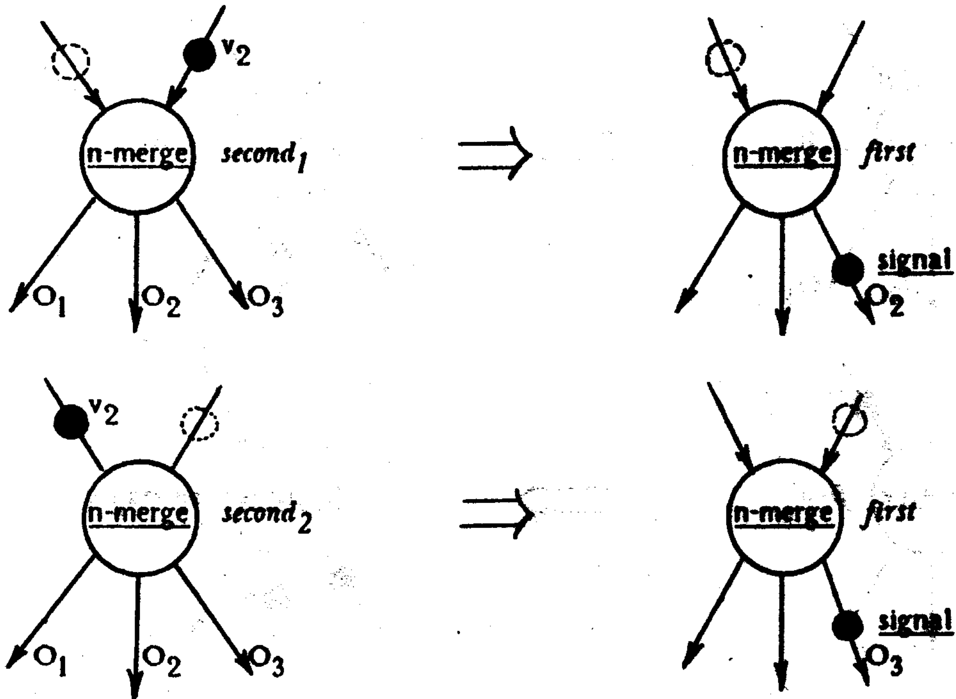


Figure 5.12(b) Firing rules for n-merge

The recursive procedure "N-Merge" in Figure 5.13 defines a nondeterminate merging of two input streams using the n-merge actor. Each activation of the recursive procedure obtains the first elements of streams S_1 and S_2 and merges the two values nondeterminately with an n-merge. The first arriving value is cons'ed to the recursive call on the other stream and the rest of the arriving stream. This recursive definition performs the merging of two streams at the expense of some redundancy in the number of first operations on the streams to be merged, since the slower of the arriving values first(S_1) and first(S_2) at the n-merge actor is discarded and the subsequent recursive activation also performs a first operation on the slower stream value. Thus, the number of first operations on two input streams of length n and m is bounded above by $2(n + m)$. Another problem of the recursive N-Merge is that the number of activations is about the same as the number of operations waiting for stream values which have not been generated.[1] It is possible to remove these inefficiencies by introducing a set of data flow actors connected in a cyclic fashion (see Appendix B). Unless the inefficiency of the recursive definition is severe, the cyclic definition is unnecessary.

5.5 Discussion

There are a number of extensions that are convenient for writing procedures on streams. In many situations we find it necessary to generate a stream of values with a base value followed by values of some constant increment. This stream value can be simply expressed as:

[base by increment until final_value].

---

1. Notice, however, that the cost of keeping these activations active is relatively little, since only a very small number of operand records would reside in the system. But this situation can be intolerable when one of the streams is never generated or gets arbitrary behind the other.

```
N-Merge = procedure ( S1, S2 : stream of T ) yields stream of T;

          let   x, i = n-merge( first( S1 ), first( S2 ));
                Y1, Y2 = rest( S1 ), rest( S2 )
          in if i = 2
                then if undefined( x ) then S1 else cons( x, N-Merge( S1, Y2 )) end;
                else if undefined( x ) then S2 else cons( x, N-Merge( Y1, S2 )) end;
                end;
          end N-Merge;
```

Figure 5.13 A recursive nondeterminate merging of two streams

Conversion between an array and a stream is also often necessary.

A more important language problem, however, is whether data types stream of <stream type> are needed. The implementation described in Section 5.2 naturally extends to stream of <stream type>. It is not clear, however, that such extensions are of significance to expressing concurrent operations on streams. From the point of view of defining formal semantics for the language, it is much cleaner to have data types stream of stream, or array of stream.

We give an example for illustrating the expressiveness of stream of stream. In performing computations on arrays it is often useful to have the type stream of stream. The program in Figure 5.14 is often referred to as a "hyperplane" computation on arrays. Figure 5.15 is a diagrammatic explanation of the manner in which the computation "Hyper" is performed. The top horizontal array C corresponds to the stream C, and the left vertical array B' corresponds to the stream B. In the lower right quadrant bounded by the two arrays C' and B', the two dimensional array D' corresponds to the output of the procedure "Hyper". Each point on a row of D' is computed using the procedure "Compute" by taking the west, the north-west, and the north neighbors of the point. The value of the point is computed by applying the function "Neighbor" on the values of its neighbor. The dotted lines show how points of the array D' (or the stream of stream D) are produced as the computation proceeds.

In this example, the amount of concurrency is at most the number of elements in the stream B, but this concurrency is not achievable if the computation is expressed with arrays.

Extensions of the language to include other forms of nondeterminate primitives are of critical significance. Can streams be used to implement language primitives similar to the monitor [Hoar72]? We leave this as a further research issue.

```
Hyper = procedure( B, C : stream of integer)  yields  stream of stream of integer;

        if empty( B ) then  []
            else let    b : integer,  D : stream;
                        b = first( B );
                        D = Compute( C, b );
                in  cons( D, Hyper( rest( B ), cons( b, D ) ))
                end;
        end;
    end Hyper;

Compute = procedure( C : stream of integer,  b : integer )  yields  stream of integer;
        if  empty( C ) then [ ]
            else  if  empty( rest( C ) ) then  [ ]
                    else  let   d : integer;
                                d = Neighbor( b, first( C ), rest( C ) );
                        in  cons( d, Compute( rest( C ), d ))
                        end;
                end;
        end;
    end Compute;
```

Figure 5.14 An example using stream of stream

Array C'

C'[0] C'[1] C'[2]

B'[1]

Compute(C', B'[1])

B'[2]

Compute(D'[1,*], B'[2])

Array B'

B'[3]

Compute(D'[2,*], B'[3])

S_1

S_2

D'[5, 3]

Array D'[row, column]

S_3

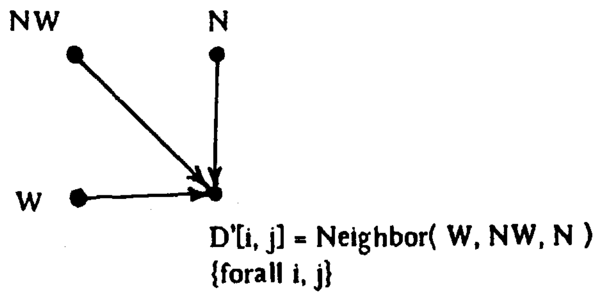


Figure 5.15. An illustration of a hyperplane computation

This page intentionally left blank.

Chapter 6. Supporting Data Structures and Activation Records

In this chapter we state several requirements for designing the Packet Memory to support the structures used to implement the language. The Packet Memory stores three types of objects: data structures (including procedure structures), activation records, and holes. We propose the implementation of all objects is based on allocation of items which are of fixed size. Based on this design decision, we show how operations on these objects can be implemented efficiently. Since the design of the Packet Memory has been pursued previously by [Denns75, Acker77], we will not treat the Packet Memory in great detail. What concerns us is the manner in which the Packet Memory must be used to correctly implement the objects. Functionally, the Packet Memory maintains a pool of uid's for free items. Each item contains a fixed number of tuples (s, c), where s is a selector name of some predefined size and c is either a scalar or the uid of an item. For brevity, we will often use the word "item" to mean the content of the item and/or its uid.

We discuss how these objects can be efficiently implemented in a Pakcet Memory organization that has multiport and multicache memory. Of particular interest in this organization is the cache organization which achieves concurrency of simultaneous access to an item; and this organization may be applicable to other concurrent systems.

6.1 Packet Memory

The organization among the Packet Memory, Structure Controller and Execution Controller is shown in Figure 6.1. The Structure Controller receives data structure operation packets from the Arbitration network and sends result packets to the Distribution Network. The hole-operation output port of the Structure Controller is connected to an input port of the Arbitration Network. (This connection is not shown in Figure 4.1 of Chapter 4.) The function

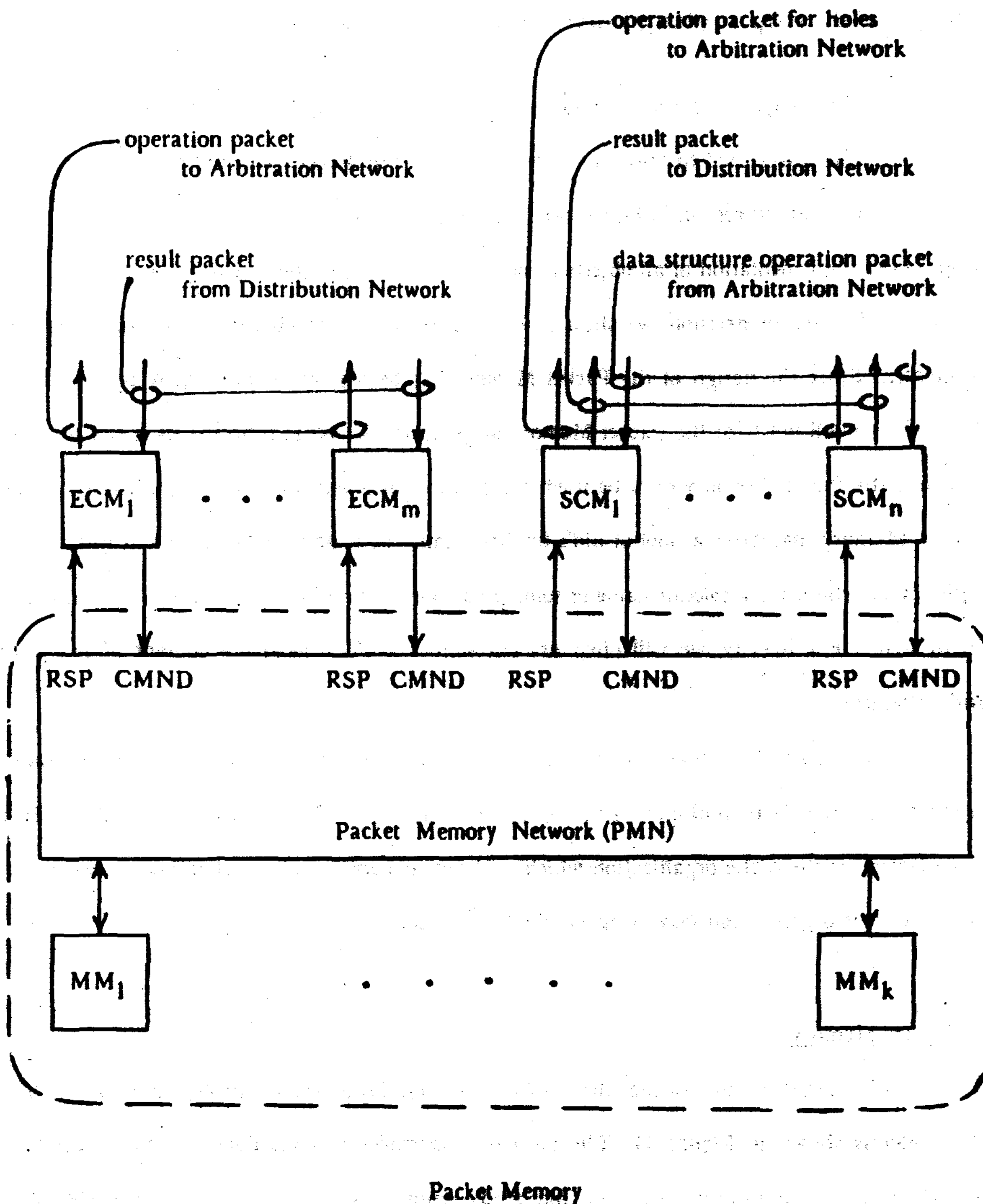


Figure 6.1. Organization among SC, EC, and PM

of the port is explained in Section 6.2. Each Structure Controller Module (SCM) and Execution Controller Module (ECM) is connected to the Packet Memory via a Command (CMND) port and a Response (RSP) port. A Command port receives commands on an item specified by its uid, and the response is eventually returned to the Response port associated with the Command port. The types of commands include reading an item, writing an item, requesting a free uid, and changing the reference count of an item. These commands are issued by both ECM's and SCM's, and processing of a result packet or a data structure operation packet may require more than one commands.

The Packet Memory consists of a Packet Memory Network (PMN) and a set of Memory Modules (MM). The PMN is a packet routing network whose nodes may be cache modules (CM) that have cache memory for frequently accessed items and necessary control functions for management of the cache. One approach for generating unique identifiers is to let a uid be an address from the physical address space formed by storage nodes of the lowest level of the memory hierarchy of the Packet Memory.[1] For example, using current technology, the physical address space would consist of all addresses of secondary on-line storage devices such as disks. Each storage module in higher levels of the hierarchy acts as a cache, and in general each entry in such a storage module must contain both the data of the item and its full physical address (i.e. its uid). Many techniques can be applied to the design of caches for finding an item: for instance, searching(possibly including tree search techniques), hashing, or hardware

1. Another method for generating unique identifiers is to use counters that are never reset, or are reset very infrequently. Our approach is shared by Snyder's work [Synde79] on architectures for object-oriented languages like CLU [Lisko78]. The main reasons for not choosing the counter scheme are that it requires the lowest level memory to store both the uid of an item and the data and that accessing an item can be prohibitively expensive if search needs be conducted at the lowest level of the hierarchy. We should remark that the efficiency arguments presented here may not be justified considering the projected technological developments and increasing sophistication of storage devices.

associative matching. The criteria for placement and replacement of an item in a cache is not of central issue to us here, but a possible candidate is Least Recently Used (LRU) replacement algorithm that has proven attractive for demand paging memory management. For further study, we refer readers to: [Acker77] for details of a possible implementation of the Packet Memory including the design of CM's; [Smith78] for set associative memory organization, and [Denng70] for a general discussion on paging systems.

Assuming that each Memory Module stores a distinct subset of the total uid's, a basic design consideration is the manner in which an item can be moved or copied in PMN. Informally, we say a caching scheme is a "unique access" scheme if, for each item, the set of reachable caches from CMND ports to a MM forms a linear path; otherwise, it is called "multi-access" if the set forms paths containing branches. Figure 6.2(a) illustrates a unique access structure where the network routes command packets for the same item from any command port to the same cache module, and Figure 6.2(b) and (c) illustrate two multi-access structures. It is often possible that a multi-access caching structure behaves like a unique access structure when used in a restricted manner. For example, when commands on an item are always presented at the same input port of the cache structure shown in Figure 6.2(b), the only caches reachable from the port to the MM associated with the item forms a linear path. The structure in Figure 6.2(c) does not have this property because the set of caches on the paths from the input port I_1 to the memory module MM_8 does not form a linear path.

For PMN, we expect its caching structure to belong to the class exemplified by the structure in Figure 6.2(b). We classify items into two classes *restricted* and *unrestricted* according to *how they are used*. We do not statically partition all items into two classes because it is desirable to be able to use a free item in either manner and because the distribution of their usage is not a parameter that we can determine safely. Using this classification, we describe the manner in which an item is handled by the cache structure of the Packet Memory.

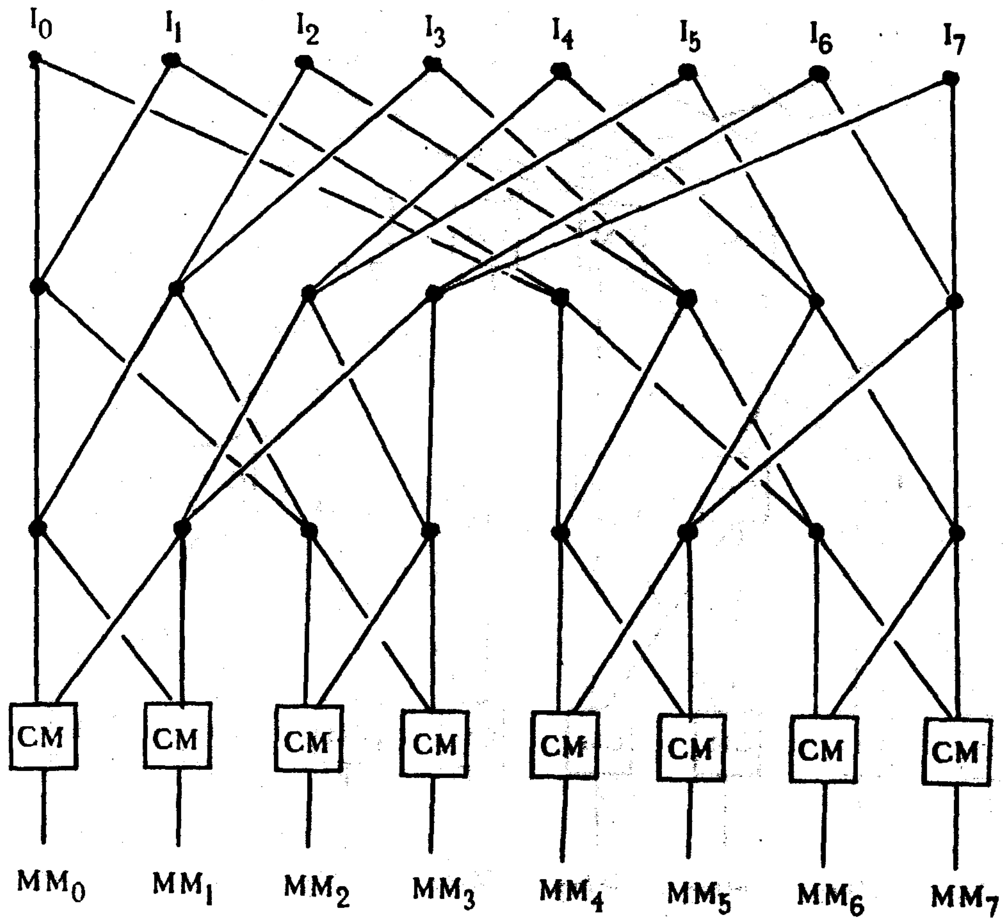


Figure 6.2(a). A unique-access Packet Memory Network

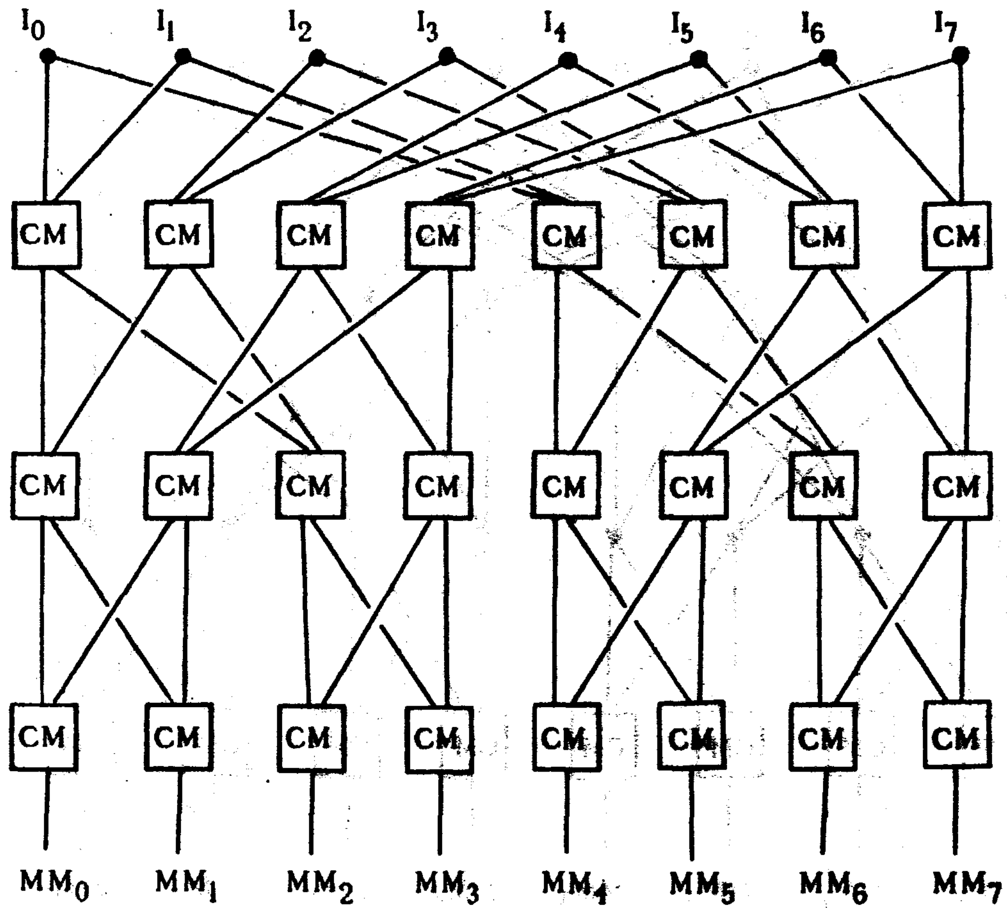


Figure 6.2(b). A multi-access Packet Memory Network with unique-access property

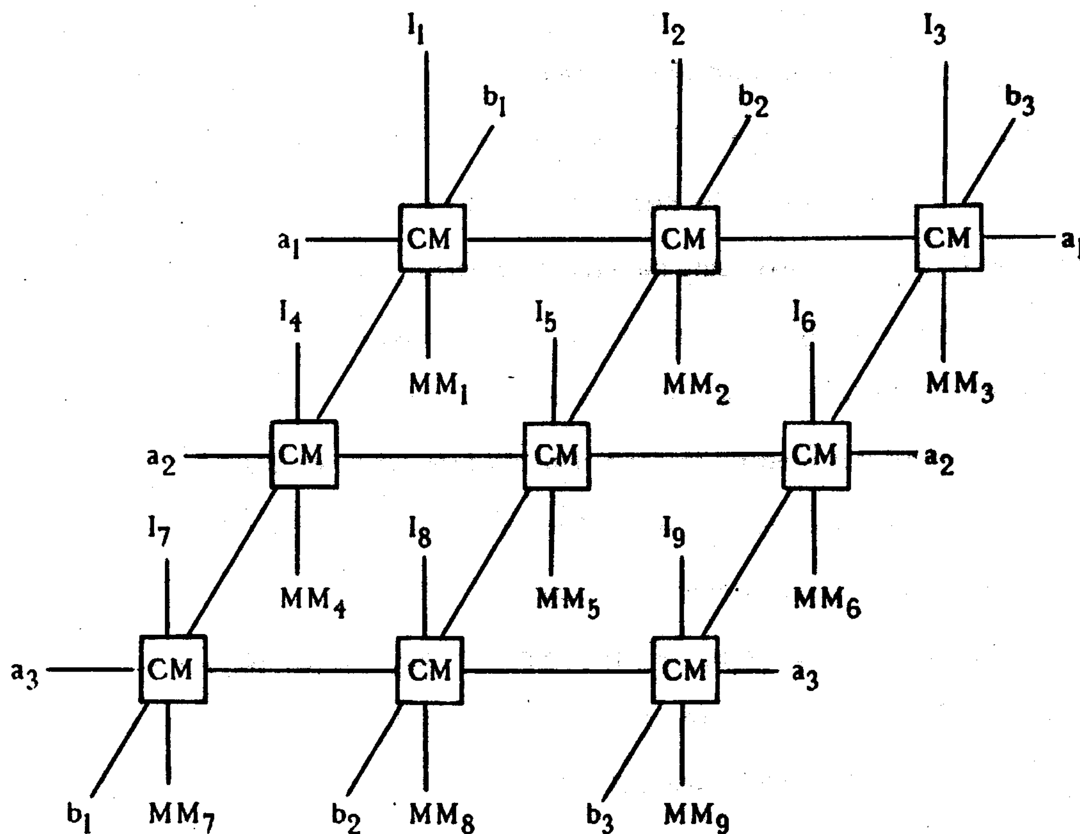


Figure 6.2(c). A multi-access Packet Memory Network

Since a restricted item is accessed only through a particular CMND port[1] and all commands result in memory references along the unique path, there is no need to have several copies of the item. A restricted item is, therefore, moved along the caches on the path rather than copied. The first use of a free restricted item is a writing command to a CMND port which creates an instance of the item. Subsequent commands on the item must be from the same CMND port and may cause the item to be moved into a cache at a higher level of the PMN hierarchy. Such items have a nice property that they can be updated without the consistency problem of multiple copies in several caches (or, the multi-cache coherence problem). A consequence of this property is that a restricted item can be garbage collected as soon as its reference count becomes zero. As we shall see in Section 6.2, we use this property of restricted items to implement activation records and holes.

For unrestricted items, we allow copies to exist in several CM's to provide the opportunity for alleviating contention over a single copy of the item by storing several instances of the item in different caches. We shall call such copies *instances* of an item. Initially, an item must be written by a command from some CMND port. This command must write through all caches leading to a unique memory module MM from which all higher level caches can access the item. The command does not acknowledge completion of the operation until this write-through operation is completed. Subsequent commands on the item may cause instances of the item to stored in caches of higher level and operations are performed on them. It is evident that it is possible to have inconsistent instances if the content of an unrestricted item can be updated. Therefore, we require that all subsequent operations on unrestricted items are

1. The particular port for accessing an item is fixed over the lifetime of an item - i.e. from its removal from the free uid port until it is garbage collected again - but need not be the same in different lifetimes for the cache structure shown in Figure 6.2(b).

commands on reference counts or for reading the item. This requirement is naturally satisfied by the semantics of the language whose data structure operations are free of side-effects. We now present a scheme by which an item can be garbage collected correctly. This garbage collection scheme is correct only when the set of caches reachable for an item forms a tree-like structure with the MM as its root such as the structure shown in Figure 6.2(b). Furthermore, no garbage collection is performed on copies in the PMN.

Each instance of an unrestricted item contains a *copy count* indicating how many copies have been made directly from it. Each time an item is copied from one cache to another, the reference count and the copy count of the new instance is set to zero, and the copy count of the source instance are incremented by one. Upon completion of copying, commands can be exercised on the new instance. If an instance is displaced from a cache, its reference count is added to the reference count of the source instance whose copy count is then decremented by one. We require that an instance is displaced from a cache only if its copy count is zero, this ensures that all existing instances form a properly connected tree and that only instances at the leaf nodes are displaced. For all instances created by the initial write-through, except the one in MM, reference counts will be zero, copy counts will be one. The instance in MM contains a reference count of one, and a copy count of one; and possibly a tag identifying it as the root node instance.

This scheme allows an inaccessible item to be garbage collected eventually as the result of merging instances of inaccessible items displaced from caches. That the reference count of the final unique instance is correct can be seen by noticing: the correct reference count is the sum of all reference counts, some negative, of all instances; and the strict displacement algorithm and the tree-like access paths ensure that the copy count of the unique instance is zero if and only if all reference counts have been accumulated. The garbage collection on an item takes place if the reference count and the copy count of the root node instance are found to be zero.

The scheme can be very slow in reclaiming inaccessible items if some instance is not displaced from a cache. This situation could be a problem if free items in the Packet Memory are in short supply and the the system is in a state such that instantes are not displaced from caches due to lack of movements of items in the Packet Memory. This situation, however, would not arise frequently in a well designed Packet Memory.

6.2 Activation records and holes

We implement activation records and holes with restricted items because efficient implementation of these objects requires updating the contents of items. Operations on restricted items are handled differently in implementing these objects for efficiency. The *lifetime* of an item is defined from its removal from a free list to the next time it is placed on a possibly different free list. If an item is used by an ECM as a part of an activation record, then all subsequent commands are guaranteed to be issued by the same ECM. But if an item is used as a hole, during its lifetime, its uid can be sent to different ECM's or SCM's. Thus, there must be a way to guarantee all commands are received by the same CMND port. Conceptually, the CMND port can be different over different lifetimes. But this is difficult to implement, since all ECM's and SCM's must somehow know the different CMND ports designated to different lifetimes of an item. The simplest way to ensure that all ECM's and SCM's send commands on an item to the same CMND port is to assign the CMND port statically using some function F from all uid's to CMND port identifiers. We elaborate on this when we discuss an implementation of holes.

6.2.1 Activation records

An activation record is a dynamic tree-like structure representing an array such that an operand record for an instruction instance (A, i) can be reached from the root node item A by

accessing a set of items using the binary bit representation of the selector i. Each item may contain an operand record, or either one or both tuples in { ("0" : α_0), ("1" : α_1) }, where α_0 and α_1 are uid's. We envision that an operand record can be stored in an item since we can make all actors have a small number of input and output arcs.

Initially, an activation record consists only of the root node A with a single component "text".[1] The Distribution Network routes a result packet (A, i; k, v, count) to an Execution Control Module $ECM_{H(A)}$ determined by some hash function H from uid's to indices of ECM's. The arrival of the result packet modifies the activation structure A using the bit string representation of i by accessing all items until the operand record is found. If the operand record is not in the activation record, the last item on the path of access is modified to include the necessary items by acquiring more free restricted items. Thus, the first arriving operand always results in allocation of free items, and subsequent arrival of operands to the same operand record simply modifies the existing operand record.

We now present how reference counts can be used to manage items in an activation record:

(a) <u>create-activation</u>(P)

This operation creates an activation record A whose reference count is one and the reference counts of items leading to the "text" component are set to one. The leaf item has the uid of the procedure structure P.

(b) <u>insert</u>(A, i, v)

This operation adds one to reference counts of all items leading from the root node A to the operand record (A, i).

---

1. We assume that the selector "text" can be encoded as a binary bit string without conflicting with integers used for instruction numbers.

(c) remove(A, i)

This operation is performed by an SCM when it finds an instruction is enabled after an insert operation. The operation decrements all reference counts of items leading to (A, i) by the value of count.

(d) free(A)

This decrements the reference count of the root node of the activation record by one - thus, allowing it and the "text" component to be garbage collected.

The scheme maintains the reference count of an item such that it is equal to the number of arrived operands in operand records which are waiting for enabling and can be reached from the item.

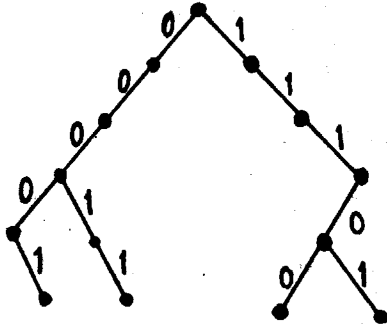
The presentation has been made based on the assumption that a selector name used in each item is a single binary digit "0" or "1". This makes operations on activation records easier to understand, but introduces an apparent inefficiency that many items are required to encode the instruction number i. Since an activation record is likely to be sparse most of the time, it is possible to reduce the number of items used to represent the sparse structure by using prefix compression. An example of such a representation of is shown in Figure 6.3. This added saving on usage of items results in faster instruction execution on the average. While this representation using prefix compression requires a more complex update operations on items, we feel the complexity is justified considering the cost of accessing an items.

Similarly, we believe prefix compression can be applied profitably to the representation of data structures in general.

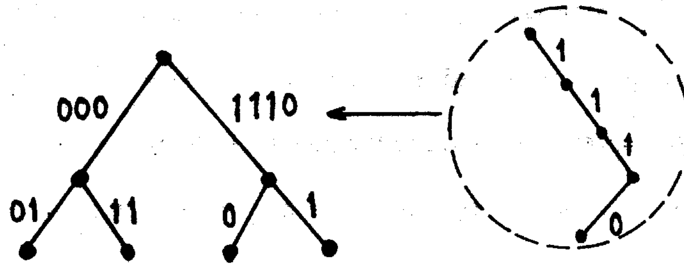
6.2.2 Holes

The create-hole operation simply obtains and tags "unfilled" into an item; and the uid is marked as a "hole" and returned as its result. If the hole is in the "unfilled" state, data

(a) An activation record not using prefix compression



(b) An activation record using prefix compression



After inserting (A, "10111")

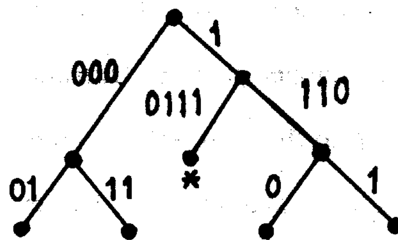


Figure 6.3. An example of prefix compression

structure operations or commands[1] on a hole that require reading its data are simply stored as a pool of items storing these operations. Commands such as reference count updates need not be stored since they do not need to use the data of the hole. Since a hole may occur as a component of a data structure, a Structure Controller may encounter a hole when processing a data structure operation packet. The hole-operation output port allows a SCM to send a data structure operation packet through the Arbitration Network to a specific SCM associated with the $CMND_{F(uid)}$ port. To guarantee this, the design of the Arbitration Network is much simplified if the function F is implemented in the routing algorithm.

The reference count processing for restricted items used for holes is the same as reference count accounting for items used in data structures. Note that operations pooled for a hole should not change the reference count of the item until the hole is filled. This avoids the potential problem that the reference count of a hole may become zero before these operations are processed.

6.3 Remarks

We have informally discussed how activation records and holes can be implemented using restricted items. This is based on the assumption that the Distribution Network must route all result packets with the destination (A, i) to the same ECM. Thus, all operations on restricted items used in the activation record A are guaranteed to be sent to the same CMND port. Using this representation, then, a natural optimization is to allocate an activation record "close" to the procedure structure or its copies in caches. Similar optimization is possible for data structure operations if the Arbitration Network can try to route most data structure

---

1. We do no mean commands only here, because holes could be used to hold part of data structures on which we want to further perform data structure operations.

operation packets on an item to the same SCM if the contention for the same SCM is not severe. This optimization will tend to make effective use of the cache memory bandwidth by allowing a higher hit rate on the item.

The question of how far this optimization based on locality of data access should go depends on the understanding of program behavior and is a challenging issue. On the other hand, for a large procedure, it may create more enabled instructions than a single ECM can handle; in this case, a different approach for storing activation records may be devised that allows an activation record to be distributed over several ECM's.

This page intentionally left blank.

Chapter 7. Conclusion

Summary

The expressiveness of a programming language affects not only programming tasks but also how the underlying architecture can attain high performance through concurrent operation of hardware. We feel that a language based on an applicative style of programming is sufficiently expressive for most applications and, augmented with additional features, can provide an approach for structured concurrent programming. That an applicative style of programming is preferred is based on the observation that unexpected side-effects greatly compromise the confidence in correctness of programs. For applications requiring high performance systems, data flow analysis must be performed on programs to reveal the hidden concurrency and this analysis is more complicated than necessary because of language features based on sequential notion of execution. In this regard, APL has been suggested as a language for vector and array processors, because it is more amenable to such analysis. APL, however, is limited in its expressiveness because data structures presented in Chapter Two of this thesis cannot be easily mapped into arrays. Concurrency is expressed in several ways in the value-oriented language that we introduced. Procedure activations allow many activations to be simultaneously executed. Streams can be used to express concurrency in computations with a strict ordering on accessing sequences of values. The forall constructs are for explicitly specifying concurrent operation on data structures, particularly arrays.

The implementation of streams can be readily extended to stream of stream and is based on the notion of "holes". Two forms of forall constructs have been defined and can be used to express computations on components of data structures using associative operations. Concurrency expressed in these constructs derives from the property of associativity of operations on components of data structures.

To show how concurrency in computation can be exploited, we used recursive data flow schemas into which a program in the language can be translated. We proposed an extended form of data flow processor that implements recursive data flow schemas using procedure structures and activation records. These objects are supported by the Packet Memory with a multiport and multicache storage structure. A solution is given to the problem of maintaining the consistency of reference counts used for memory management; and this allows simultaneous accesses to multiple instances of a data structure. We suggested in Chapter Two a *split-reference-weight* scheme of memory management that removes the need for reference count updates for each data structure operation. This scheme is of particular interest when a data structure is frequently copied as it is the case in forall's.

Data flow architectures differ from conventional concurrent systems particularly because concurrency at primitive operation level is easily achieved; and the difficulty of process switching in conventional multiprocessor organizations can be avoided.

Suggestions for further research

We first discuss language issues: the generality of streams and data structures whose components may be all holes; cycles in data structures and in communication paths between processes; and nondeterminacy. We then discuss architecture issues.

Streams and data structures with holes

The concept of streams can be captured in terms of lists, queues, and arrays which are accessed in a constrained manner. Streams provide a reasonable abstraction for expressing concurrency among cooperating computations, but it requires some degree of adjustment to think in terms of sequences of values. Since the manner in which accesses to structures are constrained may not be immediately obvious to a casual user, it may not be easy to see when the

notion of stream is applicable. We see many computations, such as the hyperplane computation illustrated in Chapter Five, where concurrency is substantially improved if we expressed programs using streams. But as the reader may note, it is easier to understand the recurrence equation for the computation than to understand the lengthy program using stream of stream. Should we provide a compile time translation for such equations? How general can such translators be?

If we allow data structures which are accessible when they do not have all of its components, do we need streams? The author's opinion is that streams can be defined in terms of a recursive data type which can be accessed when some of its components may not be available -- using holes. But does use of such data structures cause undesirable situations to arise? One can conceive of a situation where the Packet Memory is overloaded with references made to components which do not exist yet. How often do theses situations arise? Can one control such situations?

Another issue relates to the general question of defining semantics of aggregates of data values such as data structures, streams, and a list of expressions. In this thesis, we assumed that all computation terminates and errors in the constituents of an aggregate do not imply the error of the whole aggregate. In this view it is desirable that we can define a consistent way of dealing with nonterminating computations which supply the component values. In general, it may be required to determine when the output value of a nonterminating process is not needed so a computation can be forcibly terminated to avoid wasting computing resources. This can be done either continually, periodically or only when resources become scarce. One scheme of garbage collecting unwanted processes continually has been proposed by Baker [Baker78]. Can and should the scheme be applied to the data flow concept of computation?

Cyclic data structures and communication among processes

The need for cyclic data structures and cyclic communication paths between processes are actually two separate issues.

The need for some representation of conceptual cycles in representation of objects is undeniable. But how are such conceptual structures mapped into data structures whose operations have no side-effects? Consider the example of a doubly linked list L from which we need to delete a node N. There are two ways to represent the list without side-effects: by using immutable cyclic structures based on Henderson's work [Hende75], or by using an acyclic structure. In the scheme using immutable cycles, a delete operation requires about the same number of operations as the number of nodes in the list L, because a new cyclic structure must be constructed to avoid side-effects. Thus the physical resemblance of the immutable cyclic structure to conceptual cycles does not imply the conceptual simplicity of delete operations on such a cycle. For the scheme using acyclic structures, one can see that a delete operation now can be performed as a data structure operation which roughly costs log(n) operations on items, where n is the number of nodes in the list L. This observation can be extended to operations on graphs of other forms.

The implementation of procedures as values is related to data structures with cycles when we need a mechanism to construct a procedure from existing ones using *binding* of procedure names to its representation [Hende75]. Using immutable cycles to represent recursive procedures seems natural in that there is no need to introduce the notion of environments in the definition of procedural values. But the operations involving cyclic structures of procedure representations will have the same problem as we have discussed previously.

Many forms of programs are more naturally expressed as a set of processes communicating amongst themselves using cyclic communication paths. Examples are often seen in various distributed message passing systems. Constructs of this form are not included in this

thesis, because we have not found one that allows deadlock property to be determined at compile time. It may be possible, however, to provide deadlock detection mechanisms at runtime. If the mechanism does not introduce too much overhead for computations that do not deadlock, such an approach may be desirable. In addition, it may also detect deadlocks due to resource allocation. Much work has been done for deadlock detection of processes due to resource allocations. Not much work, however, can be found in the area of detection of processes which are in deadlocks due to either synchronization or message handling. We hope further work in this area provides additional insights to the complexity of these deadlock detection schemes.

Nondeterminacy

In large systems such as data base systems, operating systems, real time control systems, and point of sale systems, the function of the systems is not necessarily determinate. Often, an implementation of such systems must allow some degree of nondeterminacy and possibly tolerate temporary inconsistency in their data base to achieve a reasonable performance criteria. The nondeterminate merge function that we have introduced in this thesis is inadequate for expressing many such forms of nondeterminacy.

Architecture

In the architecture we presented, the performance is derived from concurrency on a large scale. We made no assumptions about how concurrent operations can be mapped into Execution Controllers such that two instructions are located in some close neighborhood to reduce communication delays -- thus improving its performance.

Is it possible that heuristics for allocating instructions close to each other can degrade the potential performance of the processor due to bad allocation strategies? (Such processors must have functional units close to the Execution Controller Modules and the network

structures may be quite different.) It is hard to evaluated these suggestions without understanding both the behavior of programs and the technology of the hardware modules. This issue is important because the cost of communication hardware is determined by assumptions about locality of computation.

The issue of fault-tolerance must be adequately answered for a system such as our data flow processor which has a large number of modules. We emphasize that when we are dealing with a faulty system some additional operating system functions for handling faults may be needed.

Ideally, we hope that a system based on data flow concepts can support a community of users with the performance that concurrent operation can provide. Such a system necessarily must provide a set of programming languages and various input and output functions. In addition, it must provide reasonable mechanisms for controlling total activities in the system such that finite computing resources can be used effectively. In conventional systems these functions are supported by software and explicit machine level primitives for controlling processors. How these functions can be provided on data flow processors is a very interesting research issue.

Bibliography

[Acker77] Ackerman, W. B., "A Structure Memory for Data Flow Computers," LCS-TR-186, M.I.T., Sept.

[AckDe79] Ackerman, W. B., J. B. Dennis, "VAL Reference Manual," Computation Structure Group, Lab. for Computer Science, M.I.T., Camb., Mass., 1979.

[Adams68] Adams, D. A., "A Computation Model With Data Flow Sequencing," School of Humanities and Sciences (Technical Report CS-117), Stanford University, Stanford, Calif., Dec. 1968.

[AhHoU75] Aho, Hopcroft, and Ullman, The Design and Analysis of Computer Algorithms, Pub. Addison Wesley, 1975.

[ArGoP77] Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, Monitors, and Dataflow," The Sixth ACM Symposium on Operating Systems Principles, Nov. 1977.

[ArvGo77] Arvind, and K. P. Gostelow, "Some Relationships between Asynchronous Interpreters of a Data Flow Language," Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, August 1977.

[ArvGo77] Arvind, and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Proceedings of IFIP Congress 1977, August 1977.

[Backu78] Backus, J., "Can Programming Be Liberated from the Von Neumann Style? *A Functional Style and Its Algegra of Programming*," Comm. of ACM, Vol. 21, No. 8, August 1978.

[BaBoE70] Baer, J. L., D. P. Bovet, and G. Estrin, "Legality and Other Properties of Graph Models of Computations," Journal of the ACM, Vol. 17, No. 3, July 1970.

[Bahrs74] Bahrs, A., "Operation Patterns," Lecture Notes in Computer Science 5, Springer-Verlag, New York 1974.

[BakHe77] Baker, H. G. Jr., and C. Hewitt, "The Incremental Garbage Collection of Processes," ACM SIGART-SIGPLAN Symposium, Roch. N.Y., Aug. 1977.

[Barn*68] Barnes, G., R. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes, "The ILLIAC IV Computer," IEEE Trans. on Computers, C-17-8, August 1968.

[Batch74] Batcher, K. E., "STARAN Parallel Processor System Hardware," 1974 NCC, AFIPS Conf. Proc., Vol. 43, pp405-410.

[Berkl75] Berklin, K. J., "Reduction Languages for Reduction Machines," Proceedings of the Second Annual Symposium on Computer Architecture, Jan. 1975, pp133-140.

[Bisho77] Bishop, P. B., "Computer Systems with a Very Large Address Space and Garbage Collection," Ph.D. Thesis, Dept. of EECS, M.I.T., also LCS-TR-178, M.I.T.

[Brock78] Brock, J. D., "Operational Semantics of a Data Flow Language," TM-120 Laboratory of Computer Science, MIT, December 1978.

[Burge75] Burge, W. H., "Stream Processing Functions," IBM Journal of Research and Development, Vol. 19, No. 1, Jan. 1975, pp12-25.

[Conwa63] Conway, M. E., "Design of a Separable Transition-Diagram Compiler," Comm. of the ACM, Vol. 6, No. 7, July 1963.

[Davis78] Davis, A. L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," Proc. of the Fifth Annual Symposium on Computer Architecture, Computer Architecture News 6, 7(April 1978), 210-215.

[Denns72] Dennis, J. B., "On the Design and Specification of a Common Base Language," MAC-TR-101, 1972, M.I.T.

[Denns74] Dennis, J. B., "First Version of a Data Flow Procedure Language," Lecture Notes in Computer Science, 19 (G. Goos and J. Hartmanis, Eds.), Springer-Verlag, N. Y., 1974, pp 362-376.

[Denns75] Dennis, J. B., "Packet Communication Architecture," Proceedings of the 1975 Sagamore Computer Conference on Parallel Computation.

[DenFo73] Dennis, J. B., and J. B. Fosseen, "Introduction to Data Flow Schemas," Computation Structure Group Memo 81-1, Lab. for Computer Science, M.I.T., Cambridge Mass., Sept. 1973.

[DenMi75] Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," The Second Annual Symposium on Computer Architecture: Conference Proceedings, January 1975.

[DenWe7] Dennis, J. B., and K.-S. Weng, "Application of Data Flow Computation to the Weather Problem," Proceedings of the Symposium on High Speed Computer and Algorithm Organization, April 1977.

[Dijks68] Dijkstra, E. W., "The Structure of THE-Multiple System," Comm. of the ACM, Vol. 11, No. 5, May 1968.

[Dijks75] Dijkstra, E. W., "Guarded Commands, Nondeterminacy and Formal Definition of Programs," Comm. of the ACM, Vol. 18, No. 8, Aug. 1975.

[Ellis74] Ellis, D., "Semantics of Data Structures and References," MAC-TR-134, 1974, M.I.T.

[Enslo77] Enslow, P.H. Jr., "Multiprocessor Organization - A Survey," ACM Computing Surveys, Vol. 9, No. 1, March 1977.

[Flynn72] Flynn, M. J., "Some Computer Organization and Their Effectiveness," IEEE Trans. Computers C-21, 9, September 1972.

[Fosse72] Fosseen, J. B., "Representation of Algorithms by Maximally Parallel Schemata," S. M. Thesis, Dept. of E.E.C.S., M.I.T., Camb., Mass. 1972.

[FriWi76] Friedman, D. P., and D. S. Wise, "The Impact of Applicative Programming on Multiprocessing," Proc. of the 1976 International Conference on Parallel Processing, Aug. 1976.

[FriWi78] Friedman, D. P., and D. S. Wise, "Aspects of Applicative Programming for Multiprocessing," IEEE Trans. on Comp. Vol. C-27, No. 4, April 1978.

[GurWa77] Gurd, J., I. Watson, "A Multilayered Data Flow Computer Architecture," Proc. of the 1977 International Conference on Parallel Processing, Aug. 1977.

[Hende75] Henderson, D. A., "The Binding Model: A Semantic Base for Modular Programming Systems," Lab. for Comp. Sci. TR-145, Feb. 1975. M.I.T., Camb., Mass.

[Hintz72] Hintz, R. G., and D. P. Tate, "Control Data Star-100 Processor Design," Proceedings of CompCon 72, IEEE Computer Society Conf. 1972, IEEE Press.

[Hoare74] Hoare, C. A. R., "Monitors: An Operating System Structuring Concept," Comm. of the ACM, Vol. 17, No. 10, Oct. 1974.

[Hoare78] Hoare, C. A. R., "Communicating Sequential Processes," Comm. of the ACM, Vol. 21, No. 8, Aug 1978.

[KarMi66] Karp, R. M., and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing," SIAM Journal of Applied Mathematics Vol. 14, Nov. 1966.

[Kelle77] Keller, R. M., "Denotational Models for Parallel Programs with Indeterminate Operators," Formal Description of Programming Concepts, (E. J. Neuhold, Ed.), August 1977, North-Holland Pub. Co., N.Y. N.Y. pp 337-366.

[KePaL78] Keller, R. M., S. Patil, and G. Lindstrom, "An Architecture for a Loosely-Coupled Parallel Processor (Draft)," Dept. of Comp. Sci. (UUCS-78-105), University of Utah, Salt Lake City, Utah, July 1978.

[KisRu75] Kishi, T, and T. Rudy, "STAR TREK," COMPCON 75, IEEE, N.Y. 1975, pp.185-188.

[Kosin73] Kosinski, P. R. "A Data Flow Language for Operating Systems Programming," SIGPLAN Notices, No. 8, 1973.

[Kuck77] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming," ACM Computing Survey, Vol. 9, No. 1, March 1977.

[Lampo74] Lamport, L., "The Parallel Execution of Do Loops," Comm. of the ACM, Vol. 17, No. 2, Feb. 1974.

[LauCa75] Lauer, P. E., and R. H. Campbell, "Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes," Acta Informatica, Vol. 5, pp 297-332, Springer-Verlag 1975.

[McIlr68] McIlroy, M. D, "Coroutines: Semantics in Search of a Syntax," Oxford University and Bell Laboratory, Inc. (Unpublished Paper)

[MilMi79] Milne, G., and R. Milner, "Concurrent Processes and Their Syntax," Journal of the ACM, Vol. 26, No. 2, April 1979, pp 302-321.

[Miran77] Miranker, G. S., "Implementation of Procedures on a Class of Data Flow Processors," Proceedings of the 1977 International Conference on Parallel Processing, Syracuse University, IEEE.

[Misun75] Misunas, D. P., "Deadlock Avoidance in Data-Flow Architecture," Proc. of the 1976 International Conf. of Parallel Processing, Aug. 1976.

[Misun75] Misunas, D. P., "Structure Processing in a Data-Flow Computer," Proc. of the 1975 Sagamore Computer Conf. on Parallel Computation.

[Misun78] Misunas, D. P., "A Computer Architecture for Data Flow Computation," LCS/TM-100, Laboratory for Computer Science, M.I.T., Camb., Mass.

[Ornst75] Ornstein, S. M., Growth, W. R., Kraley, M. F., Bressler, R. D., Michel, A., and F. E. Heart, "Pluribus - A Reliable Multiprocessor," 1975 NCC, AFIPS Conf. Proc. pp551-559.

[Plotk76] Plotkin, G., "A Powerdomain Construction," SIAM Journal of Computing, Vol. 5, No. 3, 1976, pp 452-487.

[RamLi76] Ramamoorthy, C. V., and H. F. Li, "Pipeline Architecture," Computing Surveys, Vol. 9, No. 1, March 1977.

[Rumba75] Rumbaugh, J. E., "A Parallel Asynchronous Computer Architecture for Data Flow Programs," MAC-TR-150, 1975, M.I.T. Cambridge, Mass.

[Stoy74] Stoy, J. E., "Proof of Correctness of Dataflow Programs," Computation Structure Group Memo-110, Laboratory for Computer Science, MIT, September 1974.

[Stoy77] Stoy, J. E., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge Mass., 1977

[Swan78] Swan, R. J., "The Switching Structure and Addressing Architecture of an Extensible Multiprocessor: CM*," CMU-CS-78-138, Carnegie-Mellon University, Computer Science Department, August 1978.

[SwFuS77] Swan, R. J., S. H. Fuller, and D. P. Siewiorek, "CM* : a Modular, MUlti-Microprocessor," AFIPS Conf. Proc. Vol. 46, 1977, National Computer Conference.

[SyCoH77] Syre, J. C., D. Comte, and N.Hifdi, "Pipeline, Parallelism and Asynchronism in the LAU System," Proceedings of International Conference on Parallel Processing, August 1977.

[Trele77] Treleaven, P. C., "Principle Components for Data Flow Computers," Computing Laboratory TR-108, University of Newcastle upon Tyne, Newcastle upon Tyne, England, July 1977.

[Weng75] Weng, K.-S., Stream-Oriented Computation in Recursive Data Flow Schemas, Laboratory for Computer Science TM-68, MIT, Oct. 1975.

[Wulf72] Wulf, W. A., and C. G. Bell, "C.mmp - A Multi-Mini-Processor," 1972 FJCC, AFIPS Conf. Proc., Vol. 41, No. 2, Apr. 1965, pp 270-271.

[YauFu77] Yau, S. S., H. S. Fung, "Associative Processor Architecture - A Survey," ACM Computing Survey, Vol. 9, No. 1, March 1977.

This page intentionally left blank.

Appendix A. Implementation of the n-merge actor

The implementation of n-merge actor presented here requires two firings and needs an additional input value F which represents the *first* state of the actor. For convenience, we use a notation In[1:v_1, 2:v_2,3:F] to mean that an operand record contains three input values v_1 at the first input arc, v_2 at the second input arc, and F at the third input arc for the state. If there is no value present for an input arc we use the symbol $ in its place. For example, In[1:$, 2:$v_2$, 3:$] means only one input has arrived at the operand record. We use a similar notation Out[1:v_1, 2:1, 3:$] to mean that the firing of the actor produces two outputs v_1 on the first output arc, 1 on the second output arc, and no token on the third output arc.

The enabling count of the actor is defined to be two, thus, the actor is enabled with any two of the three inputs. We describe the possible firing by cases:

(1) In[1:v_1, 2:$, 3:F]

The output is Out[1:v_1, 2:1, 3:$] and in addition a result packet containing S, representing the *second* state is sent to the same operand record at the third input. Since the only value that has not arrived is v_2, the next firing will contain In[1:$, 2:$v_2$, 3:S] and the result of this firing is Out[1:$, 2:$, 3:signal].

(2) In[1:$, 2:$v_2$, 3:F]

The output is Out[1:v_2, 2:2, 3:$] and in addition a result packet containing S is sent to the same operand record at the third input. Since the only value that has not arrived is v_1, the next firing will contain In[1:v_1, 2:$, 3:S] and the result of this firing is Out[1:$, 2:$, 3:signal].

(3) In[1:v_1, 2:v_2, 3:$]

The firing must choose one of the two possible outputs:

(3a) The output is Out[1:v_1, 2:1, 3:$] and in addition a result packet containing S is sent to the same operand record at the first input. Since the only value that has not arrived is F, the next firing will contain In[1:S, 2:$, 3:F] and the result of this firing is Out[1:$, 2:$, 3:signal].

(3b) The output is Out[1:v_2, 2:2, 3:\$] and in addition a result packet containing S is sent to the same operand record at the first input. Since the only value that has not arrived is F, the next firing will contain In[1:S, 2:\$, 3:F] and the result of this firing is Out[1:\$, 2:\$, 3:<u>signal</u>].

The firing rules above does not include the case for all three input values to be in the operand record. This is because the <u>insert</u> operation on an operand record is implemented as a critical region that allow one insertion to take place at a time and an operand record is enabled as soon as two values arrive. Notice that each firing will cause an instruction fetch, and this is the consequence that we would like Execution Controllers to process all instructions in the same manner.

Appendix B. A cyclic schema for merging two streams

The schema shown in Figure B has two inputs S_1 and S_2 each receiving a stream represented as a structure, and Out is the output of the schema. The n-merge$_2$ actor is enabled as soon as one input arrives and produce two vales: the stream value arrived on the s output arc, and a boolean value on the output A: true if it is the first input, and false if it is the second input. The schema uses a false gate F in the model of Dennis and Fosseen to avoid excessive use of sink actors. The two actors cons$_2$ and write-hole together form the cons actor introduced in Chapter Five. The capitalized letters at the end on each arc implies connections between actors to avoid confusion.

The cyclic schema works by constructing a stream using the cons$_2$ and the write-hole actor for each value of the two input streams. The schema recycles the arrived stream structure to the proper input s_1 or s_2 determined by the boolean output B. The construction of output stream is rather complicated because the whole schema must signal its completion of operation in some manner. And this is achieved by using the signal output of the write-hole actor. The schema terminates its operation when one of the input stream is empty and this adds additional complexity to the diagram.

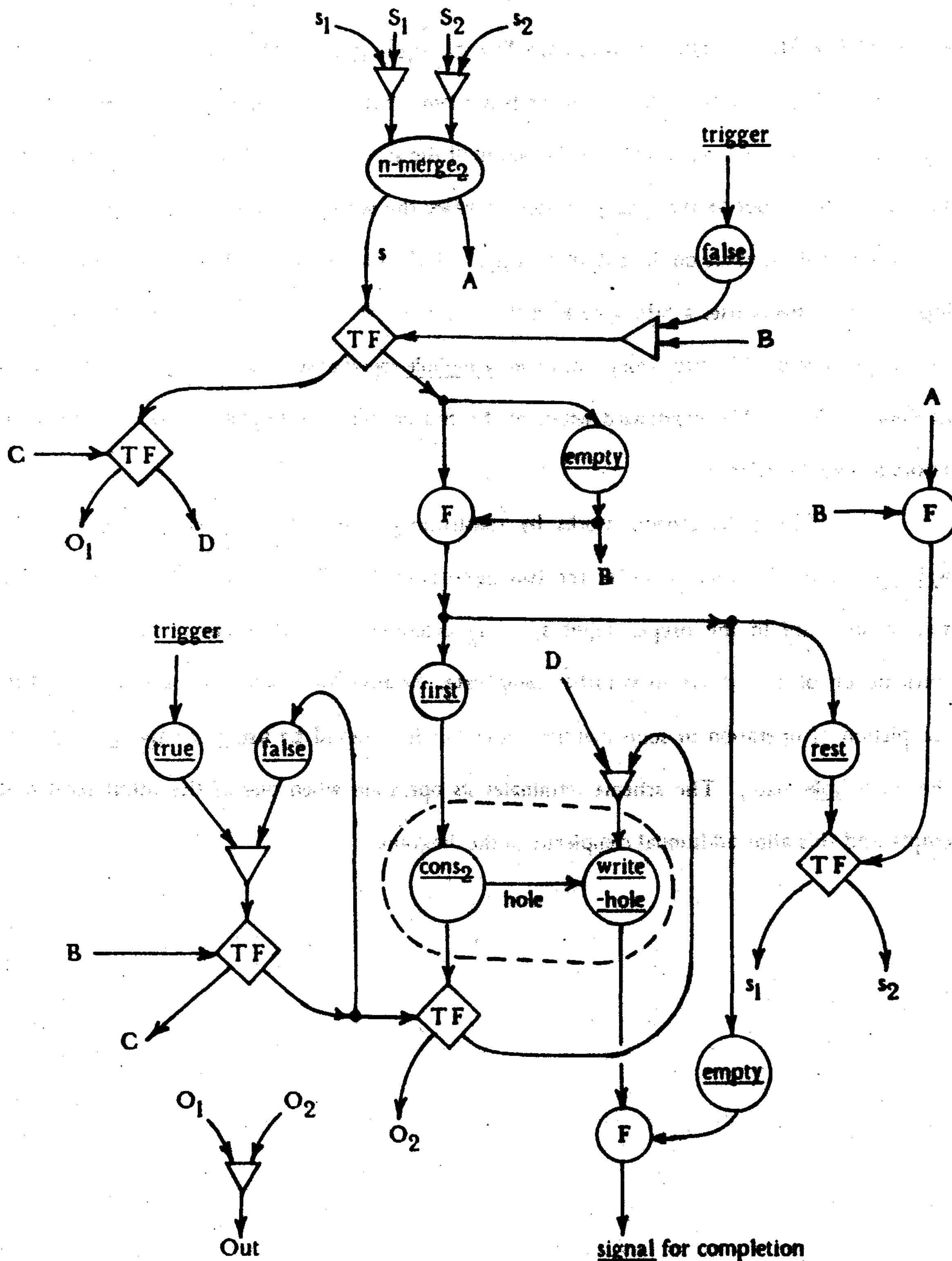


Figure B. A cyclic schema for merging two streams