MIT/LCS/TM-68

STREAM-ORIENTED COMPUTATION IN RECURSIVE DATA FLOW SCHEMAS

Kung-Song Weng

October 1975

MIT/LCS/TM-68

STREAM-ORIENTED COMPUTATION IN RECURSIVE DATA FLOW SCHEMAS

KUNG-SONG WENG

OCTOBER 1975

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

STREAM-ORIENTED COMPUTATION IN
RECURSIVE DATA FLOW SCHEMAS

by

Kung-Song Weng

Submitted to the Department of Electrical Engineering and Computer Science on September 2, 1975 in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

In this thesis we present a parallel programming language based on a parallel computation model known as data flow schemas. Syntactically, the language resembles programming languages such as Algol 60, but does not have GOTO's, WHILE-loops, and non-local variables. The attractiveness of this approach lies in the inherently determinate nature of data flow schemas and the possibility of formalizing the semantics of the language within the formalism suggested by Scott and Strachey. The language provides programming features for stream-oriented computation and intercommunicating systems. We introduce the notions of proper initialization and termination of such systems. A subclass of determinate systems in which these properties can be easily checked is defined and a translation into recursive data flow schemas is given.

THESIS SUPERVISOR: Jack B. Dennis
TITLE: Professor of Computer Science and Engineering

ACKNOWLEDGEMENT

I wish to express my gratitude to Professor Jack B. Dennis for the invaluable guidance and support throughout phases of this research. I would also like to thank David Ellis for his constructive help in the preparation of this thesis, and David Misunas and Clement Leung for providing helpful comments. My special thanks are due to Gloria Marshall for typing the thesis and Michelle Hoshi, my wife, for her patience, understanding and encouragement, including the long hours she spent in drawing the figures. Finally I would like to acknowledge Project MAC for the use of its facilities during the preparation of this thesis.

This work was supported in part by the National Science Foundation under the research grants GJ-34671 and DCR74-21892.

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 Parallel Computation

Over the past twenty years there have been many technical advances in the field of computing, but achieving efficient and effective utilization of computing resources still remains a significant problem. It is generally recognized that parallel computation provides for the speed-up of computations and for better utilization of computing resources. The advantages of parallelism, unfortunately, are often overshadowed by the difficulty of exploiting the natural parallelism of computations in highly parallel computing systems. This difficulty is due mostly to the lack of adequate programming formalisms and appropriate computer structures to support their efficient implementation. As the cost of software development becomes higher and the design parameters for digital systems change, it is increasingly more important to consider the principles by which programming languages and computer structures may efficiently perform parallel computation.

Conventional programming languages such as FORTRAN, Algol 60, and PL/I are essentially based on the centralized sequential control structure of the Von Neumann machine. Control primitives for parallel computation such as CALL and WAIT in PL/I, and synchronization schemes using semaphores and the semaphore primitives P and V introduced by Dijkstra [18] are natural extensions of the sequential control concept. Programs written in conventional languages which incorporate these parallel programming constructs are limited in the degree of parallelism they exhibit, and the use of these primitives may introduce undesirable side-effects. From a different point of view, the difficulty in proving the correctness of these programs also reflects the unsuitability of conventional programming languages for parallel programming tasks.

The lack of suitable languages has inhibited the development of parallel computation techniques with which many forms of computation such as pipeline or multi-pass processes may be more naturally expressed. Because of the highly disparate structure of existing highly parallel machines the parallel computation techniques used with highly parallel computers such as the ILLIAC IV or the CDC STAR-100 are highly machine dependent and therefore are not applicable in general.

In what follows we shall discuss some fundamental concepts in parallel programming and related issues.

Parallelism

The term "parallelism" generally refers to the state of existence of concurrent activities or processes (processes may be conceptually parallel). Because of the limited degree of parallelism which is explicitly expressed using control primitives of conventional programming languages, high performance computers such as IBM 360/91 [2] and CDC 6600 [43] have built-in hardware which analyzes segments of instruction sequences to determine which instructions can be executed concurrently. The performance of these computers is, however, highly dependent on the frequency of JUMP instructions. This approach often would require complex compile time analysis of a program in order to achieve an acceptable level of utilization. The overhead of the analysis is often non-negligible owing to the complexity of conventional languages. It is therefore desirable that a parallel programming language should allow simple detection of potential parallelism of instructions.

Determinacy

A program is determinate if repeated execution with the same set of input data yields the same outputs. Non-determinacy may arise if concurrent processes share common data or if there are subcomputations which are inherently non-determinate (such as random number generators). The effect of non-determinacy when not intended is undesirable. For instance, the presence of non-determinacy makes debugging and program testing very difficult because repeated execution may not reveal certain errors. One of the most significant drawbacks of extant parallel programming languages is that the use of parallel computation primitives may result in non-determinacy.

Deadlocks

The issue of deadlocks received much attention as soon as primitives for coordinating concurrent processes such as "lock" and "unlock" were suggested [14, 3, 18]. The cost of programming errors resulting in deadlocks is often high since a significant quantity of computing resources may be wasted until the situation is remedied. It is therefore necessary that good higher level features for parallel programming should guarantee that programs in the language would not cause deadlocks provided computing resources required for completing the computation is allocated.

Currently there is a strong emphasis on the avoidance of programming errors and the ease of proofs of correctness for programs. These principles should be stressed even more for parallel programming languages because of the additional complexity introduced by parallel programming features. In addition to the limitations already pointed out, conventional programming languages do not satisfy the requirement of programming modularity. For example, the use of primitives such as P and V requires names of semaphores to be specified in statements. As a result, routines employing these primitives need to be modified when used in a different environment. This observation is equally applicable to other control primitives such as co-routine primitives which require that labels for entry points or reactivation points be specified.

1.2 Data Flow Concepts

In contrast to the notion of sequential control of conventional programming languages, the "data flow" concept is based on the observation that an operation (or an instruction) should be executed as soon as the required input operands are made available by the completion of operations supplying the inputs. Among the models of parallel computation which incorporate data flow concepts (Adams [1], Bährs, [4], Rodriguez [38]), the data flow schemas introduced by Fosseen [19] are inherently determinate and sufficiently expressive to encompass schemes which model programming features such as conditionals, while_loops and procedure invocations. The attractiveness of data flow schemas as the semantic basis of parallel programming languages lies in several properties:

(i) parallelism at instruction level is exposed;

(ii) the set of rules which governs the progress of computations is relatively simple;

(iii) any schema constructed from any interconnection of data flow schemas is determinate.

Recently computer structures based on data flow models have been specified by Misunas [35] and Rumbaugh [39]. In the architecture suggested by Misunas the efficiency of the execution of a program in the data flow representation is particularly independent of the structure of the program; therefore, the analysis of the behavior of a program is very much simplified.

1.3 Statement of the Problem

The objective of this thesis is to design a textual language with data flow schemas as its semantic basis, and to consider the applicability of data flow concepts to problems in current parallel programming languages. The criteria which should be satisfied by the textual language are the following:

(1) There is a simple translation of the language into data flow schemas. The simplicity of translation rule reflects the efficient implementation of the language on a data flow processor (computer which runs on some data flow representation) and thus avoids the overhead often exist in the process of exploiting parallelism at the instruction level.

(2) There is a compile time check for deadlocks, if possible.

(3) The semantics of the language should be simple enough to suggest the possibility of formalization.

(4) The language provides programming features for stream-oriented computation.

(5) The language provides programming features for expressing a system of interconnected modules (or processes) which communicate by exchanging data through communication channels.

We have restricted the scope of this thesis to the following domains:

(a) We shall be concerned with only determinate computations. The extensions to allow non-determinacy are not considered in this thesis and are open problems.

(b) The only data types we are interested in are integer and boolean types, and data structures are not considered.

1.4 Synopsis of Thesis

Chapter 2 introduces data flow schemas and defines certain subclasses of data flow schemas which provide the framework for the development of the language in subsequent chapters. The class of recursive well formed (rwf) data flow schemas models program constructs of conditionals and recursive procedures. A textual language TDFL (textual data flow language) is defined to correspond to this class of data flow schemas. The language adopts the single assignment rule which has been used by other languages such as that suggested by Enea and Tesler [42]. The elimination of goto's and non-local references results in the semantic simplicity of the language and also simplifies dramatically the translation into rwf data flow schemas.

In chapter 3 we extend the language by defining streams and primitive operations on streams. The feasibility and expressiveness of the extension for stream-oriented computation are demonstrated by two mini-programs. The first exhibits the degree of parallelism in which the simple task of adding a stream of numbers can be expressed. The second demonstrates the conceptual simplicity and the flexibility of problem representation provided by stream-oriented computations using the sieve of Eratosthenes for generating primes.

In chapter 4 we introduce a programming construct for describing a system of interconnected modules. A summary of the results in the theory of determinate systems is also given. These results are the basis on which we assert the semantic simplicity of the construct. Two important properties of interconnected modules are introduced: proper initialization and proper termination. The question of whether these properties are decidable for the general class of determinate systems has not been explored.

We then introduce a subclass of interconnected modules in which both the compile time check for the two properties and the translation are possible. One of the main reasons why the translation into recursive data flow schemas is considered is that we feel that cycles, if possible, should be removed unless the translation process is unfeasible.

In the final chapter we discuss some of the issues which we feel can be best treated after the presentation of the language. Some of these issues are: elimination of an iteration (or while_loop) construct, efficiency issues related to primitives for stream operation and the firing rule for procedure applications. Further work and research are suggested.

Chapter 2

Data Flow Schemas and Basic Structure of the Language.

In this chapter we introduce the data flow schemas and the rules of computation along with definitions and illustrations of subclasses of data flow schemas. The class of data flow schemas which are well formed serves as the basis for the semantics of the textual language TDFL (textual data flow language). Syntactically the language resembles conventional languages, but there are major semantic differences. A program in TDFL defines a data flow schema according to the translation rules defined in section 2.3. An identifier in a program may be thought of as a variable in the conventional sense; in this view an identifier can be assigned only once - "global" (or "non-local") variables as in Algol 60 are not allowed in TDFL. The exclusion of goto's from TDFL is a natural consequence of our preference for the simple syntactic correspondence of TDFL with data flow schemas.

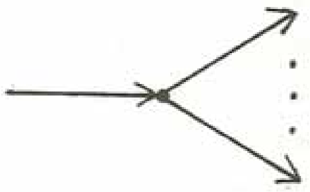
2.1 Data Flow Schemas.

An (m,n) data flow schema consists of a directed graph whose nodes are either links or actors and additional mechanisms and rules which define how computations proceed. The notation and terminology of links and different types of actors are shown in figure 2.1.

Each actor has an ordered set of input and output arcs. Arcs pointing to a node are input arcs of the node, and arcs leaving it are called output arcs. A graph of an (m,n) schema must have m link nodes which do not have any input arcs (referred to as input links) and n link nodes which do not have any output arcs (output links), and all other link nodes must have one input arc and at least one output arc emanating from it. In addition, we require that the graph must be proper in the sense that each arc leaves from an actor and terminates at an actor.

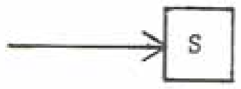
Corresponding to the notion of a procedure as in Algol-like languages we define an (m,n) module to consist of a graph of an (m,n) schema and an initial configuration. A configuration is an assignment of tokens, each accompanied by a label to some arcs of the graph. An assignment of a token to an arc is represented by the presence of a solid circle on an arc. The label of a token

a) <u>link</u>

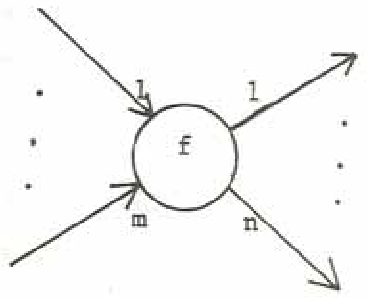


b) <u>actors</u>

(i) <u>terminal</u> <u>sink actor</u>



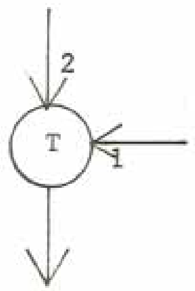
(ii) <u>operator</u>



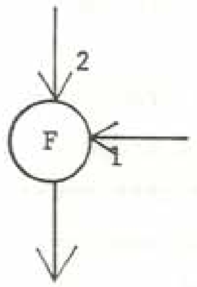
the letter "f" denotes a function under some interpretation

(iii) <u>gates</u>

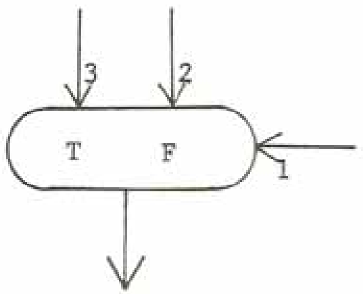
<u>T gate</u>



<u>F gate</u>



<u>M gate</u>



<u>Figure 2.1 Types of nodes</u>

denotes the value carried by the token and may be omitted when the value is irrelevant to our discussion, (refer to figure 2.3).

Informally, the presence of a token on an arc means that a value denoted by the label is made available to the node to which the arc points. The initial configuration may be thought of as the initialization of variables used in a procedure.

To describe a computation of an application of a module to some input values we introduce the notion of snapshots:

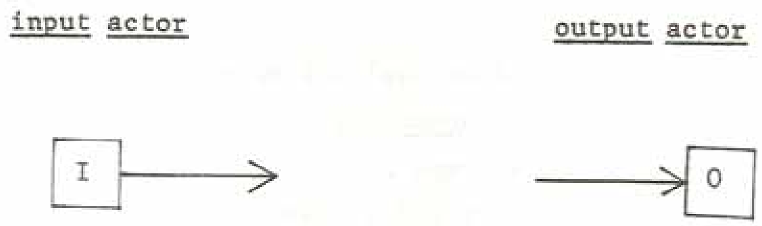
> A snapshot consists of a graph of a data flow module connected to a set of input and output actors and a configuration. The diagrams for input and output actors are shown in figure 2.2. The figure also shows how a graph of a data flow module is connected to these actors.

The computation of a data flow module when applied to a set of input values is described by a sequence of snapshots. The initial snapshot of the sequence shows the graph obtained from that of the module as described above and a configuration the same as that of the module. In addition, each input actor has a specification of what values are to be supplied to the input link node to which it is connected during the computation. The computation advances from one snapshot to the next through the firing of some node that is enabled in the first snapshot. The condition under which a node is enabled is depicted in figure 2.3. The firing rules for the input and output actors are also shown in figure 2.3. It should be noted that a necessary condition for any node to be enabled is that each output arc does not hold a token.

Firing rules.

Except for gates, a node is enabled when tokens are present on all input arcs and no token is present on the output arcs. Firing of such a node is initiated by absorbing tokens from the input arcs and completed by placing a token on each of the output arcs. The values of the output tokens are functionally related to the values of the input tokens. The links simply replicate the values received for distribution to several actors. A sink actor when fired absorbs the input token. The effect of firing an operator is to apply to the input values $v_1,...,v_m$ the function associated by an interpretation with

a) additional actors



b) a snapshot

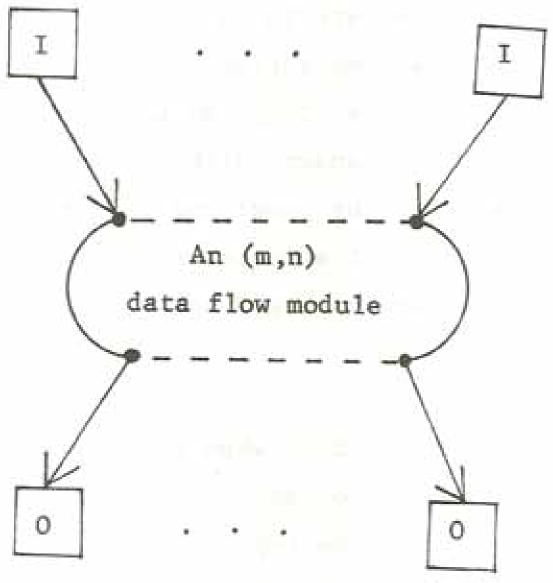
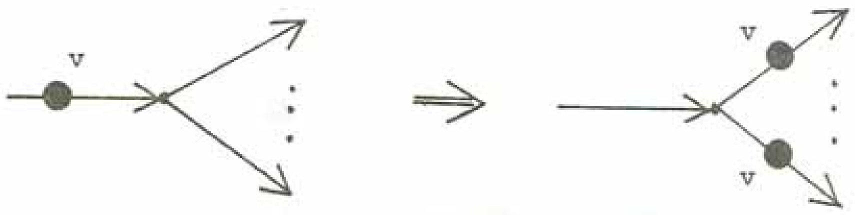


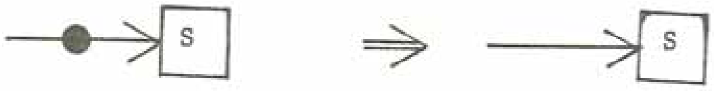
Figure 2.2 Additional actors and a snapshot

a) link

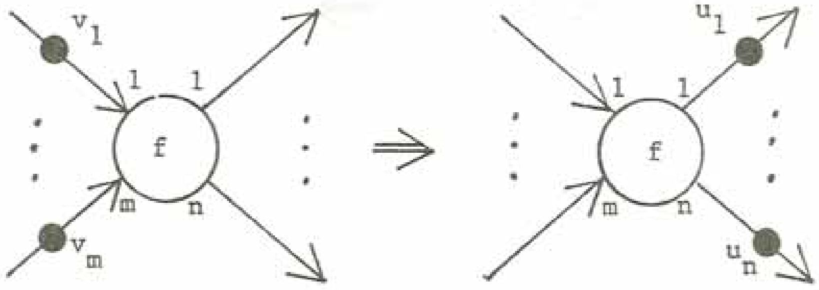


b) actors

(i) sink actor



(ii) operator



The letter "f" denotes a function defined over the domains of the values $v_1,\ldots,v_m$ and $u_1,\ldots,u_n$.

(iii) gates

T gate

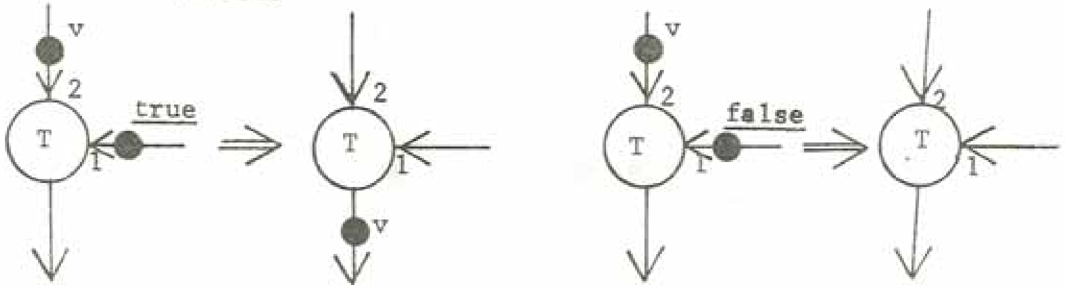
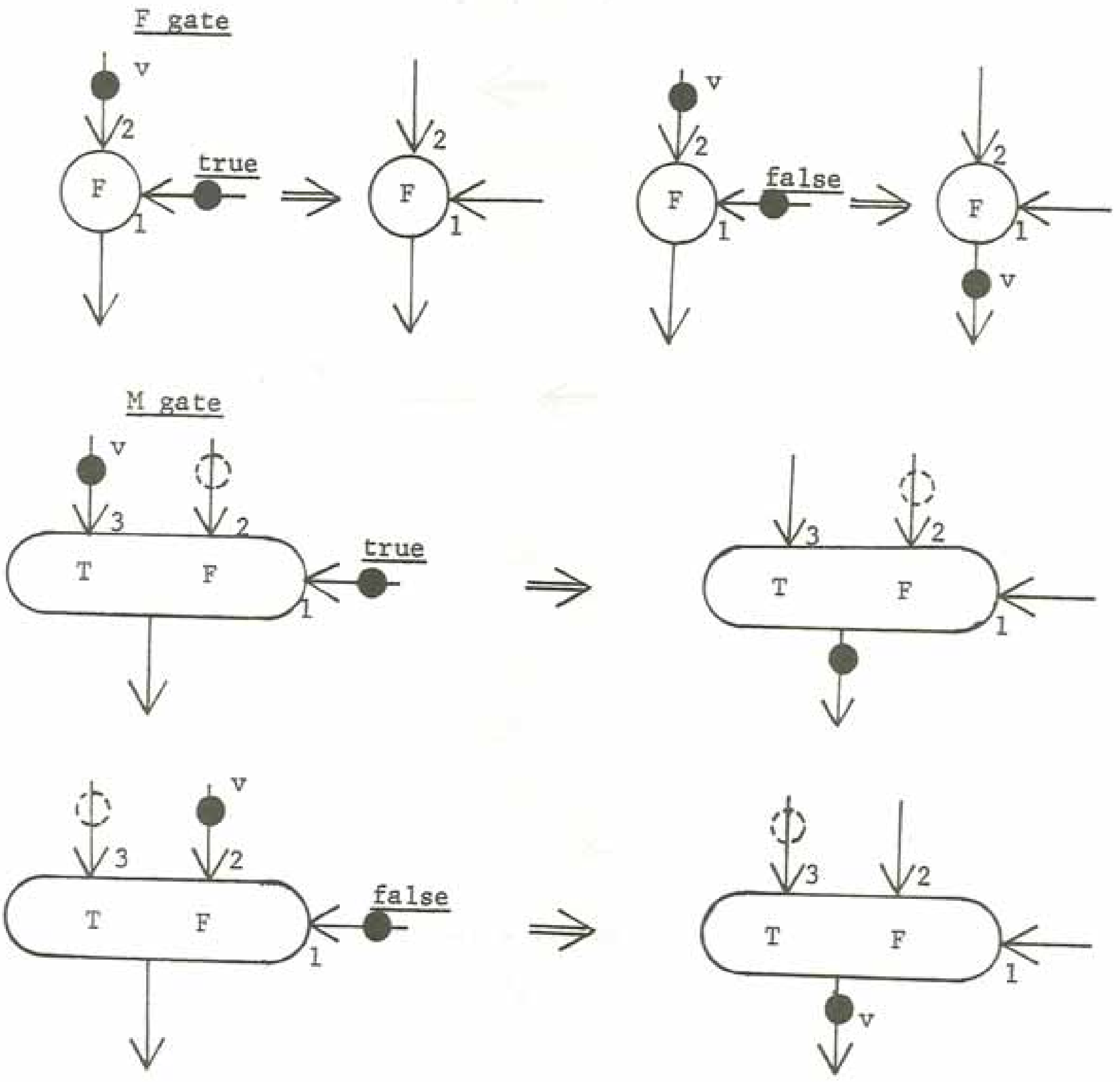


Figure 2.3 Firing rules

(iii)



(iv)

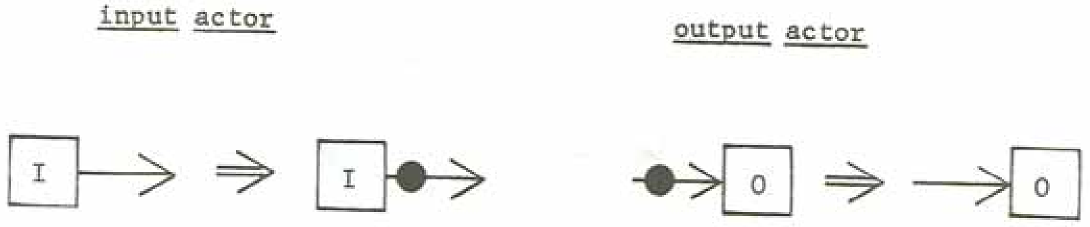
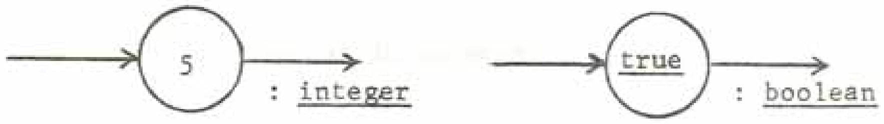


Figure 2.3 (continued) Firing rules

the function letter written inside the operator to yield output values $u_1,\ldots,u_n$. Since the operators may not be defined for all types of values, we require labels to be used to identify the type of the values for which it is defined whenever ambiguity may arise (see figure 2.4).

We also require that a constant value operator should have an input arc serving as a trigger to the operator.



the input arcs can be of any type

Figure 2.5 constant functions

The gates are special operators which require that the values carried by the tokens at the first input arcs are boolean values : {true, false} (see figure 2.2). For the rest of the thesis we shall refer to these input arcs as control arcs. The boolean values are used to permit the outcome of tests performed by some operators to affect the "flow" of values to actors in the manner described hereon. A T gate (F gate) passes a value presented at the second input arc on to the output arc if the boolean value received at the control arc is true (false), otherwise the value is discarded by not placing it on the output arc. The M gate (read as "merge gate") allows a boolean value to determine which of the two input arcs passes a value to its output arc. If the boolean value true arrives at the control arc, the value present or next to arrive at the T-input arc (the third input arc) is passed. A value present at the F-input arc (the second input arc) is left undisturbed. The complementary action occurs for the boolean value false.

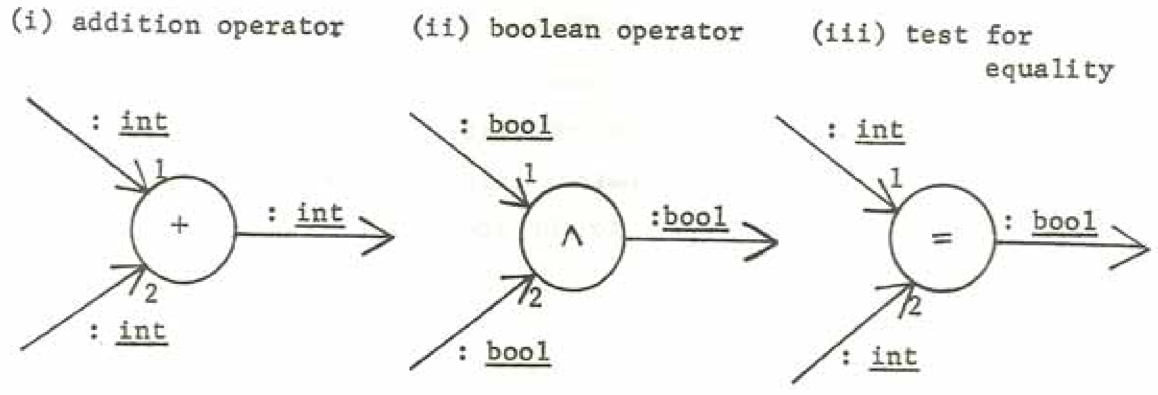


Figure 2.4 Examples of actors with typed arcs

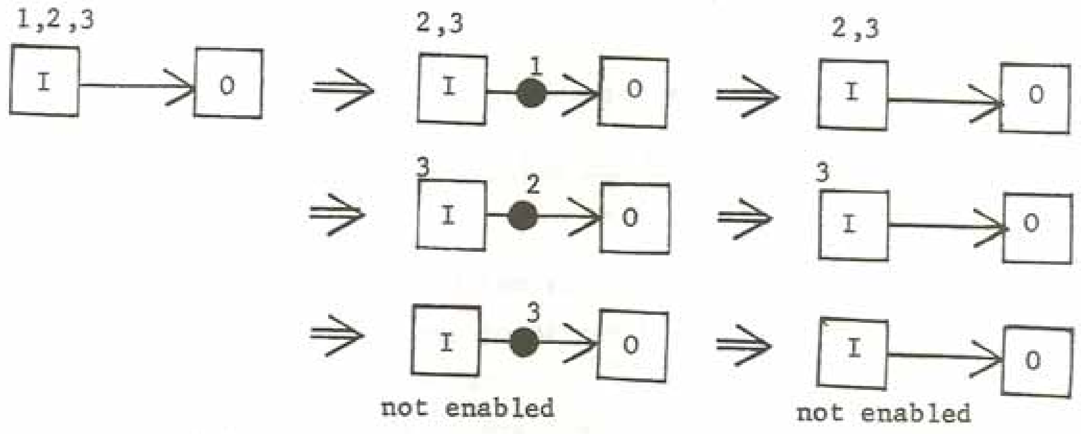
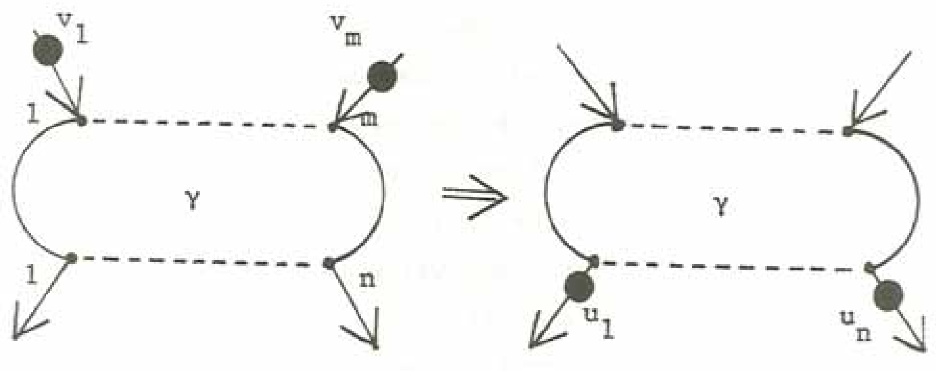


Figure 2.6 An example of a specification for an input actor



γ denotes the initial configuration of the module

Figure 2.7 Well-behaved module

An input actor is enabled when there is no token present on the output arc and the specified values have not been all placed on the output arc. Firing the input actor causes the next value in the specification to be placed on the output arc. An example of a specification is shown in figure 2.6, where the sequence of integers 1,2,3 written besides the input actor defined that the sequence of tokens placed on the output should be integer values "1", "2", and "3" in this order. In the example, the input actor after placing the value "3" on the output arc is never enabled again. The output actor simply absorbs the token arrived at the input arc.

Well-Behaved Data Flow Modules

In the rest of the chapter we shall be interested in a class of data flow modules which is a subclass of well-behaved data flow modules. A data flow module is well-behaved if the conditions (i) and (ii) hold:

(i) One set of output values is produced for each set of input values (see figure 2.7).

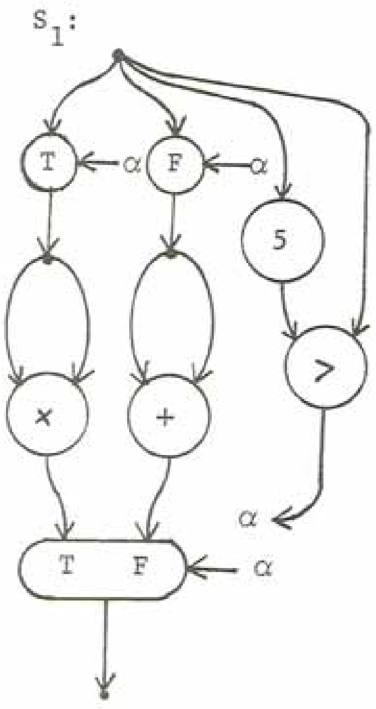
(ii) After a set of output values is absorbed the snapshot of the computation returns to its initial configuration. Furthermore, we require that in the initial configuration no actor besides the input and output actors is enabled.*

Well-behaved data flow modules are always functional in the sense that a set of output values is determined uniquely by a set of input values. The functionality of a well-behaved module follows from the fact that the links and actors are determinate systems as defined by Patil [36], and the rules of behavior of a determinate system ensures that the property of determinacy is preserved for the operation of interconnecting links and actors to form a data flow schema.

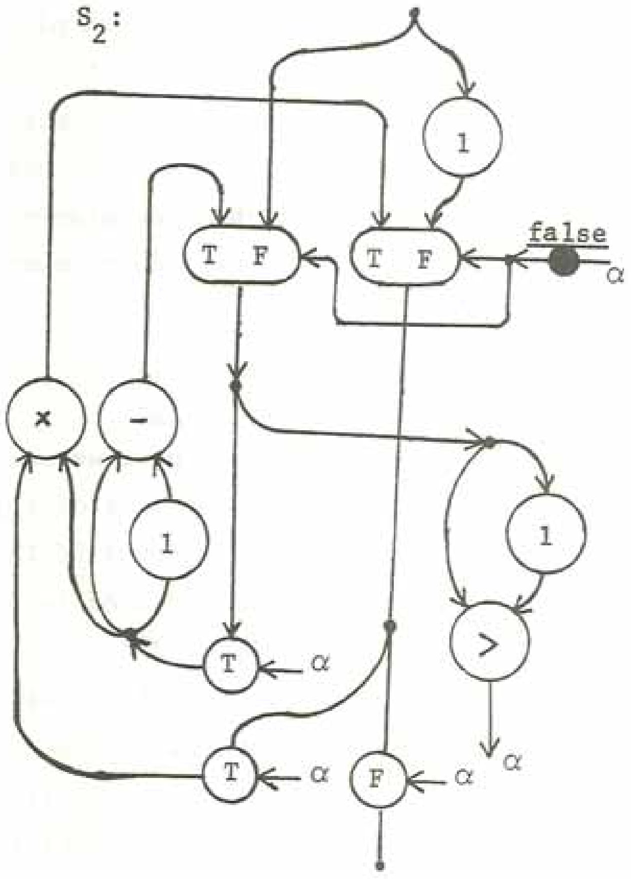
The data flow modules S_1 and S_2 shown in figure 2.8 are well-behaved modules. The module S_3 is not well-behaved since the value carried by the token is different after a non-zero value arrives at the input. The module S_4 is not well-behaved either.

* This requirement is a stronger one than that defined by Dennis [12].

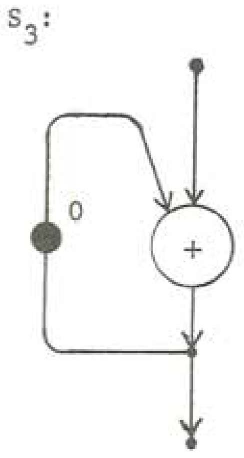
a)



b)



c)



b)

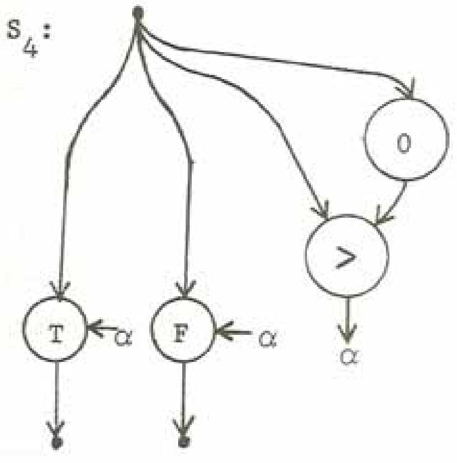


Figure 2.8 Examples of data flow modules

2.2 Recursive well-formed data flow schemas

The data flow schemas described in the previous section do not define how a data flow module may be employed in another module. We introduce module application actor whose notation and the firing rule are shown in figure 2.9. By allowing a name to be uniquely associated with a module, the name may appear inside a module application actor. A module application actor is enabled when a token arrives at each input arc. The effect of firing the actor is to modify the snapshot by replacing the actor with the module designated by the name (see figure 2.10). We allow modules to be recursively defined by allowing the name of the module to be used in an application actor in itself.

The introduction of recursive modules has resulted in the elimination of the iteration schema (figure 2.8(b)) from the class of recursive well-formed (rwf) data flow schemas defined below.

> An (m,n) rwf data flow schema is an (m,n) data flow schema formed by any acyclic composition of component data flow subschemas, where each component is either a link, a sink, an operator, a module application, or a rwf conditional subschema.
>
> An (m,n) rwf module consists of the graph of an (m,n) rwf schema and an initial configuration such that no token is assigned to any arc.

Conditional Subschemas

The diagram of a conditional subschema is shown in figure 2.11, where the heavily darkened arcs are labelled by letters denoting the number of arcs they represent. If P is a (q,s) subschema and Q is an (r,s) subschema then the conditional subschema is an (m,s) subschema. The gates T, F, and M actually represent collections of gates of the same type; each of the gates has a control arc from the same source indicated by the Greek letter α. The subschema R is any acyclic composition of operators and has one output arc which must be of type boolean.

An rwf conditional subschema is a conditional subschema whose component subschemas P and Q are rwf schemas.

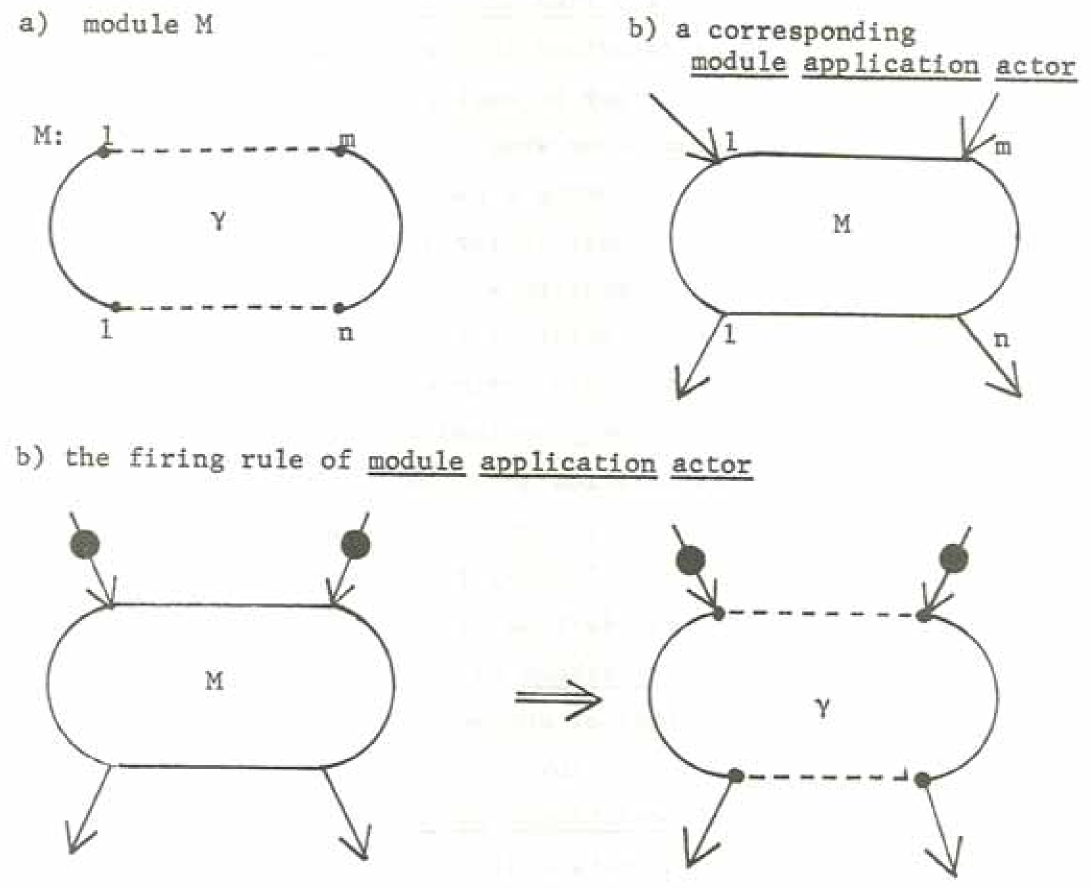

a) module M
b) a corresponding module application actor
M:
1
m
γ
1
n
1
m
M
1
n
b) the firing rule of module application actor
M
γ

Figure 2.9 The notation and firing rule for a module application actor

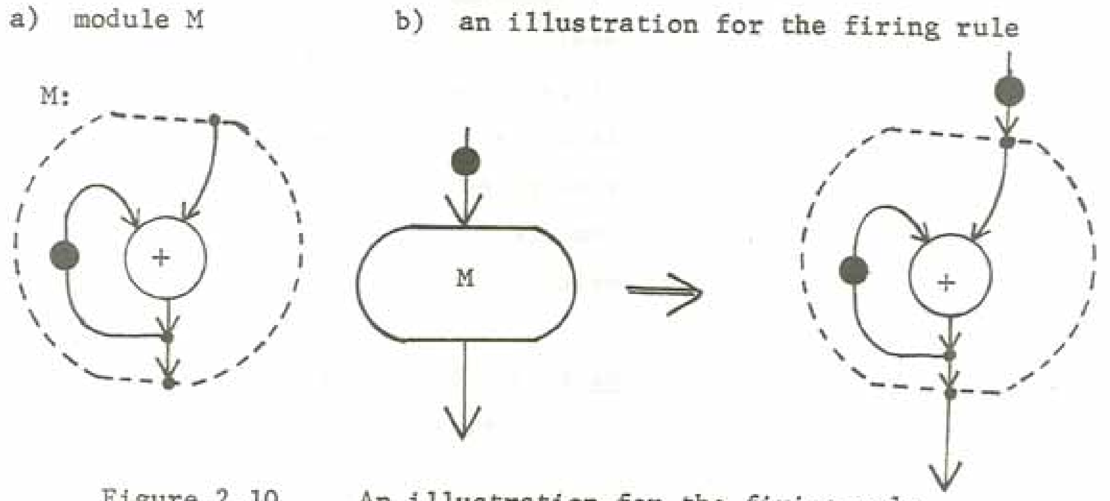


Figure 2.10 An illustration for the firing rule

Figure 2.12 shows an example of an rwf schema Fact which computes factorial function.

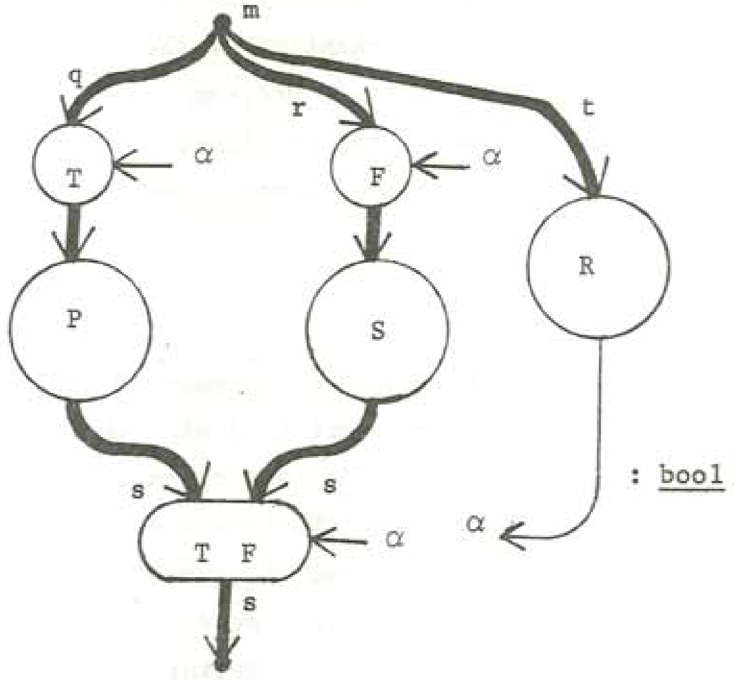


Figure 2.11

Fact :

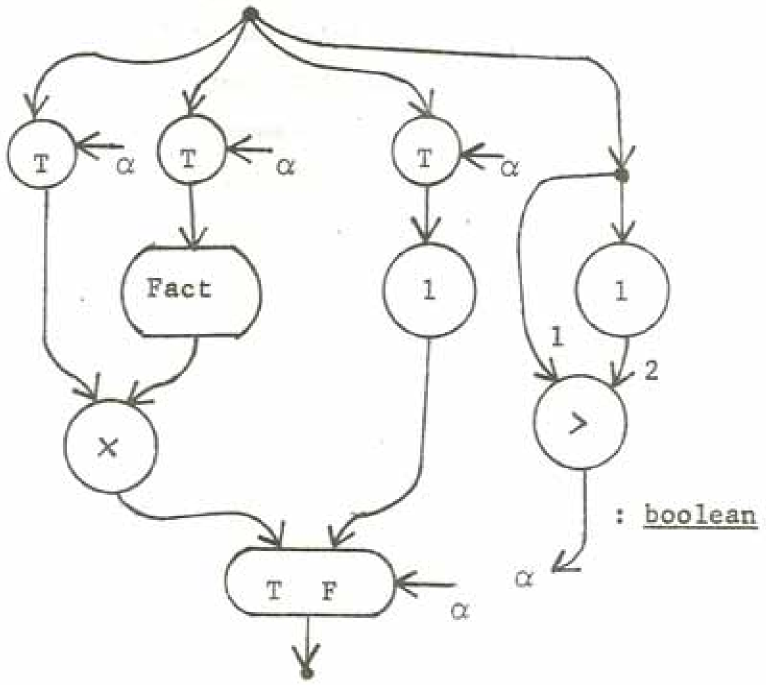


Figure 2.12 An rwf module.

2.3 Textual Data Flow Language (TDFL)

In this section we describe a textual language design based on the class of rwf-modules. The semantics of the language is defined by providing rules of translating a program in TDFL into data flow schemas. The syntax of TDFL is described in figure 2.13, where the notation "$\{\cdots\}^*$" is used to denote any number of repetitions of the syntactic objects bounded by the braces.

Syntax

In TDFL an underlined word is a reserved word, and a name can consist of any alpha-numerical characters including under-scores "_" whose first character must be a letter. The type of an identifier is declared when the name is used by specifying the type name: either <u>integer</u> (<u>int</u>) or <u>boolean</u> (<u>bool</u>). We require that all identifiers appearing in an interface must be typed and other identifiers need not be typed if no ambiguity arises. A program may have a list of module definitions followed by a list of statements separated by semi-colons. A module if defined recursively must use <u>rmodule</u> as its heading, otherwise <u>module</u> is used. The interface defines the formal parameters of the module by explicitly defining input and output identifiers. We require that there be at least one input and one output identifier for each module and that there is at least one statement in the body of a module or program. A statement is either an assignment, a module-call, or a conditional statement.

```
< program > ::= prog { < module > }* < body > progend
< module >  ::= < heading > < interface > { < module > }* < body > mend
          < heading > ::= < name > : module | < name > : rmodule
          < interface > ::= ( < in_list >; < out_list > )
          < body > ::= { < statement > ;}* < statement >
< statement > ::= < assignment > | < conditional > | < module-call >
< assignment > ::= < exp > → < id_list >
< conditional > ::= if < boolean-exp > then < body >
                                       else < body >
                                       end
< module-call > ::= < name > ( < exp_list >; < out_list > )
< module_application > ::= < name > ( < exp-list > )
< exp > ::= < id > | < arith_exp > | < boolean_exp > | < module_application >
          < airth_exp > ::= usual arithmetic expressions | <module_application>
          < arith_op >  ::= + | - | x | /
          < boolean_exp > ::= usual logical expressions | < module_application >
          < b_op >      ::= ∧ | ∨ | ¬ | > | < | =
          < truth_value > ::= truth | false
< exp_list > ::= { < exp >,}* < exp >
< id_list > ::= { < id >,}* < id >
< id > ::= < name > {:< type > }
< type > ::= integer | boolean
< in_list > ::= < id_list >
< out_list > ::= < id_list >
```

Figure 2.13 Syntax of TDFL

An assignment has an expression to the left of the assignment operator "$\rightarrow$", and a list of identifiers to the right. Each assignment statement may be regarded as defining the values of the identifiers to be the value of the expression. Normally, if the expression is an arith_exp or a boolean_exp only one identifier is in the id-list. If more than one identifier exists, then they are all defined to have the same value. For a module_application expression, the number of identifiers in the id-list must match the number of output parameters as specified by the interface of the module definition; similarly the number of identifiers used as input parameters must match that of the definition.

A module_call is another form of application of modules. A module call is analogous to a procedure (or function) application in most of programming languages. For a conditional statement we require both branches of the conditional, <u>then</u> < body > and <u>else</u> < body >, are specified followed by an <u>end</u> to delineate the conditional.

<u>Semantics</u>

A module definition specifies a data flow module which may be used in a module_call or a module_application. The statements in the body of a module definition describes a data flow module where an identifier may be regarded as specifying a link node. The execution of a program then is the application of the data flow module described in the body. To maintain a straightforward correspondence between a program and a data flow module, several semantic constraints are imposed on the language. The language under these constraints has characteristics of a single assignment language in which each identifier (or variable) stands for a well-defined value and cannot be updated (i.e., reassigned another value). To describe the semantics a definition is in order.

> A name is <u>defined</u> if either it is used as a module name, it appears in the < out_list > of a module_call or it appears on the right-hand side of an assignment statement.

a. Scope rules

The scope rules for names used as module names and those used as identifiers are different in the following way:

(i) A module name defined in a module M is local to M, and within M the name extends in scope throughout the module including other module definitions defined in M. This facilitates construction of modules employing other modules.

(ii) A name used as an identifier (or non-module name) is strictly local in the sense that the scope of an identifier in a module is bounded within the module and does not extend into the bodies of module definitions. Thus all identifiers in a module are either defined in the module or an input parameter.

b. Single assignment rule

We require that within the scope of a name it can be defined only once except when the name is defined in the body of branches of a conditional statement. The exception allows an identifier to be defined in both branches of a conditional, and within the body of each branch it must satisfy the single assignment rule. Without this exception the "value" (in the sense discussed earlier) of an identifier cannot be affected by the value of the boolean expression. In the case when the identifier is defined in only one of the branches, the identifier may not be referenced outside of that conditional statement and can be referenced only within the body of the branch in which it is defined. Therefore an identifier defined in nested conditional statements can be referenced outside of that nesting if and only if it is defined in all branches. An example is shown in figure 2.14 to illustrate the rule.

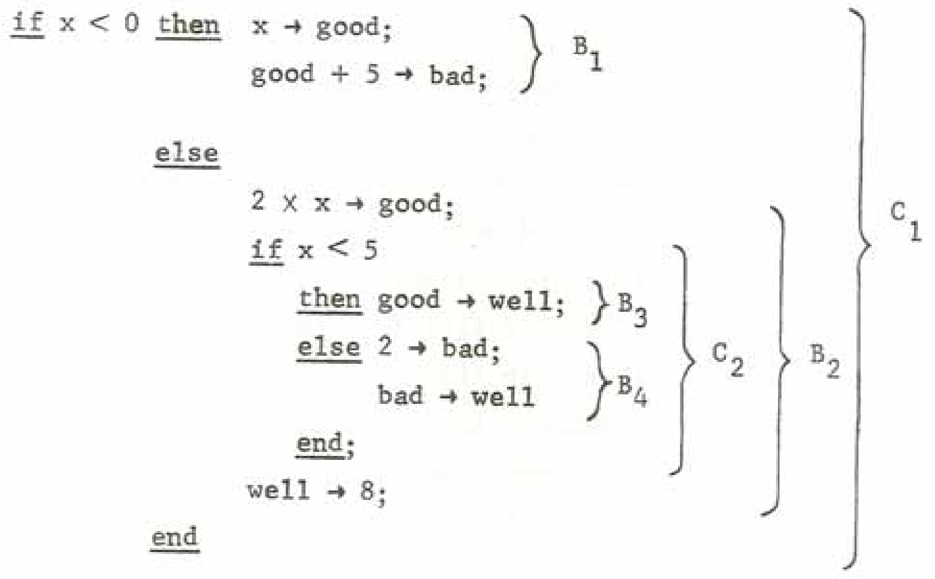
The identifier "good" satisfies the single assignment rule and can be referenced outside the conditional C_1. The identifier "well" can be referenced outside C_2 but not outside C_1. "Bad" cannot be referenced except in B_1 and B_4.

c. Well-defined identifiers

An identifier is *well-defined* if in the statement defining the identifier all identifiers referenced are *well-defined* in the preceding statements.

An identifier is automaticallly well-defined if no reference of other identifiers is made in the statement, or if it is an input.

Thus we require all identifiers to be well-defined and they must satisfy the single assignment rule. This requirement guarantees that an identifier is properly defined in the sense that it may be regarded as designating a unique value.



```
if x < 0 then  x → good;
               good + 5 → bad;

         else
               2 × x → good;
               if x < 5
                  then good → well;
                  else 2 → bad;
                       bad → well
                  end;
               well → 8;
         end
```

Figure 2.14 An example for the single assignment rule

With these semantic constraints, we can define the translation rules by associating with each statement in the body of a module or a program a subgraph of the data flow module it corresponds to. An assignment statement is a specification of how each link node represented by identifiers and operators (including constant function operators) are connected. A conditional statement specifies a conditional subschema where the statements in each body of a branch specifies the subschema in the conditional schema. A module call simply represents a module_application in a data flow schema. The rules of translation are described below informally. With the details given here the readers should be able to formalize the translation process on his own.

Rules of Translation

The translation rules are defined recursively and we always translate a list of statements corresponding to the syntactic unit of < body >. We introduce several definitions used in the translation rules.

Def. Let L be a list of statements, then the input and output of L denoted as I(L) and O(L) respectively are recursively defined as follows:

(i) for each assignment statement a, we define O(a) to be the set of identifiers defined by a, and I(a) to be the set of identifiers referenced in a.

(ii) for each conditional statement c: if B then P else Q end, we define $O(C) = O(P) \cap O(Q)$

$$I(C) = I(B) \cup I(P) \cup I(Q).$$

That is, the output of c is the set of identifiers common both to P and Q. The set of input of c is the set of identifiers referenced in c and not defined in P and Q.

(iii) $$I(L) = \bigcup_{s \in L} I(s) - \bigcup_{s \in L} O(s)$$

$$O(L) = \bigcup_{s \in L} O(s)$$

The set of outputs of L is simply the union of the outputs of each statement $s \in L$. The set of inputs consists of all identifiers referenced but not defined in L.

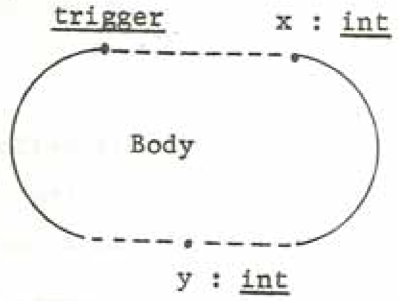
To translate L into a graph of a data flow module G(L), we perform:

(i) Determine the sets I(L) and O(L). For each identifier in the sets we create a link labelled by the identifier. These are input links and output links of the graph. In addition to these links, we also create a special link node called trigger, which is connected to all constant function operators, (refer to figure 2.15).

(i) module definition

S : module (x : int ; y : int)

$\Longrightarrow$



(ii)

x : int + 5 → y : int

$\Longrightarrow$

M(y : int, 5 ; z : int)

$\Longrightarrow$

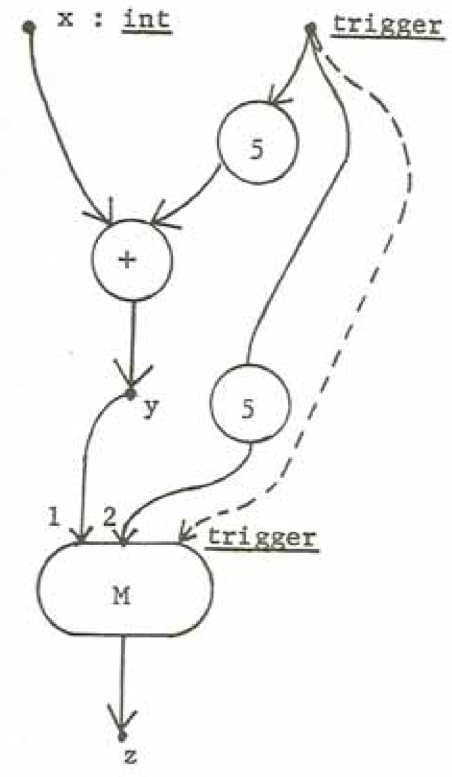


Figure 2.15 Examples of translations

(ii) For each assignment whose < exp > is not a module application (refer to (iv)) we create an acyclic graph of operators according to the rules of evaluation for the < exp >. We also connect the input arcs of operators to the links labelled by identifiers appearing in the statement. For a constant value we must create a constant function operator whose input is connected to the <u>trigger</u> node created in (i) (refer to figure 2.15).

(iii) For a conditional statement c : <u>if</u> B <u>then</u> P <u>else</u> Q <u>end</u>, the following actions are performed (refer to figure 2.16):

(1) Translate P to obtain the graph G(P) and the sets I(P) and O(P).

(2) Similarly for Q.

(3) We create an acyclic graph of operators for the boolean expression B, and connect the graph to links specified by I(B); the graph also has a boolean-value output arc which is connected to a link labelled b for distributing the value to gates created in (4), (5), and (6) below.

(4) For each id $\in$ I(P) we create a T gate which is connected to the links labelled id created in step (i) and step (iii)(1). The T gate has a control arc from the node b as defined in (iii) (3). We also create a T gate to connect the <u>trigger</u> link of G(P) to the link <u>trigger</u> in (i).

(5) Similarly for Q, except all gates are F gates.

(6) For each id $\in$ A = O(P) $\cap$ O(Q), a merge gate is created; the output arc is connected to the link nodes labelled as id created in step (i); the input arc on the side of the symbol T is connected to the link also labelled as id in G(P) created during the translation of P; similarly we connect the other input arc on the F side of the gate to the link id in G(Q) created when Q is translated. The control arc is connected to the link b.

(7) For each link node labelled by an id $\in$ O(P) $\cup$ O(Q) but not belonging to A (as in (6)) we connected it to a sink actor, since these outputs are not referenced outside of the conditional c.

A conditional statement

```
if  x > 0  then   5 → out ;
                  y + x → z
           else   w + y → temp ;
                  temp × 2 → z
           end
```

Translated conditional subschema

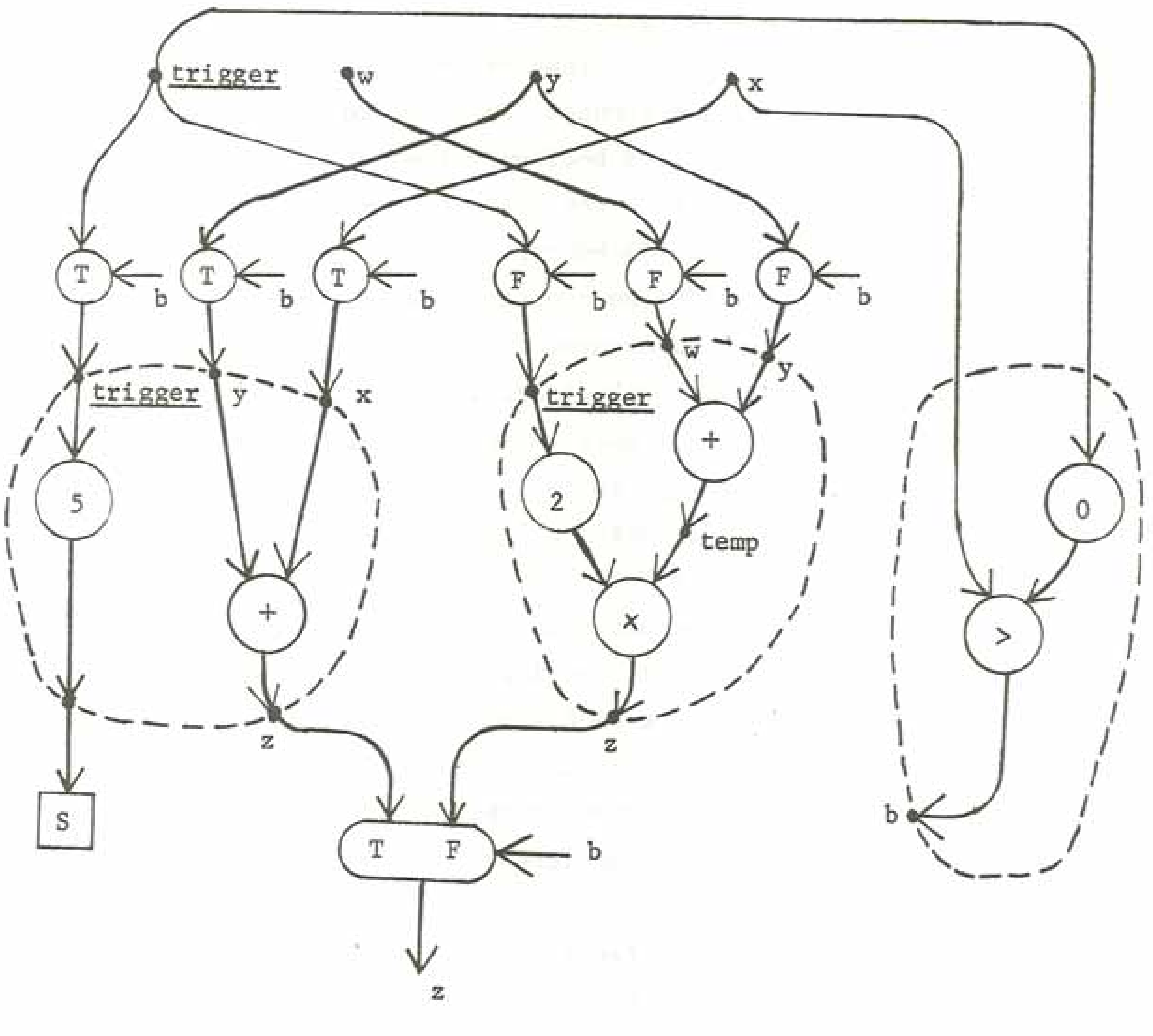


Figure 2.16 An example for the translation rule (iii)

(iv) For each module_call or module_application, in addition to connecting the module application actor to proper link nodes we must provide an extra input arc to the module_application actor from the trigger link node defined in (i). This is the result of the decision to translate a module of m inputs and n outputs into a data flow schema having one extra input link as trigger.

This concludes the translation rules for a list of statements. We noted that in step (iii)(7), sink actors are used to provide a place where tokens may be discarded. The translation rules are not intended to be optimal and in an implementation we may perform some steps concurrently and the translation of each statement may also be done in parallel.

We translate each module definition by first translating the statements L in the body to obtain the graph G(L) and the sets I(L) and O(L), then the set of links in I(L) corresponding to the input identifiers together with the special trigger link are chosen to be input links, and output links are selected from O(L) and all other links are connected to sink actors if they don't have any output arcs. We recall that for a program satisfying the constraints of single assignment and well-defined identifiers all identifiers in I(L) should be inputs of the module.

The graph translated from the body of a program should not have any input links except the link trigger, since there are no inputs to a program. The output links of the graph are all connected to sink actors.

The data flow modules corresponding to module definitions and the program consist of the translated graph and an initial configuration which is empty in the sense that no token is assigned to any arc. The execution of a program is to apply the data flow module to input actor which fires only once (see figure 2.17).

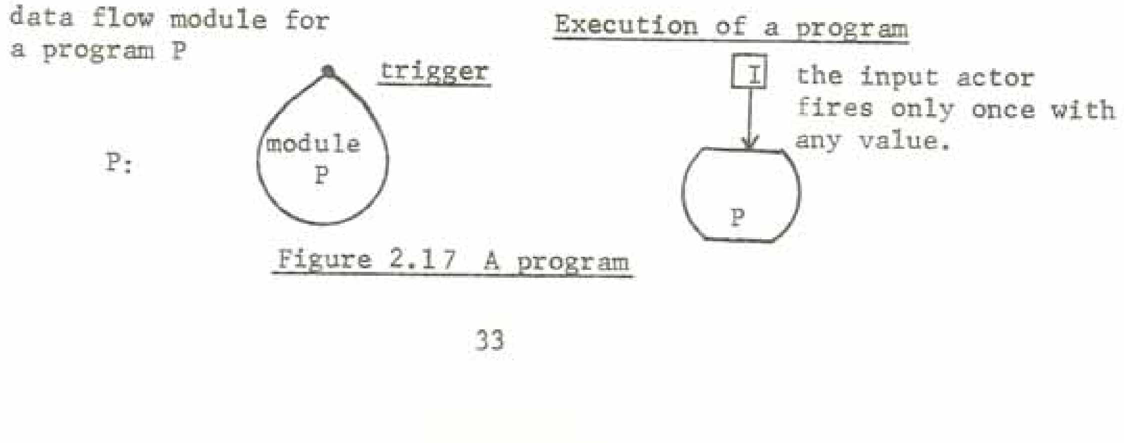

data flow module for a program P
P:
trigger
module P
Execution of a program
I
the input actor fires only once with any value.
P

Figure 2.17 A program

Summary

In this chapter we have defined TDFL by describing a translation rule of a program into a data flow module. The language, however, can be defined independently of data flow schemas. A program may be thought of as defining a function associated with each module and a set of identifiers representing some well-defined values according to the statements.

A simple module to compute the greatest common devisor using Euclid's algorithm is shown below:

```
Euclids: rmodule (X : integer, Y : integer; gcd : integer)
         if x = y  then      y → gcd
                 else
                     if  x > y then   x - y → z;
                                      Euclids (y,z) → gcd
                               else   y - x → z;
                                      Euclids (x,z) → gcd
                               end
                 end

          mend
```

Chapter 3

3. Streams

The types of computation expressible in data flow schemas encompass a large class of computations over sequences of values. In this chapter we describe an extension to TDFL which provides a basis for the expression of computation on "streams" (we shall use the word synonymously with "sequences") of values.

3.1 Motivation

In many programming applications we would like to describe a computation as a function defined over streams of input values and output values. For example, in applications involving signal decoding, a decoder may be described as a transformation on the sequence of input signals which produces a sequence of characters or words. This transformation in many cases can be though of as a function defined on sequences of signals or symbols. In computer systems input and output routines are more easily understood as computations over sequences of characters (or possibly sequences of compacted symbols). We can view the structural organization of a compiler, for instance, as having several phases.

These phases are often treated as a set of co-routines between which sequences of items representing syntactic components of the compiled program are passed. Thus a lexical analyzer receives a sequence of characters and generates a sequence of words, and a syntax analyzer may be constructed to receive a sequence of words and produce a sequence of data structures representing some syntactic component of a program such as a statement or a block.

Computations of this kind are often represented by co-routines. The advantages of co-routine structures are pointed out by Conway [7] and Knuth [27]. The co-routine primitives, however, are not suitable when parallelism is desired. This is a significant drawback since there generally is a substantial degree of parallelism in these computations.

The lack of suitable programming language constructs for such computations has motivated the use of data flow schemas as the basis for a programming language which can support computation on streams.

As pointed out in section 2.2, a data flow schema may be used to define a computation on sequences of values. Non-restricted use of data flow schemas, however, is not desirable. We therefore introduce some basic operations in TDFL as primitives from which programs for stream-oriented computation can be constructed. In what follows we specify the semantics for the extended TDFL and the rules of translation into data flow schemas.

3.2 Semantics for Streams

The extended syntax for TDFL is shown in figure 3.1 where the extensions of the syntax is enclosed by dashed boxes. For computation on streams, we define two new types <u>stream</u> <u>integer</u> (or <u>st</u> <u>int</u>) and <u>stream</u> <u>boolean</u> (or <u>st</u> <u>bool</u>) in addition to <u>integer</u> and <u>boolean</u> types. For convenience we shall refer to the <u>st</u> <u>int</u> and <u>st</u> <u>bool</u> types as <u>stream</u> types, and the <u>int</u> or <u>bool</u> types as <u>simple</u> types. An <u>empty</u> <u>stream</u> is a stream with no items. Syntactically, a stream-value is denoted as an ordered sequence of constants of the same type bounded by square brackets "[" and "]" with the ordering from left to right. Thus an integer stream consist of the integers 7,5,9and 11 in this order is written as [7,5,9,11]. An empty stream is therefore denoted by [ ], and a stream with the single integer 5 is denoted by [5].

```
< exp > ::= < arith_exp > | < boolean_exp > | < module_application > |
            < stream_exp >

< stream_exp > ::= < stream_operator > ( < in_list > ) |<stream_constants>
< stream_operator > ::= rest | con-s
< stream_constants > ::= < boolean_stream > | < integer_stream >
      < integer_stream > ::= [{ < integer >,}* < integer > ] | [ ]
      < boolean_stream > ::= [{ < truth_value >,}* < truth value > ] | [ ]

< arith_exp > ::= usual arithmetic expression | first < in_list >
< boolean_exp > ::= usual boolean expression | empty < in_list >
```

Figure 3.1 Extended Syntax for TDFL

In TDFL — In data flow schemas

[7,5,9,11] ⇒

[ ] ⇒

[5] ⇒

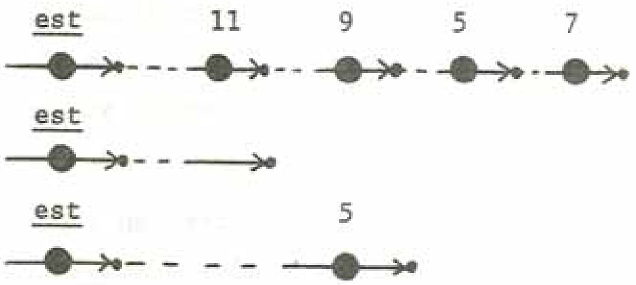


Figure 3.2 Examples and Representation in Data Flow Schema

In data flow schemas an integer stream is represented as a sequence of integer values followed by a special token designated as end of stream token (est). Figure 3.2 illustrates how a stream is represented.

In the extended TDFL we allow assignment of stream expressions to stream typed identifiers. We shall describe the semantics of the operations empty, first, rest, and con-s in terms of the effects on stream values. The notation and examples are shown in figure 3.3; data flow schemas defining these operations are shown in figure 3.4, where the operator eos is a test for the special value est whose output is true if the input token is est and false otherwise. The operator eos is not allowed in the extended TDFL and hence safeguards against ill-formed sequences of tokens. Note that these data flow schemas have non-empty initial configuration and when a sequence of tokens representing a stream passes through these operators the initial configuration is reestablished.

(i) empty

The operator empty is a predicate requiring an input of type stream and an output of type bool. If x is of type stream, then the value of empty(x) is true if x is an empty stream; otherwise it is false.

(ii) first and rest

The operator first requires an input of type stream which must not be an empty stream and yields a simple value which is the first item in the stream. The operator rest is also defined only on non-empty stream values; the output of rest when applied to a stream x is the stream obtained from x by removing the first item.

(iii) con-s

The operator con-s requires two inputs. Let x be of type stream and y be of the corresponding simple type (e.g., if x is typed as stream integer, then y must be of type integer); then the output of con-s (y,x) is a stream resulting from attaching y to the beginning of the stream x.

Thus, if z is defined as:

con-s (y: int, x: st int) → z : st int;

then first(z) yields the value designated by y and rest(z) yields the stream designated by x.

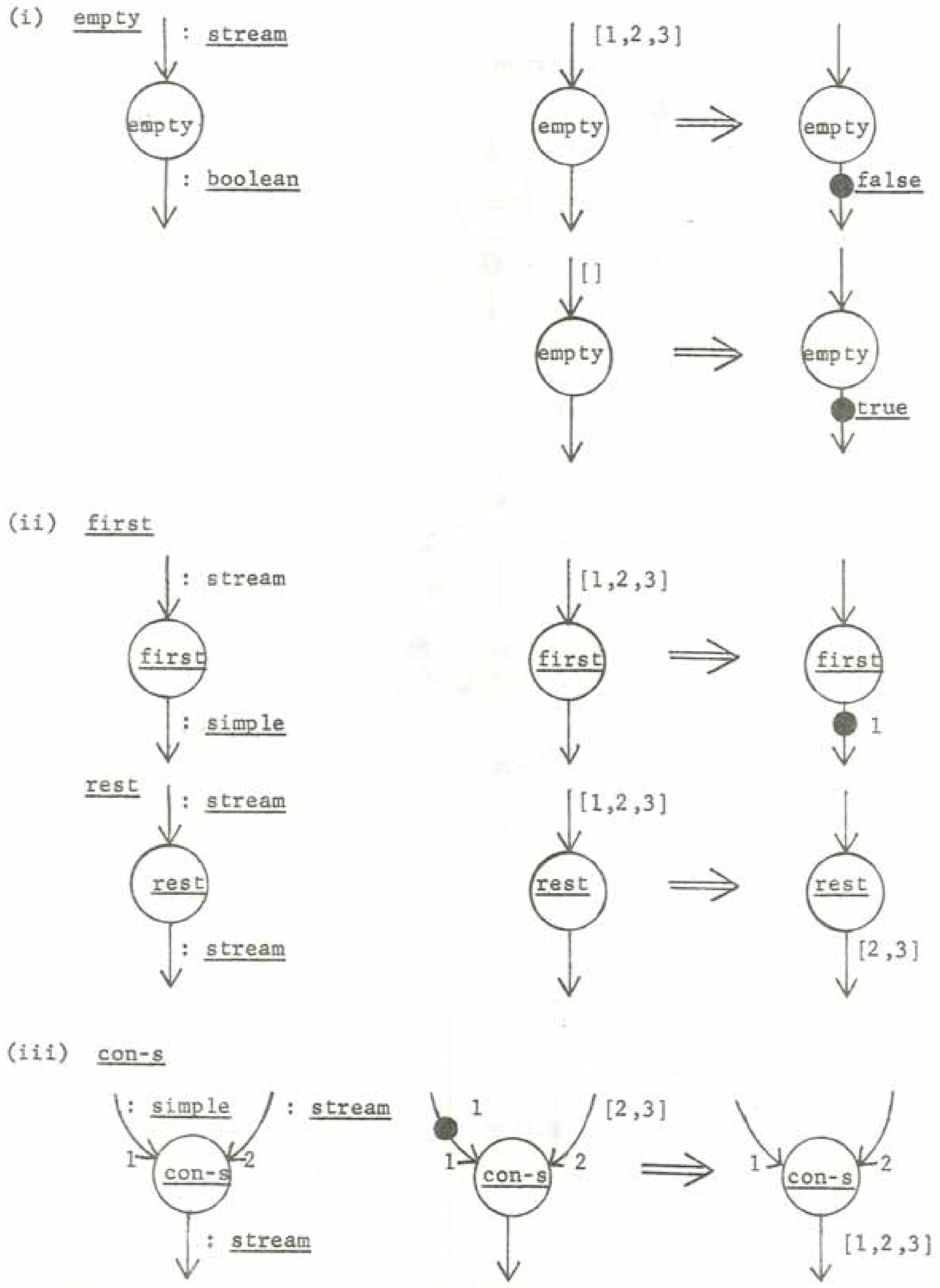


Figure 3.3 Notations and examples for operations on stream.

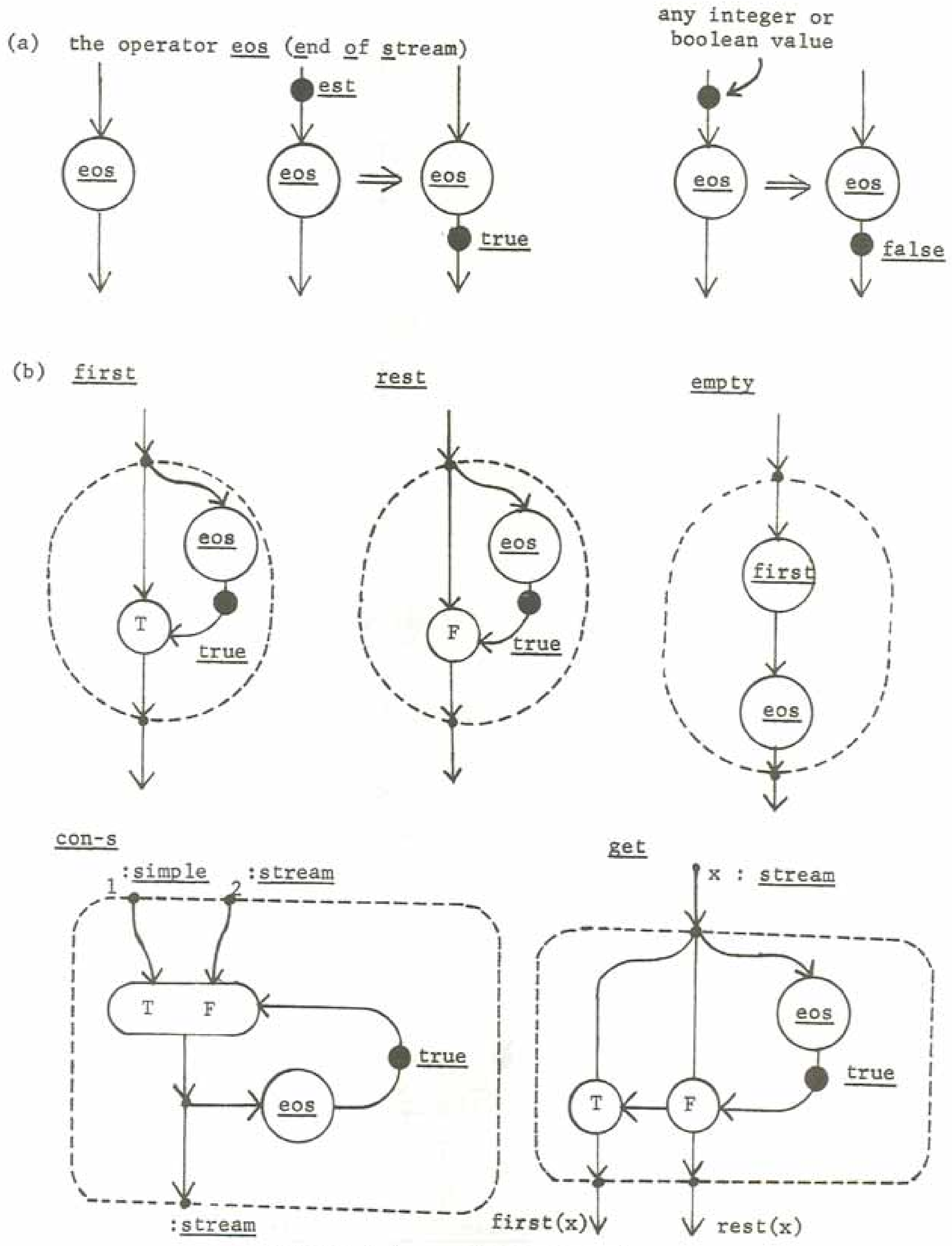


Figure 3.4 Stream operators

In figure 3.4 we also define an operator get which produces two outputs corresponding to first(x) and rest(x) when applied to the input stream x.

3.3 Translations

The translation rules for a program in the extended TDFL described in section 3.2 are basically the same as those described in section 2.3, with the exception of the following:

a. Constant stream values should be translated into data flow schemas which generate the stream when a token is received for triggering the generation of the sequence of tokens from the trigger link node.

b. Translation rules for conditional statements must be amended so that the semantics of an assignment statement embedded in a conditional will be properly defined as described below.

Constant Stream Operators

The translation rule for constant stream value is illustrated by an example shown in figure 3.5. The notation for a constant stream operator is simply to write the constant value in the operator. The operator est is a constant operator which generates the special token est. The reader should note that for an empty stream only the est token is generated.

Conditional Statements

The translation rules described in section 2.3 can be applied directly to a program using streams in the extended TDFL when there are no conditional statements. As described in the semantics, we would like to use an identifier to stand for a stream, the semantics of the conditional statement:

```
(α) if x > 0 then y: stream → z: stream
              else [2,3,4] → z
              end
```

is naturally understood as:

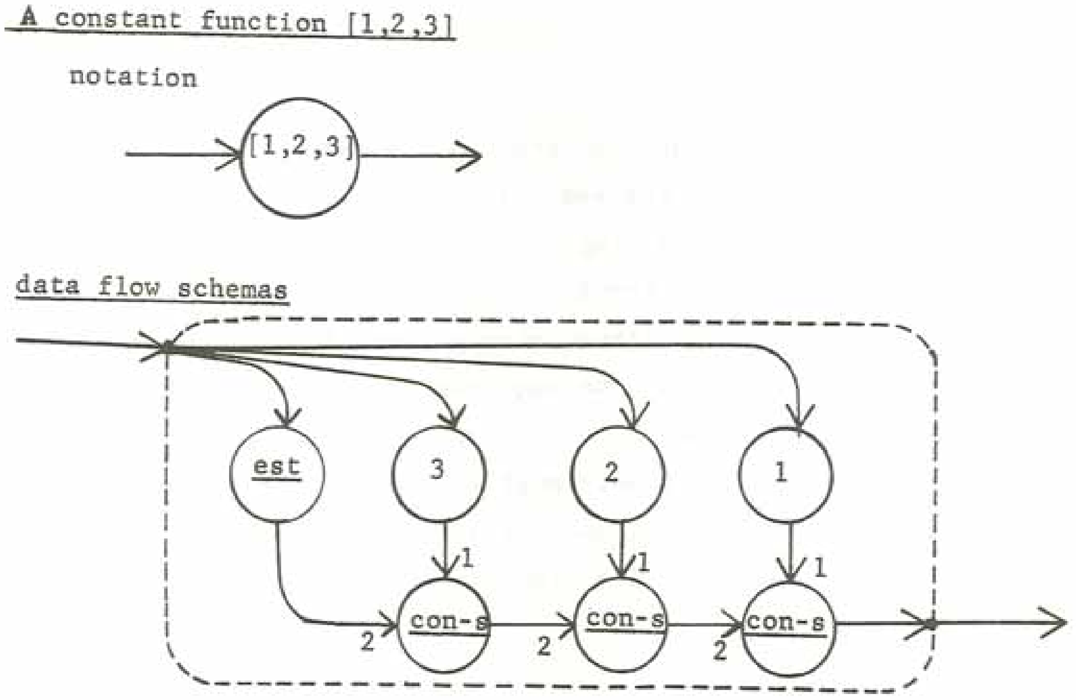


Figure 3.5 An example of constant function for a stream value

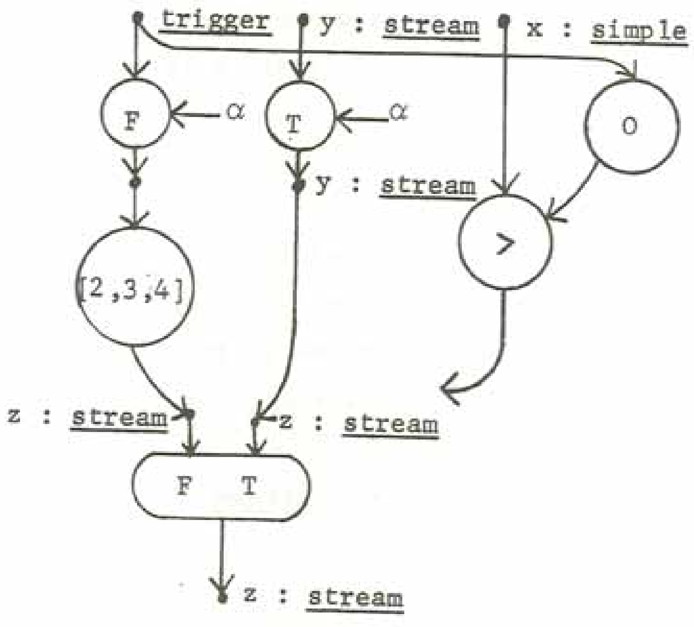


Figure 3.6 A conditional schema improperly translated.

(β) if x is greater than 0, then z is defined to have the value of y which is a stream, otherwise the constant stream value is assigned.

The translation rule when applied to the example above yields a data flow schema as shown in figure 3.6. From the discussion in section 2.3, readers should be able to verify that the boolean operator ">" receives only a single token from the link actor labelled as "x" during the computation of the program. Each gate actor, therefore, receives only one control token, i.e., a boolean value, which implied that the link actor z receives only one token rather than a stream.

To provide a translation rule to support the semantics exemplified by (β), we introduce new types of gates Ts, Fs, and Ms corresponding to T, F and merge gates, respectively. The notation and data flow schemas for the gates are shown in figure 3.7. The Ts and Fs gates allow a stream to pass upon receiving a proper boolean value. The Ms gate selects the input arc from which a stream value is to be passed to the output according to the boolean value received.

The translation rule (iii) specified in section 2.3 is modified in parts (4,), (5), and (6) as follows:

(4)' For each id $\in I(P)$ we create a T gate with proper connections if the id is of type simple, otherwise a Ts gate is employed.

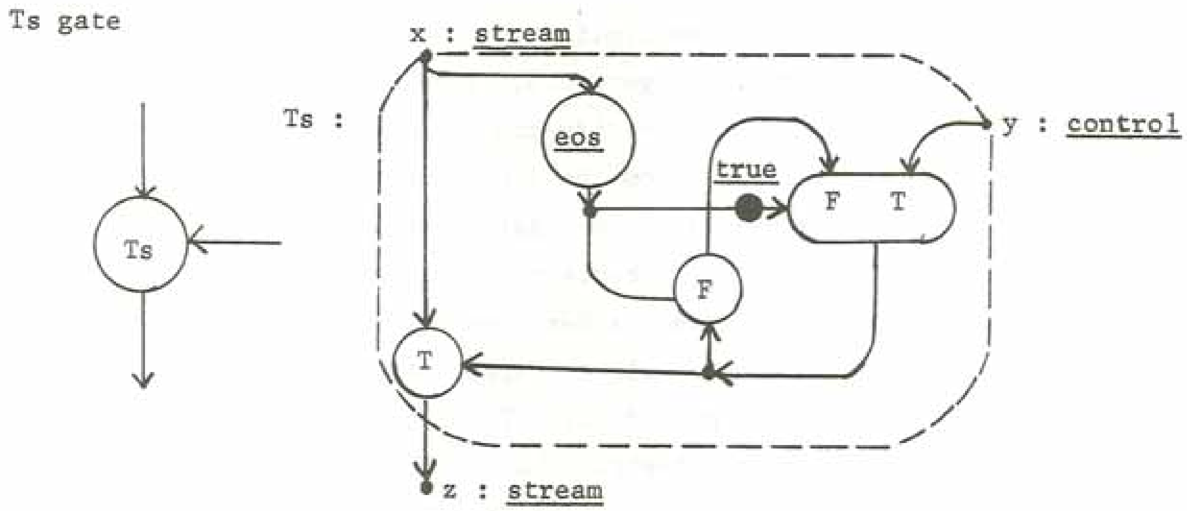
(5)' Similarly for Q except gates are either F or Fs.

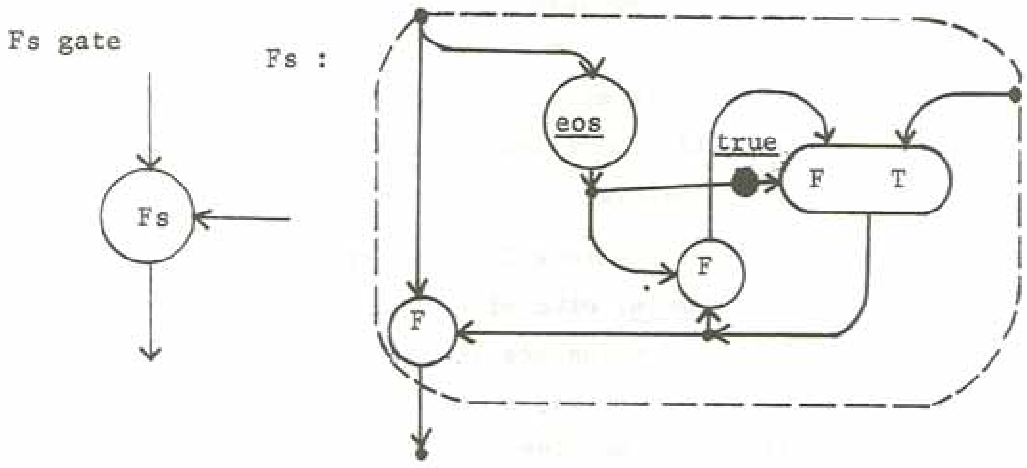
(6)' For each id $\in A = O(P) \cap O(Q)$, a merge gate is created if the id is of type simple, otherwise an Ms gate is used.

The result of applying the modified translation rule to the statement (α) is shown in figure 3.8. The reader should be able to verify that the translated data flow schema does implement the semantics described in (β).

3.4 Example Programs

In this section we demonstrate how computations on streams may be defined in TDFL. The examples chosen are based on computations on integer values. We believe that by extending the domain of the language to include string values and data structures, programs in other areas of application can be expressed with the same degree of clarity as that exhibited by the examples.





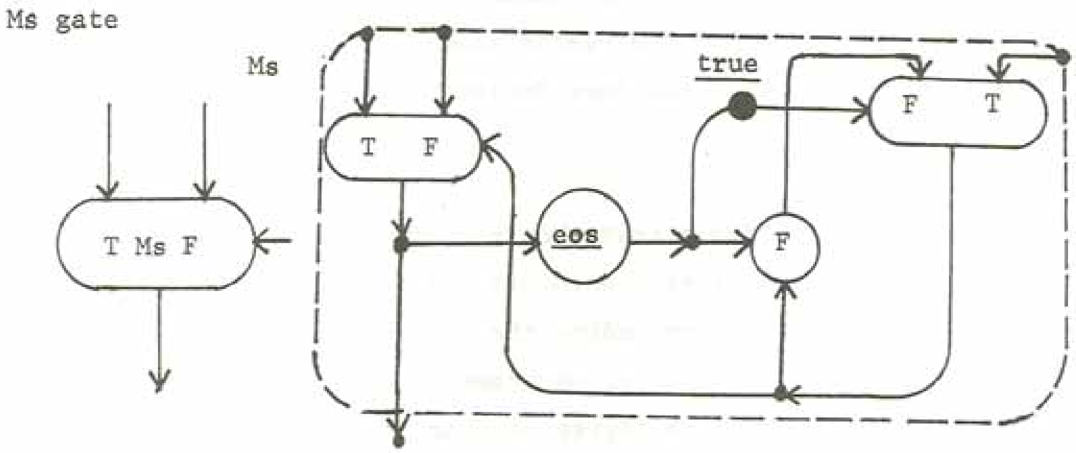


Figure 3.7 Gates for streams.

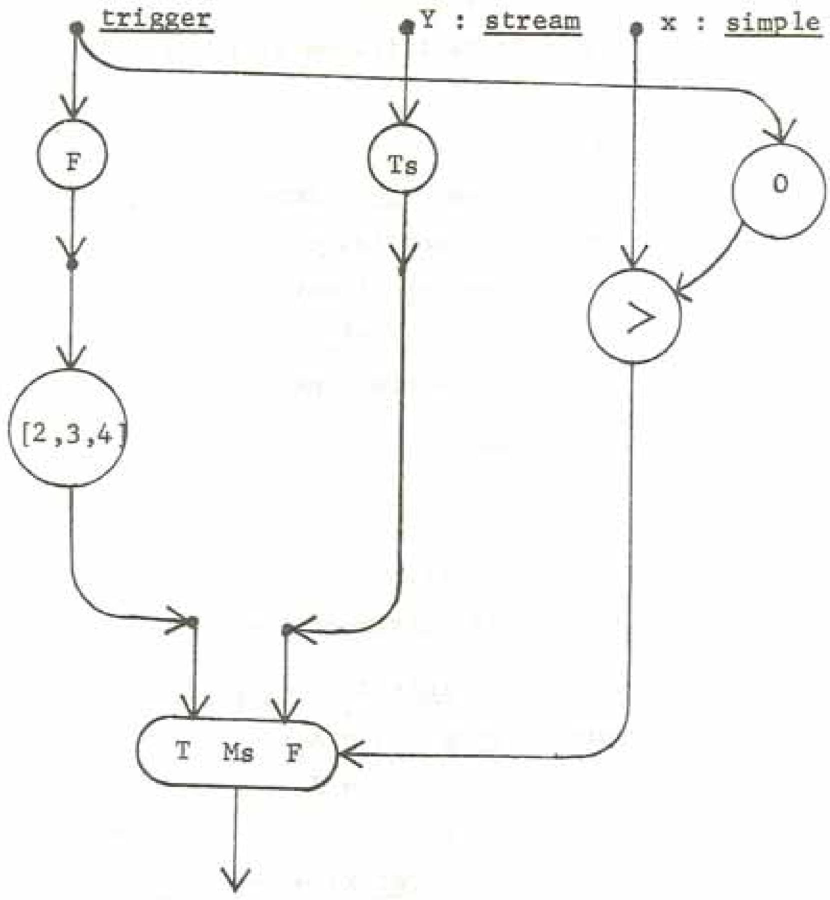


Figure 3.8 A properly translated data flow schema.

Example I. Computing the sum of integer values in a stream.

We shall present two programs. The first utilizes a straightforward method of adding each item in the stream to the accumulated partial sum. The other computes the sum by a method commonly known as "binary tree addition." Only the module definitions of these programs are presented.

Program Ia. Serial addition.

```
serial: rmodule(partial_sum : int, input : st int ; sum : int)
     if empty (input) then partial_sum → sum;
                      else get (input) → head : int, tail : st int
                           partial_sum + head → new_sum : int
                           serial (new_sum, tail) → sum;
                      end
     mend
```

Program Ib. Binary tree addition.

Firstly we define a module "alternate" as follows:

```
alternate : rmodule(X : st int; X1 : st int, X2 : st int)
            if empty(X) then [ ] → X1;
                             [ ] → X2
                        else
                             get(X) → head : int, tail : st int;
                             alternate (tail) → X2, X3 : st int;
                             cons (head, X3) → X1
                        end
```

The effect of applying "alternate" to an input stream is to produce two streams which are obtained by alternately assigning tokens in the input stream to each stream. Therefore, the application, alternate ([1,2,3,4,5]), yields two streams [1,3,5] and [2,4].

The module "alternate" is then employed in the module "binary-add", assuming that the sum of an empty stream is zero.

```
binary_add : rmodule (input : st int; sum : int)
          if empty (input) then 0 → sum
               else get (input) → head, tail;
                    if empty (tail) then head → sum
                                    else alternate (input) → x,y
                                    binary_add(x) + binary_add(y) → sum
                                    end
               end
               mend.
```

The module binary_add involves itself recursively if the input stream contains at least two items, otherwise it returns the value of the only item in the stream, or it returns zero as the sum if the stream is empty. The data flow schema for the module binary_add is shown in figure 3.9. The graph of the snapshot resulting from the application of the module to the input [1,2,3,4,5] is shown in figure 3.10, where for simplicity we do not show gates and boolean operators. The net of actors contained in the triangle with shaded lines distributes the numbers in the input stream to the binary-tree-like structure of "plus" operators.

Program II. Computing all primes less than n.

We shall use a variation of the method known as the sieve of Eratosthenes by representing the sieve as a stream of integer values.

The algorithm is described as:

a. Given input n : int, generate a stream of integer values consisting of a '2' followed by all odd numbers less than or equal to n in ascending order for $n \geq 2$.

For $n < 2$, an empty stream is generated.

b. Recursively delete multiples of primes using the module delete_np.

The module "generate" which performs what is specified in part a is described below:

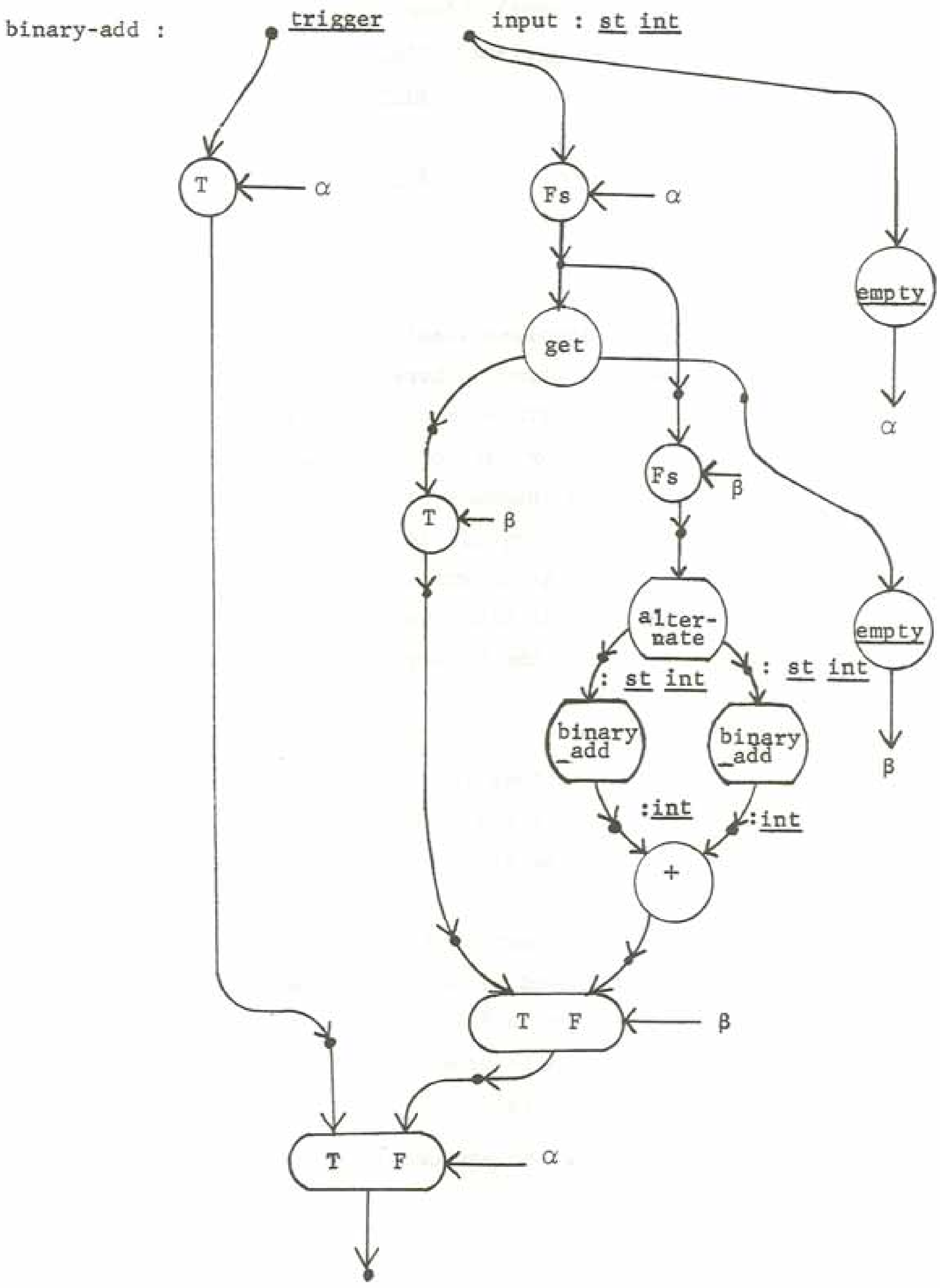


Figure 3.9 Binary-add

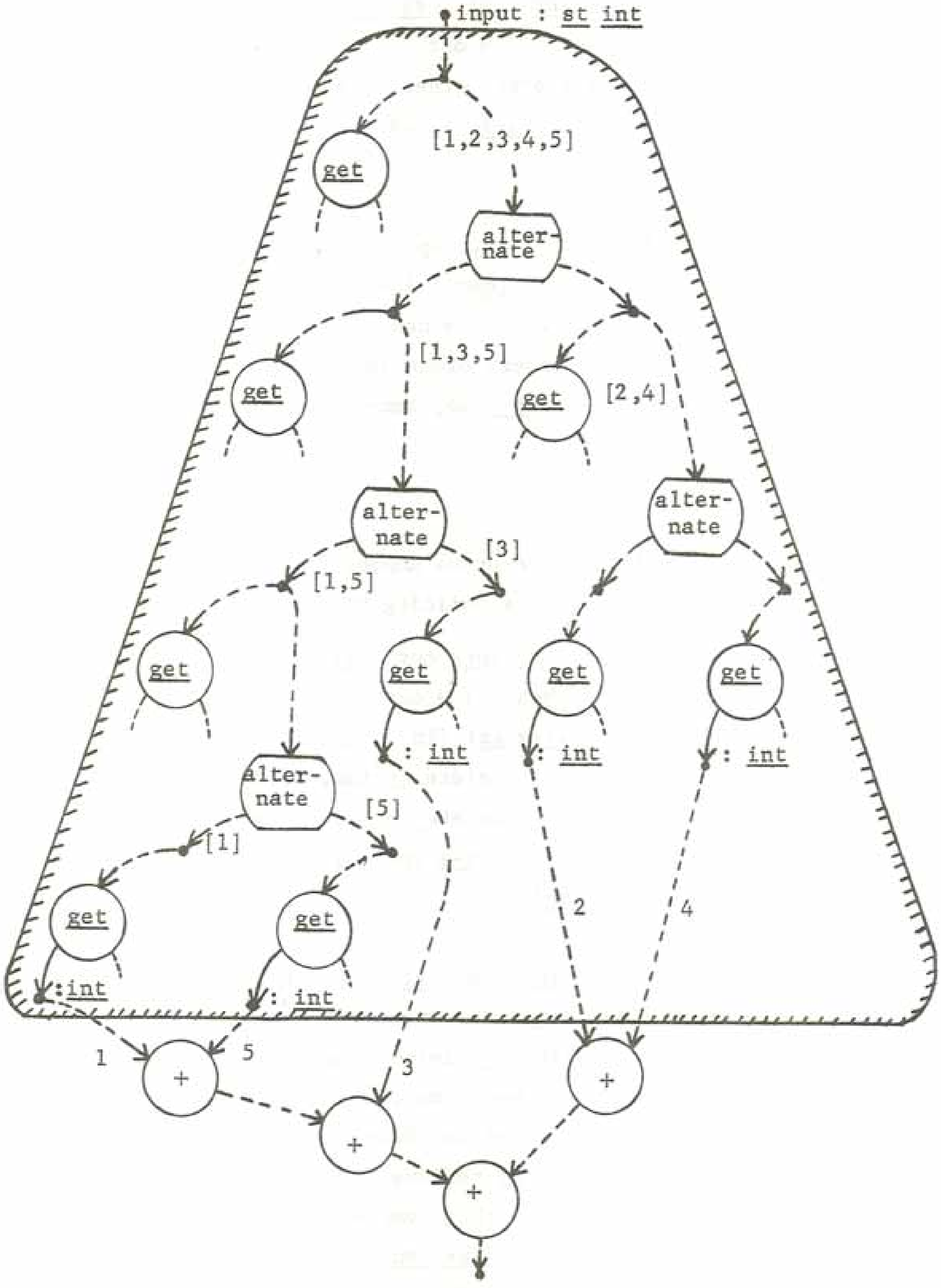


Figure 3.10 A snapshot for binary-add

```
generate : module (n : int; out : st int)
         if n < 2 then [ ] → out
                  else every_other (3,n) → odd_seq;
                       con-s (2, odd_seq) → out
                  end
         mend
every_other : rmodule (ℓb : int, up : int; out : st int)
                if ℓb > up then [ ] → out
                   else ℓb + 2 → next;
                        every_other (next,up) → temp;
                        con-s(ℓb, temp) → out
                   end
                mend
```

To perform deletion of non-prime numbers we define the module delete_np which uses "delete" as building blocks;

```
delete_np : rmodule (in : st int; out : st int)
          if empty (in) then [ ] → out
                        else get (in) → prime, tail;
                             delete (prime, tail) → new;
                             delete_np (new) → temp : st int;
                             con-s (prime, temp) → out
                        end
            mend
delete : rmodule (base : int, in : st int; out : st int)
         if empty (in) then [ ] → out
                       else get (in) → head, tail;
                            mod (head, base) → residue;
                            delete (base, tail) → temp;
                            if residue = 0
                              then temp → out
                              else con-s (head, temp) → out
                              end
                       end
         mend.
```

The module "mod" is the modulo function. The module delete_np simply removes the first item in the input stream and passes it as a prime number to the output; this prime is then used to remove non-primes in the remaining sequence of values by calling "delete".

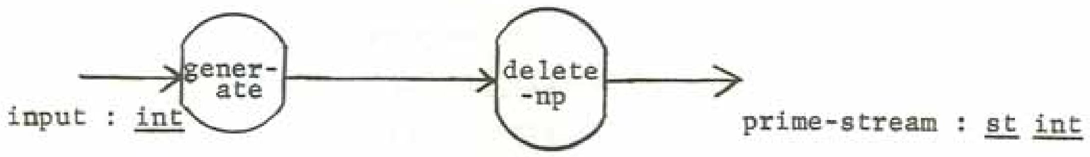
The main module "prime" is defined as:

```
prime : module (input : int; prime_stream : st int)
        generate (n) → integer_stream : st int;
        delete_np (integer_stream) → prime_stream
        mend.
```

The structure of the program is easily seen in a snapshot of the computation; again, gates and other boolean operators are not shown in figure 3.11 for simplicity.

Note that the parallelism is exhibited by the possibly concurrent firing of data flow operators in different activations of the module "delete_np."

(i)



(ii)

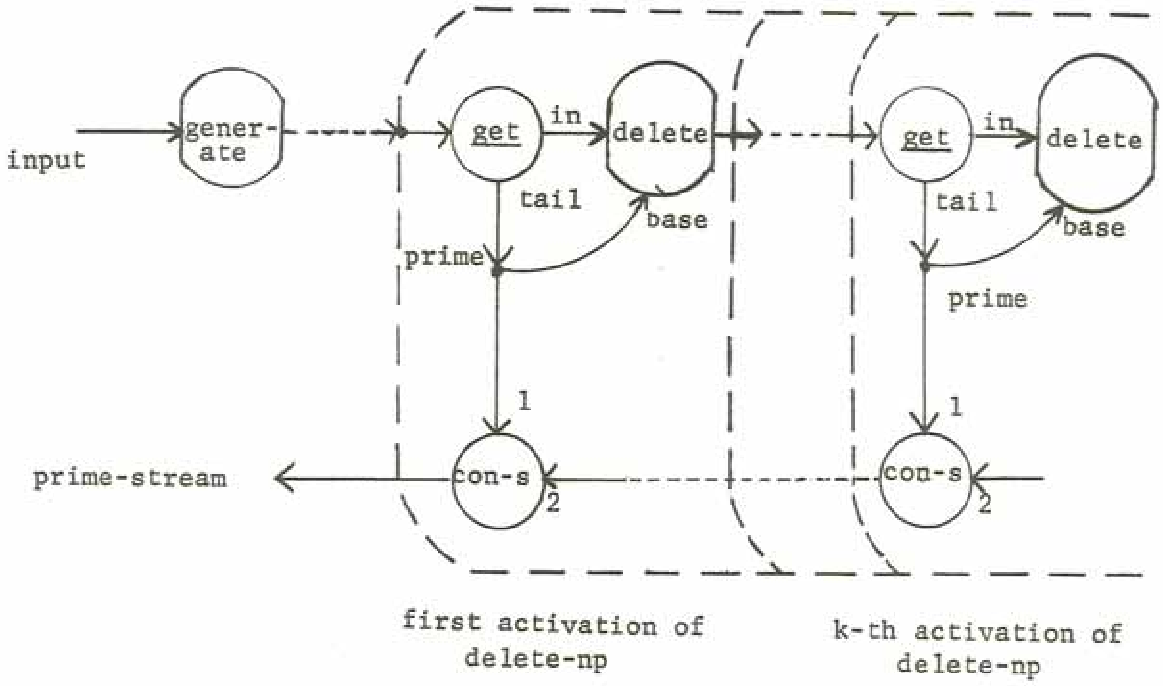


Figure 3.11 A snapshot for "prime"

Chapter 4

4. Communicating Modules

The extension of TDFL defined in chapter 3 allows us to define modules for computations over streams. The language, however, does not have semantic constructs for describing computations expressed as a system of interconnected modules communicating by passing data through communication channels. This notion of interconnected modules arises from our familiarity with the method by which we describe hardware systems and interprocess communication in operating systems.

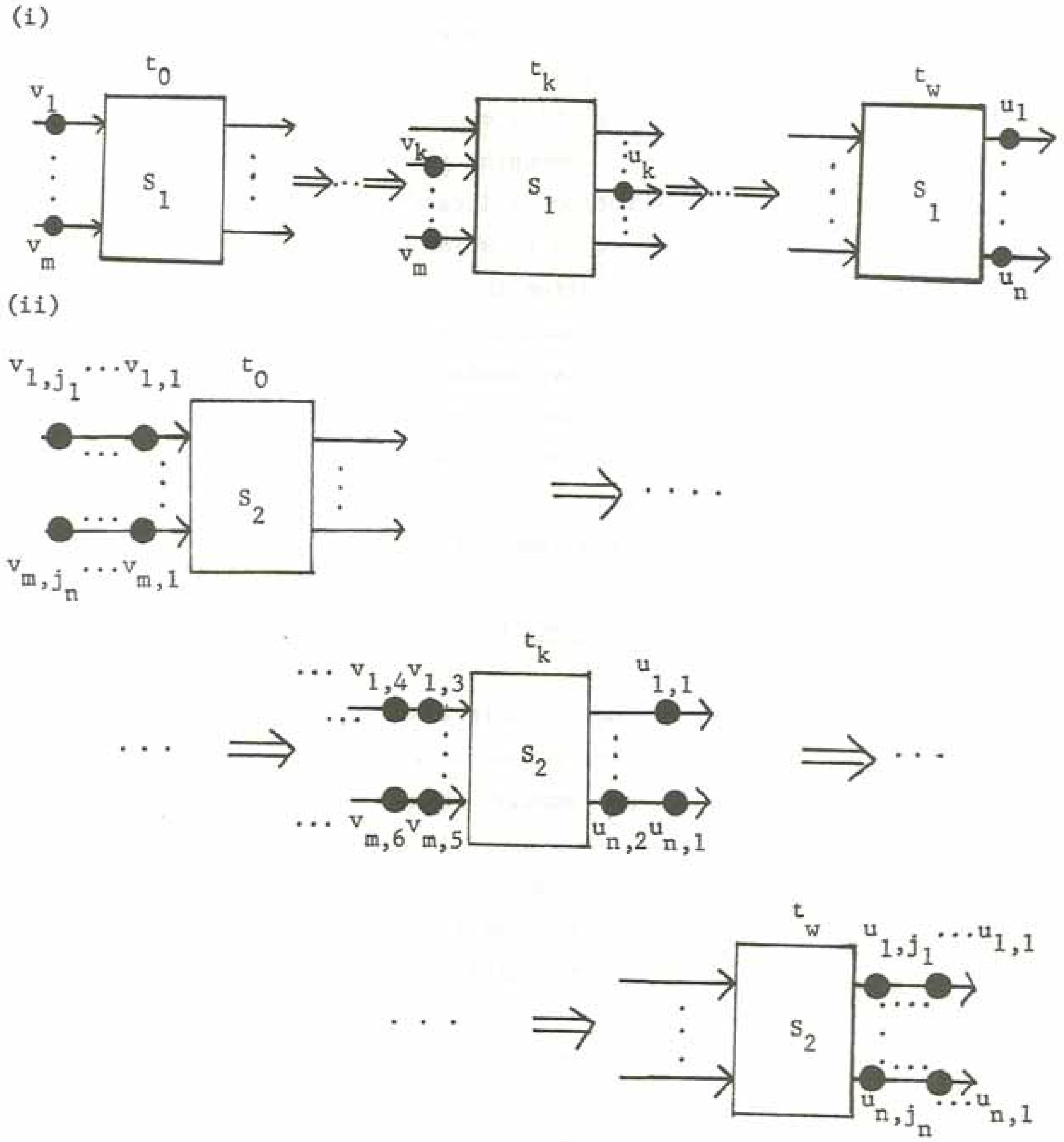
In this chapter we shall be concerned with systems which are determinate. We hope that proper semantics for expressing determinate systems may provide a firm foundation for achieving a better understanding for a more general class of parallel computation. We present a summary of some relevant results in the theory of parallel computation in section 4.1. These results justify our intention to provide a semantic construct for interconnected systems explicated in section 4.2. We also define the notion of _proper initialization_ and _proper termination_; programs having these properties are desirable for reasons detailed later.

It is evident that we may not be able to determine whether the initialization and termination are proper in general without significant analysis of the properties of each module. Therefore, an incompletely analyzed system may run into deadlocks. We propose in section 4.3 an extension of well-behaved modules. The systems constructed from these components which are well-behaved modules have a necessary and sufficient condition for proper initialization; translation into recursive data flow schemas can be defined as in section 4.3.2.

In section 4.4 we discuss several extensions which can be useful.

4.1 Properties of determinate systems

Intuitively a system is determinate if repeated application to the same set of input causes precisely the same set of outputs to be produced. In an asynchronous system the order in which each input is presented and each output is produced is immaterial provided the complete output is produced at some finite time after the complete set of input is assimilated (or absorbed) by the system. A generalization is to consider a system to be determinate when the inputs and outputs are sequences of values (see figure 4.1).



The system S_2 is determinate if for the same set of input sequences the set of output sequences produced is the same.

<u>Figure 4.1 Determinate systems.</u>

The intuitive concept of determinate system is formalized and the properties of interconnected systems whose components are determinate are studied by Patil [36]. We summarize some of the results below; readers are referred to Patil [36] for detailed definitions and further discussions.

<u>Closure property of determinate systems</u>

(i) Any finite interconnection of determinate systems is determinate provided the communication mechanism between them satisfies the <u>clash-free</u> property.

(ii) A communication mechanism is <u>clash-free</u> if: given any sequence of signals (or values denoted by the signals) observed at the sender's end of the communication mechanism, the same sequence is eventually observed at the other end and the receiving system is guaranteed to have assimilated all the signals in the same order.

(iii) Some of the clash-free communication mechanisms are
   a. fifo queues which may be finite or unbounded;
   b. ready and acknowledge signaling conventions.

As an example, actors of data flow schemas are determinate systems; and since the firing rules are defined as to be clash-free, any system of interconnected actors is determinate. The attractiveness of the closure property is the most important reason why data flow schemas are chosen as the basis for expressing determinate parallel computation.

The results above do not suggest how the collective behavior of the system can be abstracted from the behavior of the subsystems, and the closure property does not justify the use of recursive subsystems.

Recent developments in formal semantics have introduced lattice theory as a basis for defining semantics of programming languages (Scott [40], Strachey [41]). The theory has also been introduced as a theoretical base for determinate systems by Kahn [25]. In what follows we only summarize the results and discuss several implications which are relevant to further discourse.

In the formalization of determinate systems in lattice theory the communication mechanism consists of fifo queues of unbounded length. The result is the simplification of the characterization of determinate systems as continuous functions over lattices of signal sequences. The closure property of determinate systems is reformulated as:

> An interconnection of determinate systems also defines a continuous function over the lattice of signal history.

From the results of lattice theory, one can also define the continuous function characterizing the collective behavior of the system from the set of continuous functions which is the abstraction of the behavior of the subsystems. Furthermore, the use of recursive activation of subsystems is justified.

The unification of semantic basis for determinate systems and programming languages is a powerful argument for constructing parallel computation based on functions defined over signal sequences. The unification has the added advantage of making existing techniques for proving correctness of programs (Vuillemin [46]) applicable to proving properties of determinate systems.

In the actual implementation of a language supporting computation as suggested one must understand the implication of the requirement for unbounded fifo queues as the communication mechanism. If the implementation cannot provide sufficient computing resources to simulate the effects of unbounded fifo queues, the outputs of the determinate computation may be less than what could be expected otherwise (in the sense that the expected output is not completely produced). From a different perspective, a language designer should ensure that the communication mechanism provided by a language must be properly defined so that the semantics of communication mechanism satisfies the requirement.

It should be noted, however, that in such a language one must be able to prove that the program does not require unbounded computing resources.

4.2 Syntax and Semantics of Communicating Modules

The extended TDFL in section 3.2 can be used to specify any acyclic connection of data flow modules by module_calls or module_applications. Here we introduce the construct < perform_group > to describe a system of interconnected modules. Figure 4.2 shows the syntax where the body is comprised of a subset of statements allowed in TDFL. This restriction is only to simplify the complexity and may be eliminated when the basic semantics is understood. Throughout the rest of the discussion we shall conform to this restriction.

Semantics

A perform_group defines a "block" in the sense that all identifiers are local except those appearing in the interface. The identifiers which appear in the interface are "non-local" in the sense that they extend their scopes throughout the body of the module or the program containing the perform_group as a statement.

A perform_group can be defined to be a module by specifying a name. An identifier of simple type designates a link actor through which only a token carrying a simple value may pass.

Identifiers of type stream designate link actors through which a sequence of values forming a stream must pass.

```
< module > ::= < name > : < perform_group >
< statement > ::= < perform_group >
< perform_group > ::= perform < interface > < p_body > pend
< p_body > ::= { < perform_statement >;}* < perform_statement >
< perform_statement > ::= < module_call > | < limited_assignment >
< limited_assignment > ::= < limited_expression > → < id_list >
< limited_expression > ::= con-s ( < simple_primary >, < id > ) |
                           | < module_application > |<simple_primary>
< simple_primary > ::= < id > | < integer > | < truth_value >
```

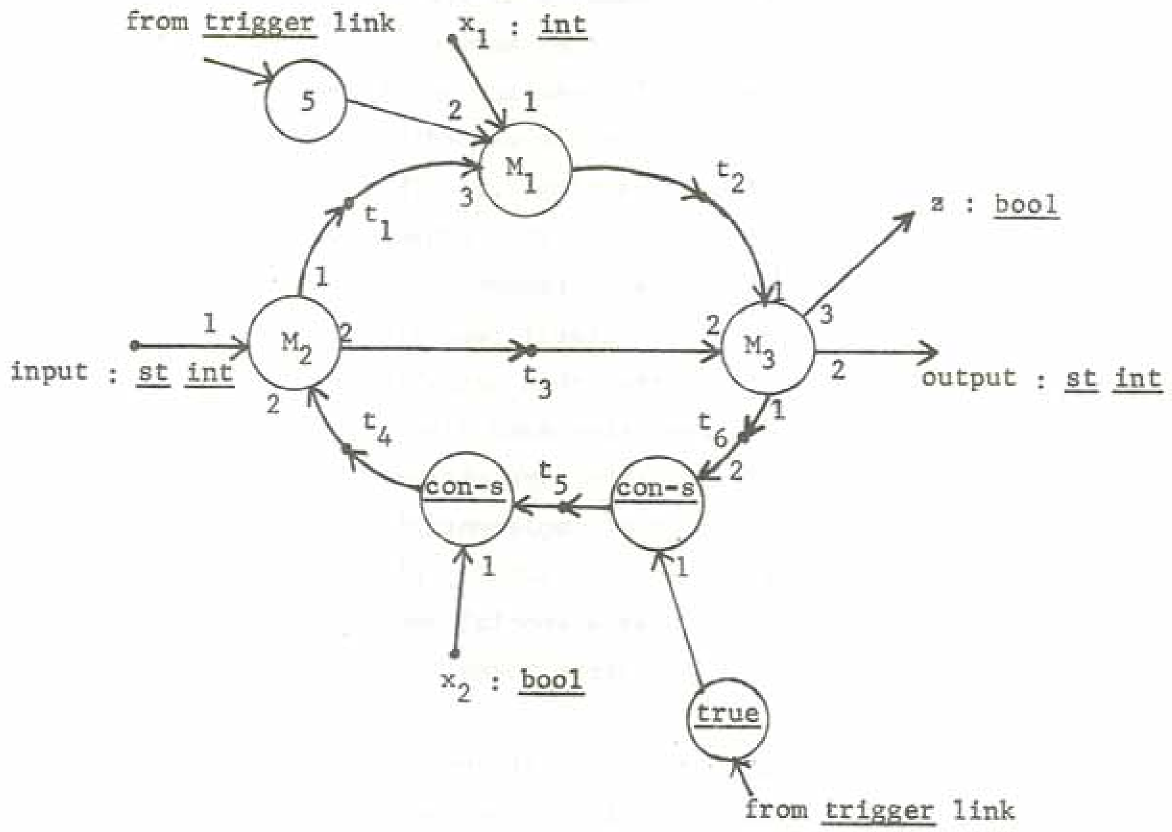
Figure 4.2

Extensions of syntax for < perform_group >

The statements in the body of a perform_group specify a data flow schema which may be cyclically connected. We do not enforce the requirement that an identifier must be defined in a statement before any reference to it within a perform_group; therefore, cyclic connections can be specified. We impose an additional constraint that data flow schemas specified by statements cannot have links of simple types on any cycle. Thus all links lying on a cycle must be of type stream.

Initializations for the system described by a perform group are defined by a set of assignments using con-s operators. For simplicity, this way of describing initialization may be preferred to the provision of new language constructs for this purpose.

Figure 4.3 shows an example of the use of a perform_group to specify a data flow schema; notice that the box enclosed by broken lines contains statements which are used for specifying initializations, and that the constant values correspond to constant value actors whose inputs are from the trigger link. The reader may have noticed that the modules are not



perform-group specification of above system

```
perform (input : st int, $x_1$ : int, $x_2$ : bool; output : st int, z : bool)
            $M_2$ (input, $t_4$; $t_1$,$t_3$);
            $M_1$($x_1$,5,$t_1$) $\rightarrow$ $t_2$;
            $M_3$($t_2$,$t_3$; $t_6$, output, z);

      - - - - - - - - - - - - - - - - - - - -
      |     con-s (true,$t_6$) $\rightarrow$ $t_5$;        |
      |     con-s ($x_2$,$t_5$) $\rightarrow$ $t_4$          |
      - - - - - - - - - - - - - - - - - - - -
            pend
```

Figure 4.3 A perform group

diagrammed as module_application actors. This is because an important discrepancy exists for the semantics of the module_calls and module_applications in perform_groups: they should be understood as macro-expansions in the conventional sense and not as module_application actors which are replaced by the corresponding modules according to the firing rule specified in section 2.2. That is, module_calls are regarded as specifications of data flow schemas whose module_application actors are all replaced by the corresponding data flow schemas. Because a data flow schema may involve itself recursively, this replacement process must be simulated rather than actually performed. Semantically, however, we may regard a recursive data flow schema specified in a perform_group as an infinite structure obtained by the above-mentioned replacement process. (A correct implementation for simulating the replacement process is to regard a module_call or module_application specified in a perform_group as a special module_application actor which is enabled whenever the first token arrives at one of its input arcs.)

We explain the reason why this discrepancy arises using a very simple example shown in figure 4.4. The module M_1 simply multiplies the inputs by two. The module M_2 adds pairwise integers from two input streams. The module M is constructed as shown in the diagram. For a module_application actor of M, the firing rule is that the actor is not enabled until one token is present on each input arc. As a result no output arrives at the link z until some token arrives at y. If the module M is substituted in place of the module_application actor, (refer to figure 4.4 (ii), where we have introduced the notation for denoting a substituted module), z would receive some outputs some time after M_1 receives inputs from x regardless of whether any input has arrived at the input y.

```
M₁ : rmodule (x : st int; z : st int)
     if empty(x) then    [ ] → z
                 else get(x) → head, tail;
                      M₁(tail) → more;
                      con-s(2 x head, more) → z
                 end
     mend

M₂ : rmodule (y: st int, z : st int; w : st int)
     if empty(y) ∨ empty(z)
                 then    [ ] → y
                 else get(y) → y₁,y₂;
                      get(z) → z₁,z₂;
                      M₂(y₂,z₂) → w₁;
                      con-s(y₁ + z₁, w₁) → w
                 end
     mend

M : module (x : st int, y : st int ; z : st int, w : st int)
     M₁(x;z);
     M₂(y,z;w)
     mend
```

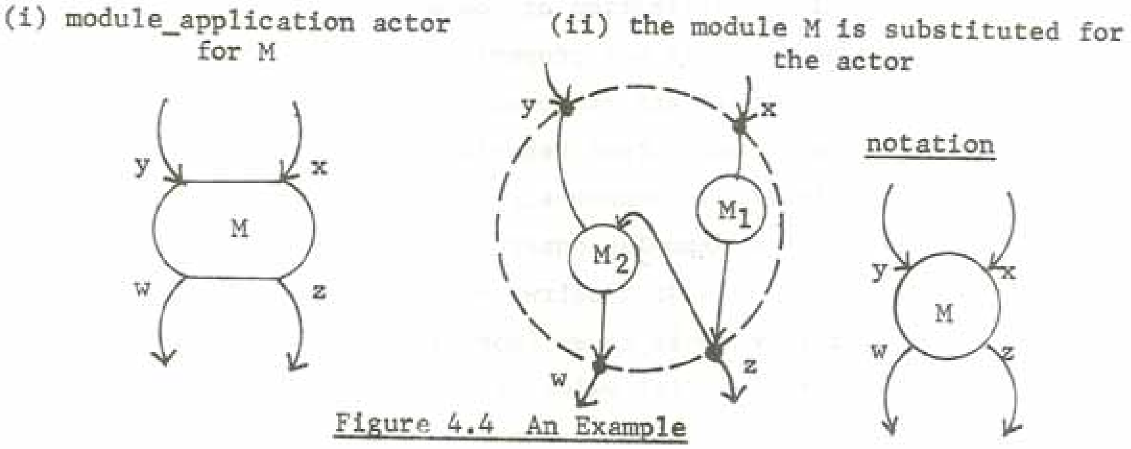

(i) module_application actor for M
y
x
M
w
z
(ii) the module M is substituted for the actor
y
x
M1
M2
w
z
notation
y
x
M
w
z

Figure 4.4 An Example

Discussion

Modules defined over streams provide a semantic basis for describing an interconnected system satisfying the requirement that the modules communicate by unbounded fifo queues. This claim is justified by noting that within each module the chains of stream operators actually simulate unbounded fifo queues. Therefore, provided that during the computation the computing resources do not exceed physical limits, the whole system can be characterized as a continuous function as suggested by Kahn [25].

We shall define two concepts which are important to the further discussion.

Def. Let S be a determinate system in which some links are designated as inputs and outputs. Then S is properly initialized if the computation does not deadlock (in the sense of reaching a configuration in which no actors are enabled) before all outputs produced are terminated by end of stream tokens (est), and S is properly terminating if after all outputs are terminated by end of stream tokens, the computation does not undergo infinite number of firing of actors.

Figuratively, if one waits at the outputs of a system which is not properly initialized, then there is no way of knowing when no more outputs are to be produced. The situation of indefinite waiting is also undesirable because of the inefficient utilization of computing resources caused by deadlocks. A system which is not properly terminating may run indefinitely after all outputs are produced. Since a system is defined only in terms of the input and output behavior, all computations performed after outputs are completed are unnecessary and a waste of resources.

Figure 4.5 shows three examples constructed from the module M which produces an output stream by adding pairwise the integers in the two input streams. The first example is an improperly initialized system because of the lack of any initial value for the second input of M. The example (ii) shows the situation where the sequence of tokens provided by the cyclically connected module never terminates. In the last example (iii), an initialization is provided for the system S_3; the system is both properly initialized and terminating because it is guaranteed to produce a stream as its output and no more computation is initiated after the module M receives an est from the first input "in."

The properties of proper initialization and termination are easy to check for the particular examples shown because the behavior of the module M (in terms of the number of inputs required to generate some number of outputs) is independent of the values of inputs. This is not the case in general since the behavior of a module may be highly dependent on the input values.

```
M : rmodule (x : st int, y : st int ; sum : st int)
        if empty (x) ∨ empty(y) then   [ ] → sum
                                else get(x) → first_x, rest_x;
                                     get(y) → first_y, rest_y;
                                     first_x + first_y → temp;
                                     M(rest_x, rest_y) → rest_sum;
                                     con-s(temp, rest_sum) → sum
                                end
        mend
```

(i) improper initialization

```
    S_1 : perform ( in : st int ; out : st int )
                 M ( in, loop ; loop ) ;
                 loop → out
          pend
```

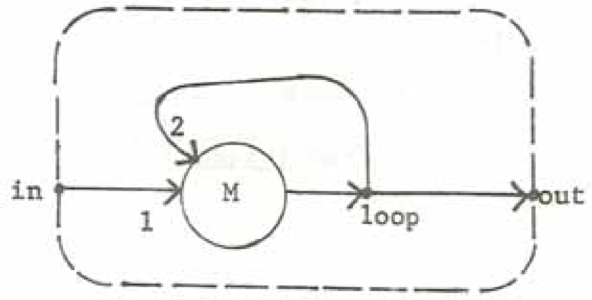
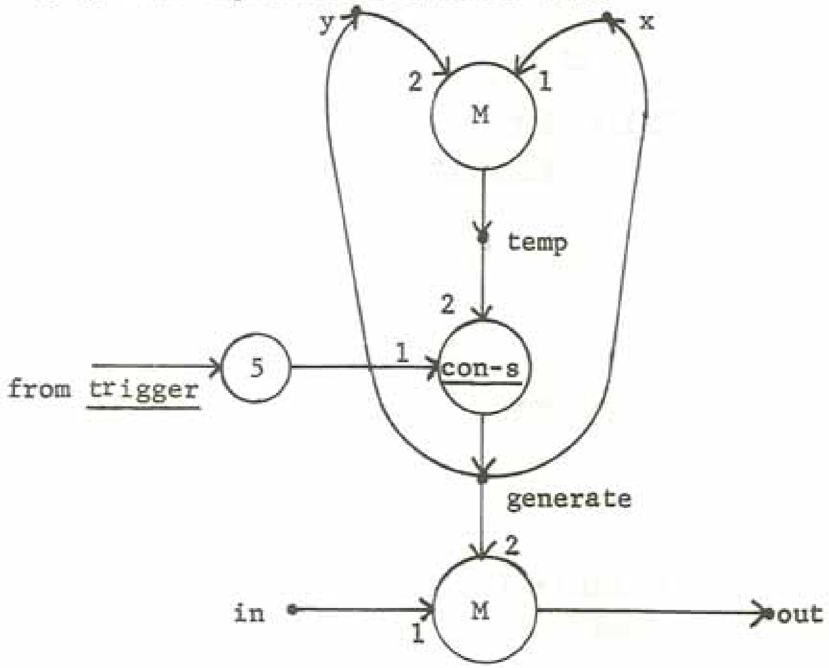


Figure 4.5 Examples for the Intitialization and Termination

(ii) proper initialization but improper termination

```
S2 : perform (in : st int ; out : st int)
        M(x,y ; temp) ;
        con-s (5, temp) → generate ;
        M (in, generate; out) ;
        generate → x,y
    pend
```

The graphical representation looks like:



(iii) proper initialization and termination

```
S3 : perform (in : st int ; out : st int)
        M (in, loop ; out) ;
        con-s (5, out) → loop
        pend
```

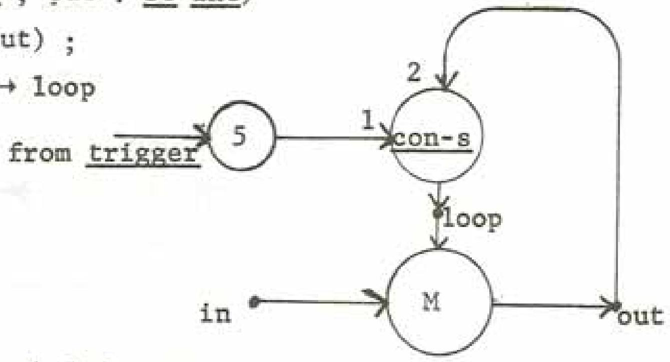


Figure 4.5 (continued)

4.3 Well-Behaved Modules

In this section we will be concerned with a system specified by a perform_group in which all modules are either defined over simple values or are extended well-behaved modules as described in what follows.

4.3.1 Semantic Extensions

In this section we define a semantic extension of well-behaved modules. so that interconnected systems can be constructed from these modules. Let M b e an (m,n) module whose inputs and outputs are of simple type, then the extension of M, denoted as M' is defined over stream values as follows: (Note that the module M may incorporate submodules which use streams. The requirement that all inputs and outputs are of either boolean or integer type means that the module M must be well-behaved.)

```
¢ all identifiers $x_1,\dots,x_n$ and $y_1,\dots,y_n$ must be of consistent types
  with the corresponding parameters of the module M ¢

        M' : rmodule ($x_1$ : st,...,$x_n$ : st ; $y_1$ : st,...,$y_n$ : st)
              if empty($x_1$) $\vee \dots \vee$ empty($x_n$)
                 then [  ] $\rightarrow$ $y_1,\dots,y_n$
                 else get ($x_1$) $\rightarrow$ first_$x_1$, rest_$x_1$;
                          ⋮
                      get ($x_n$) $\rightarrow$ first_$x_n$, rest_$x_n$;
                   M(first_$x_1$,...,first_$x_m$; first_$y_1$,...,first_$y_n$);
                   M'(rest_$x_1$,..., rest_$x_m$; rest_$y_1$,..., rest_$y_n$);
                   con-s (first_$y_1$, rest_$y_1$) $\rightarrow$ $y_1$;
                     ⋮
                   con-s (first_$y_n$, rest_$y_n$) $\rightarrow$ $y_n$
                 end
              mend.
```

Therefore the extended module produces output streams whose length, i.e., the number of simple values in a stream, is the same as the shortest stream presented at the inputs. A simple example is shown in figure 4.6.

```
M : module (n : int , m : int ; sum : int)
        m + n → sum
    mend
```

Then the assignment statement:

M'([1,2,3], [1,2]) → z : st int

produces as the result value of z, [2,4].

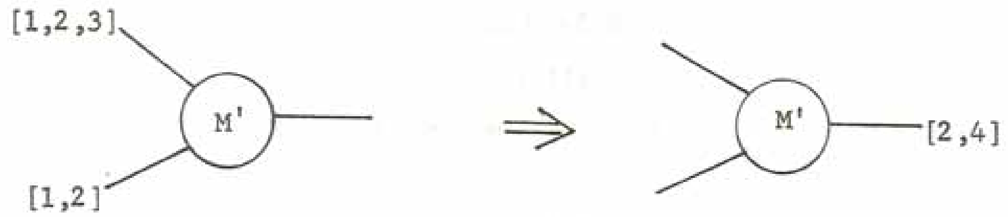
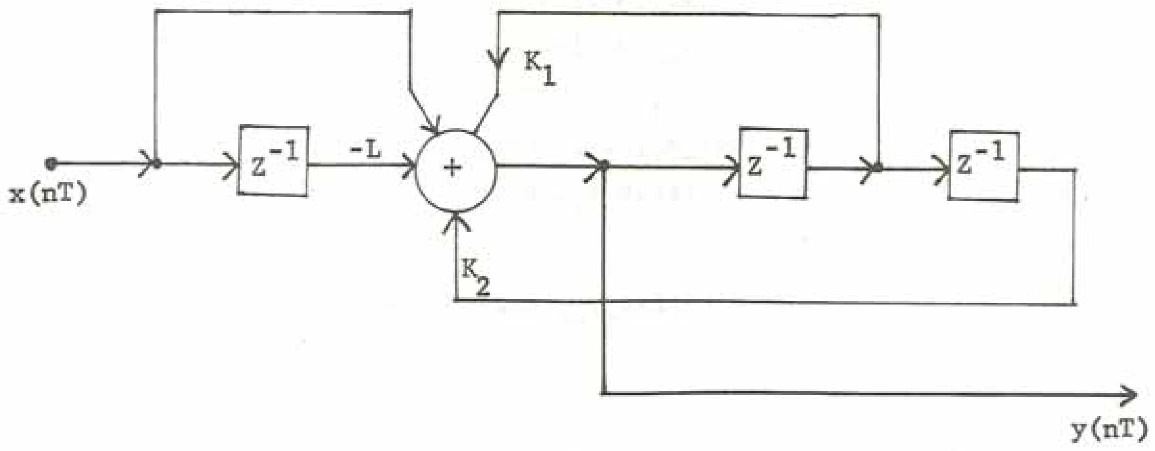


Figure 4.6 An Illustration for an Extended Module



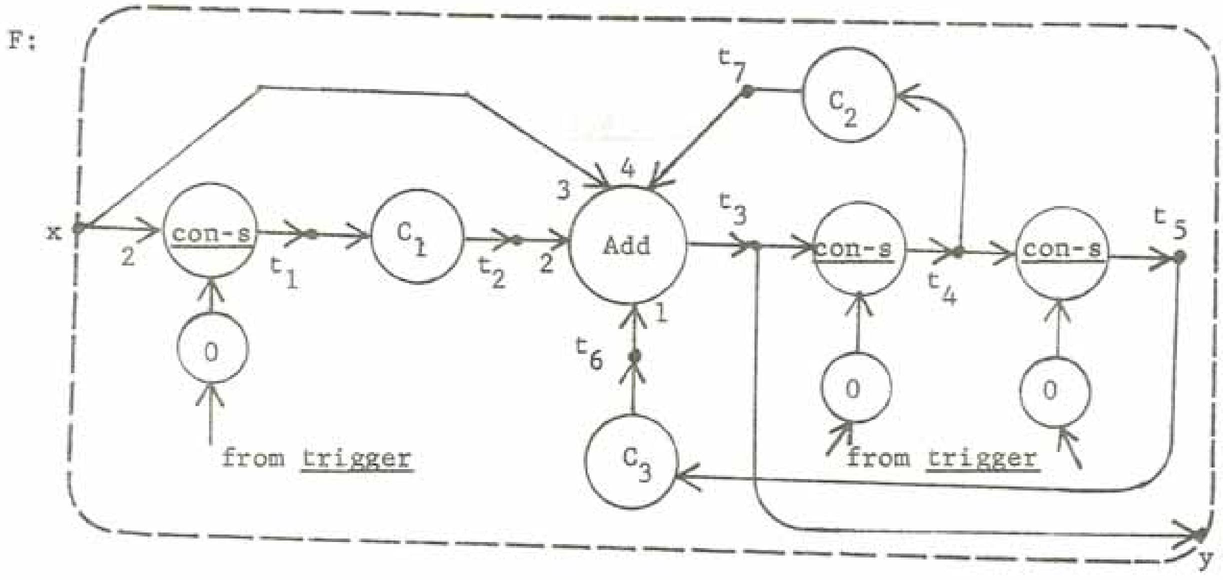
The second-order difference equation is

$y(nT) = k_1y((n-1)T) + k_2y((n-2)T) + x(nT) - Lx((n-1)T)$, where the initial values for y(-T), y(-2T), and x(-T) are zero.

Figure 4.7 An Example from Signal Processing Application (a second order digital filter).

We believe that the proposed extension is a natural one and semantically it is also easy to understand. We should mention that one of the most important properties of the extension is that the checks for proper initialization and termination can be done without detailed analysis of the module M.

The utility of the extended well-behaved module is seen in the example borrowed from the descriptive method of a digital filter in signal processing applications (see figure 4.7). The operator z^{-1} designates unit time delay which is simply an arc in the corresponding diagram drawn in the data flow schema form, figure 4.8. The constants k_1, k_2, and -L along the arcs are to be interpreted as the constant factors by which the values passing through the arcs must be multiplied; the equivalence of a constant factor is shown as a module in figure 4.8. The initialization is shown in figure 4.8 by <u>con-s</u> operators. The perform-group F defining the system shown in figure 4.8 is given in figure 4.9. As the reader may notice, the restricted syntax for perform_group results in rather cumbersome module definitions for c_1, c_2 and c_3. We believe that proper syntactic sugaring may eliminate this problem.



<u>Figure 4.8</u>

```
$C_1$ : module (x : int ; y : int)
          (-L) $\times$ x $\rightarrow$ y
        mend
$C_2$ : module (x : int ; y : int)
          $K_1$ $\times$ x $\rightarrow$ y
        mend
$C_3$   module (x : int ; y : int)
          $K_2$ $\times$ x $\rightarrow$ y
        mend
Add: module ($x_1$ : int , $x_2$ : int , $x_3$ : int , $x_4$ : int ; y : int)
          $x_1 + x_2 + x_3 + x_4 \rightarrow y$
        mend
F : perform  (x : st int ; y : st int)
        con-s(0,x) $\rightarrow$ $t_1$; $c_1'(t_1)$ $\rightarrow$ $t_2$;
        con-s(0,$t_3$) $\rightarrow$ $t_4$; $c_2'(t_4)$ $\rightarrow$ $t_7$;     ¢ all identifiers are of type
        con-s(0,$t_t$) $\rightarrow$ $t_5$; $c_3'(t_5)$ $\rightarrow$ $t_6$;       st int ¢
        Add'($t_6$,$t_2$,x,$t_7$) $\rightarrow$ $t_3$;
        $t_3$ $\rightarrow$ y
      pend
```

Figure 4.9

Proper Initialization and Proper Termination

An extended well-behaved data flow schema behaves like an actor which is enabled when a token is present at each input arc. If no est token is present, the effect of firing is to produce output values from the input values according to the well-behaved module. If any input token is an est, the effect of firing is to absorb all tokens arriving at each input and to put one est token on each of the outputs.

Figure 4.10 illustrates the behavior of the extended module M' described above. The diagram (i) shows the firing rule when a token carrying a simple value arrives at each input of M', the outputs $u_1,\ldots,u_n$ is the same as what would be produced by M if the input $v_1,\ldots,v_m$ were given. The diagram (ii) shows the situation when, for the first time, any of the inputs to M' is an est token (there may be more than one such input). The diagram (iii) exhibits the behavior of M' after the situation described by (ii) has occurred: note that some of the arcs may not have any tokens if they have already received est tokens and the input tokens are simply absorbed without any more output being produced.

There is an almost exact correspondence between a system of extended well-behaved modules and a marked graph (Hack [20]) except that the behavior of the extended module after receiving the first est token is different. This difference, however, does not invalidate the applicability of the necessary and sufficient condition for liveness to determine whether a system is properly initialized. We quote the necessary and sufficient condition for liveness of a marked graph: (We refer the reader to Hack [20])

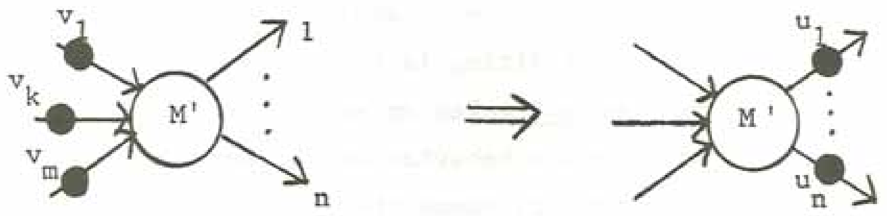
> Let G be a marked graph, then the initial marking is live if and only if there is at least one token on each directed cycle of G.

The corresponding necessary and sufficient condition for proper initialization of a system S is the following:

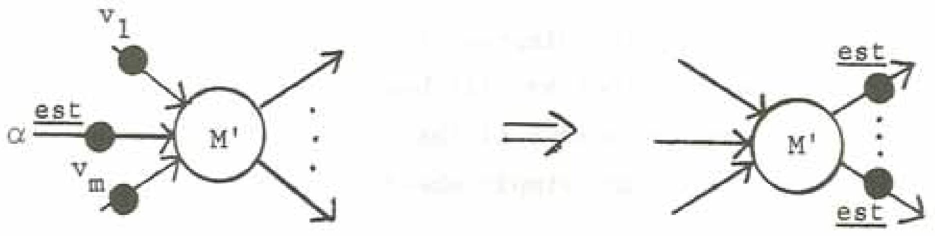
> A system S is properly initialized if and only if there exists at least one initialization (represented by the presence of a con-s operator) on each directed cycle in S.

Examples are shown in figure 4.11, where we have adopted an abbreviation

(i) when no est token has ever been presented at inputs:

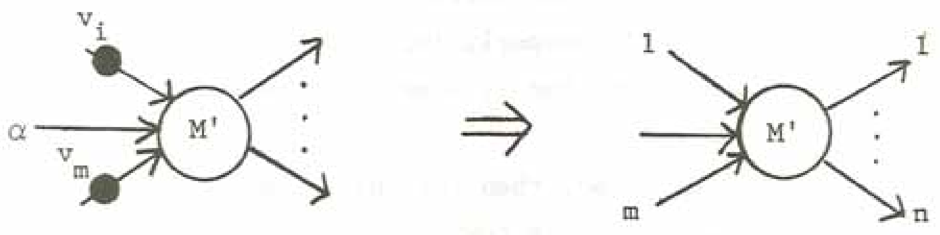


(ii) when for the first time an est token is present at some input:



Some other v_i's may be est tokens, for example the arc α has an est token.

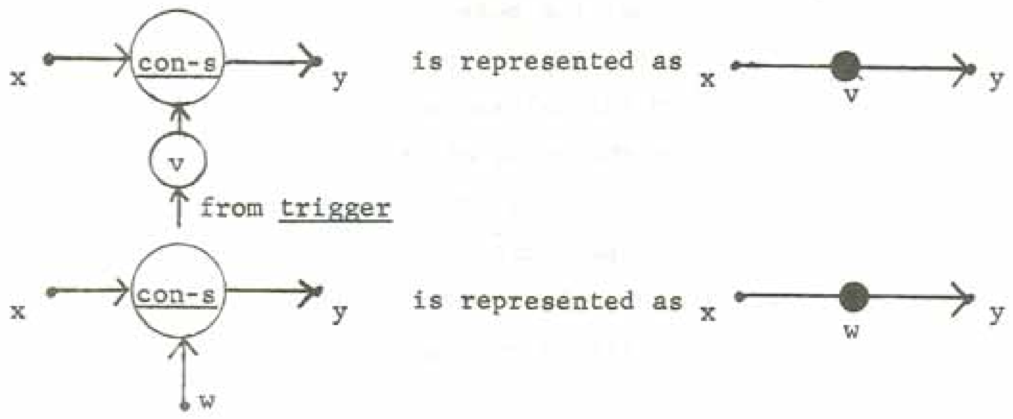
(iii) after (ii) has occurred:



Some arc should not hold any tokens if an est token has being received such as α.

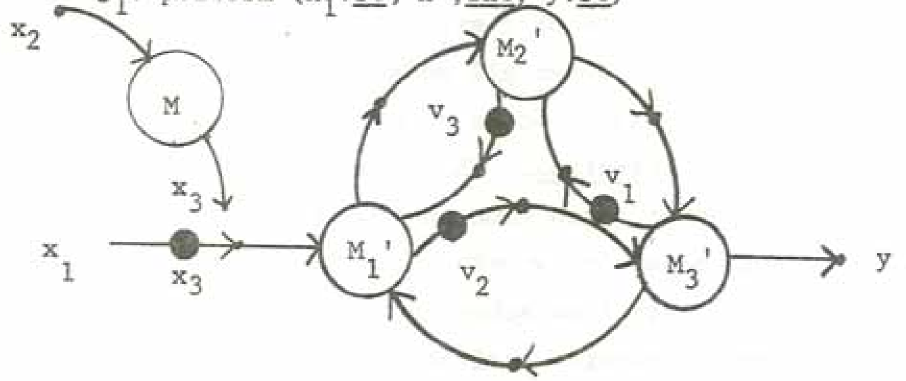
Figure 4.10 The behavior of an extention of M

(i) abbreviation



(ii) an improperly initialized system

S_1: perform (x_1:st, x ;int; y:st)



The system has the outermost cycle not being initialized.

(ii) a properly initialized system

S_2: perform (x_1:st, x_2:int; y:st)

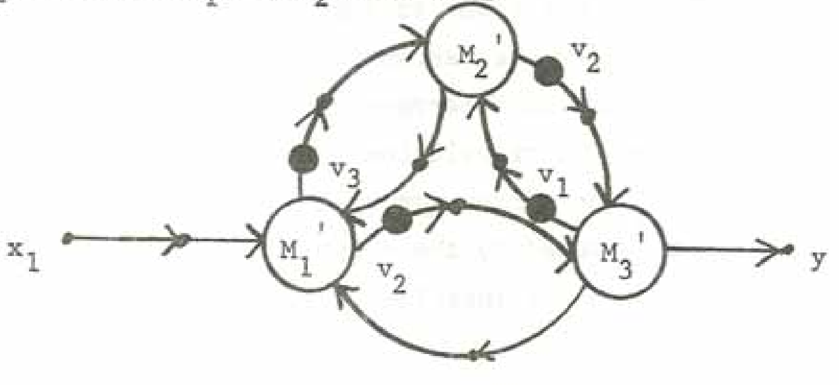


Figure 4.11 Examples for initializations.

for representing initializations as shown in (i). The system S_1 is not properly initialized because the outermost cycle does not contain any initialization. The system S_2 is properly initialized and there are two cycles containing two initializations.

In section 4.3.2 we give the rules of translation for a system of extended well-behaved modules. The translation rules are defined such that the condition for proper initialization can be checked and the resulting recursive data flow schema is always properly terminating even when the translated system is not.

4.3.2 Translation into Recursive Data Flow Schema

The translation rules given here use the property of marked graphs that if the initialization of a graph is live the initial configuration is reestablished after all nodes (corresponding to well-behaved modules) fire once. The process of reestablishing the initial configuration is called an iteration. Therefore, in a system of extended well-behaved modules, each time an iteration is completed a token would have been absorbed by the module to which an input link is connected. The recursive data flow schema resulted from the translation actually performs in each recursive activation the computation required for each iteration. Because some of the modules may receive end of stream tokens, part of the system may not be performing any computation. It is necessary that the recursive schema properly determines which part of the computation needs not be performed.

We define the following notations to facilitate the definition of the rules of translation, under the assumption that we are translating the data flow schema specified by a perform_group which is named S. (This assumption simplifies the translation rule since only identifiers appear in the interface of the perform_group and thus circumvent the undesirable complexity introduced by the existence of constants or other kinds of expressions in the interface.)

Def.	Ist(S)	denotes the set of input links which are inputs of S and are of type stream.
	$I_{simp}(S)$	denotes the set of input links which are of type simple.
	Ost(S)	denotes the set of output links which are of type stream.
	$O_{simp}(S)$	denotes the set of output links which are of type simple.
	L(S)	denotes the set of link actors whose input arcs are initialized by the presence of tokens. (Note that an initialization may be made by an input; in such cases the identifier of the input is written next to the token [see fig. 4.11].)
	V(S)	denotes the set of links designated by identifiers which appear as labels for the tokens indicating initializations. (Refer to figure 4.11 (i) and (ii).)

Translation Rules

The working of the translation assumes the existence of a graphical representation for the schema S and some effective way of manipulating representations of graphs. We believe that this assumption results in easily understandable translation rules. We shall outline the translation rules below, then each step is expanded in detail later.

Step(i)

In this step we obtain two schemas E and F from the schema S.

The schema F corresponds to the part of S which is specified by statements involving only simple values; therefore, it is acyclic if the semantic rules are observed. The schema E corresponds to the part of S which is specified by statements involving stream values. The schema E is obtained by breaking up the cycles in S at link nodes in the set L(S) and should not have any cycles if the system S is properly initialized. Intuitively, the schema E contains acyclic data flow schemas which perform the computation required to complete an iteration.

Step (ii)

We embed the subschemas of E in conditional schemas. The purpose of this embedding is to avoid performing the computation which is not necessary owing to the behavior of S after some extended well-behaved modules have received est tokens. We call the resultant schema E'.

Step (iii)

We use the schema E' to construct a recursive data flow schema R expressed in TDFL by allowing est tokens and the predicate eos. This recursive schema simulates the computation of the part of S consisting of extended well-behaved modules. The recursive activation of R stops when all outputs are terminated by est tokens (i.e., R is properly terminating).

Step (iv)

The desired schema T consists of a module_call of the schema F and a module_call of the schema R.

We now present the translation rules and simple examples are used for illustration.

(i) The schema S can be partitioned into two subschemas S_1 and S_2 connected as shown in figure 4.12. The schema S_1 corresponds to the part of S which is specified by statements involving only simple values; that is, it contains all links associated with simple type identifiers and all actors defined over simple values including constant value actors. The schema S_2 corresponds to the part of S which is specified by statements involving stream values and therefore includes all extended modules and the con-s operators. The schema F is simply the schema S_1 with the modules replaced by the corresponding module_application actors, and with the addition of a new set of links W(S) as shown in figure 4.13. (Note that links   are also provided as output links of constant value actors.) The schema E is obtained from S_2 with the following rules:

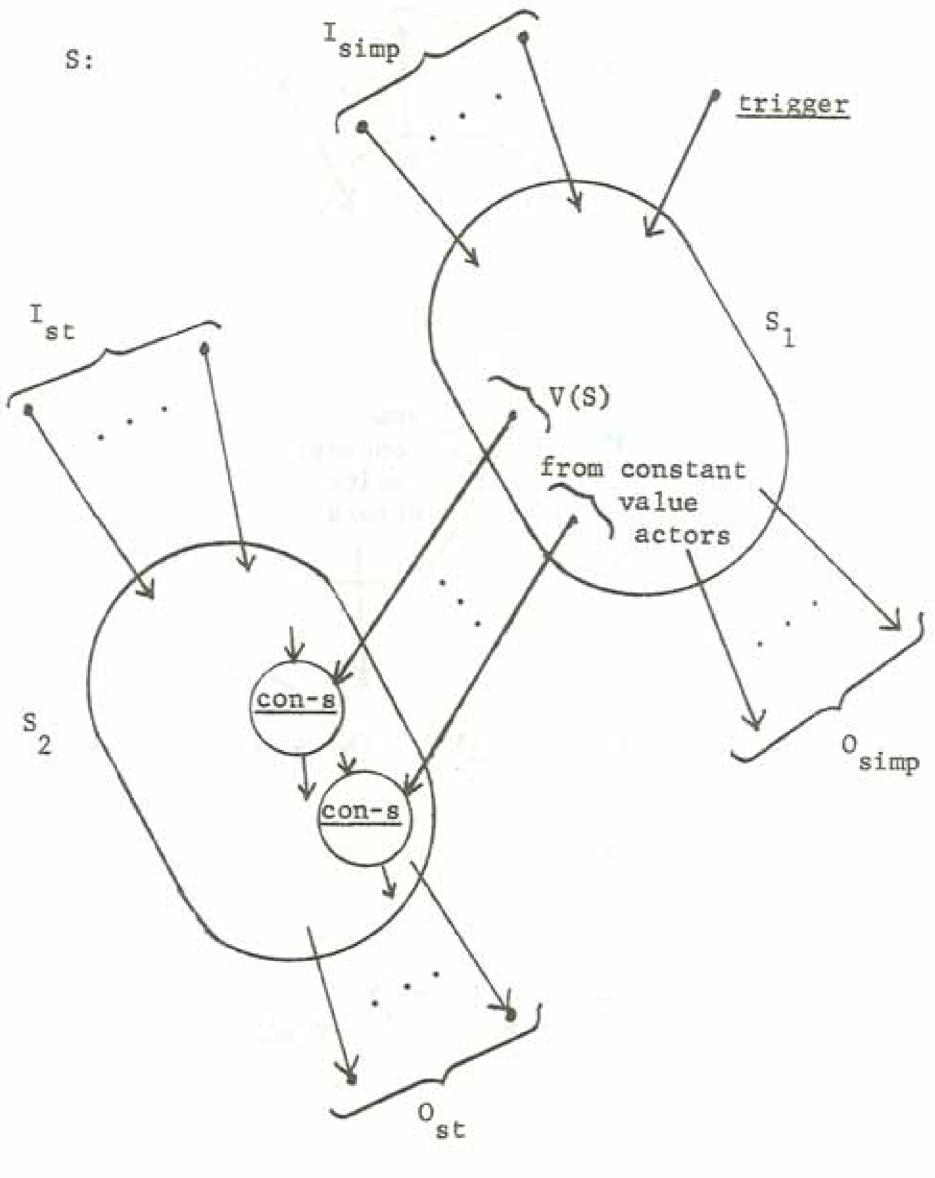


Figure 4.12 Partition of S

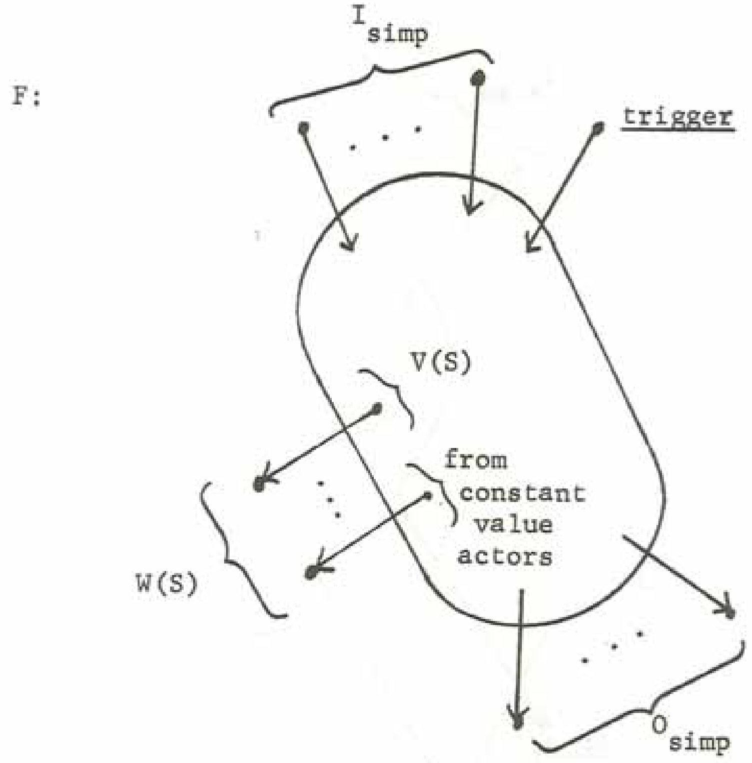


Figure 4.13 The schema F

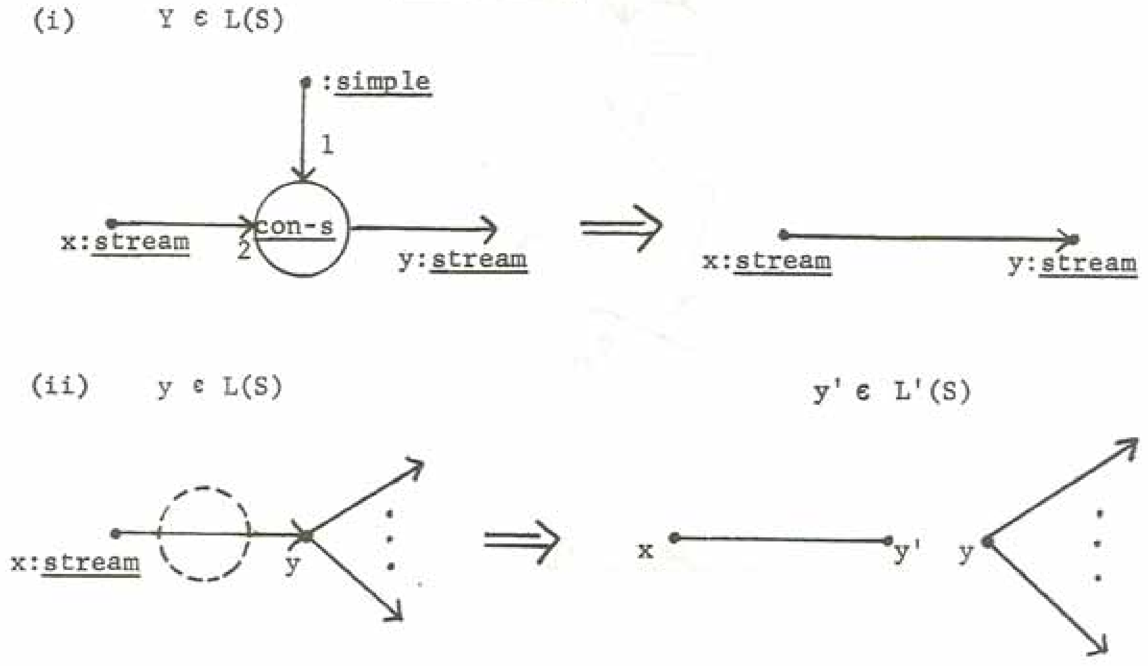


Figure 4.14 Splitting

a. Replace all <u>con-s</u> operators by arcs as shown in figure 4.14 (i).

b. Split each link y in L(S) as exemplified in figure 4.14 (ii) by creating a new link $y' \in L'(S)$ and reconnecting the input arc of y to y'.

The effect of the splitting of links is to break the cycles of the schema S_2 at the points where initializations are defined.

c. Define input and output links of the schema E by

$$\text{In} = I_{st}(S) \cup L(S)$$

and $\text{Out} = O_{st}(S) \cup L'(S)$. (Refer to figure 4.15.)

The example illustrating this step is shown in figure 4.16.

(ii) There are three substeps a, b and c. Let $o \in \text{Out}$, then by Dep(o) we mean the set of input links in In which have a directed path leading to the link o in the schema E.

a. For each $o \in \text{Out}$, we create a subschema g(o) of E which contains all intermediate links and actors on the paths between links in Dep(o) and the link o; the arcs which emanate from these actors but do not lie on the paths are terminated by sink actors (see figure 4.17(a)). Note that this process in fact may duplicate actors as is the case in the figure 4.18(a). This is because of our intention to make the translation rule as simple as possible.

b. The subschema g(o) is embedded in a conditional schema, as shown in figure 4.17(b), after the links $x_i \in \text{Dep}(o)$ and o are renamed to be x_i' and o', respectively. We shall call the resultant conditional schema C(o). The schema P contains tests for end of stream tokens and boolean operators such that the output value is <u>true</u> if none of the input tokens to each input link $x_i \in \text{dep}(o)$ is an <u>est</u> token and is <u>false</u> otherwise. The conditional schema, therefore, yields the output computed by g(o) if the output of P is <u>true</u>; otherwise it yields an <u>est</u> token. Informally, the conditional schema may "propagate" the <u>est</u> token (tokens) and thereby simulate the behavior of S when some <u>est</u> tokens are received by some modules. Examples are shown in figure 4.18(b).

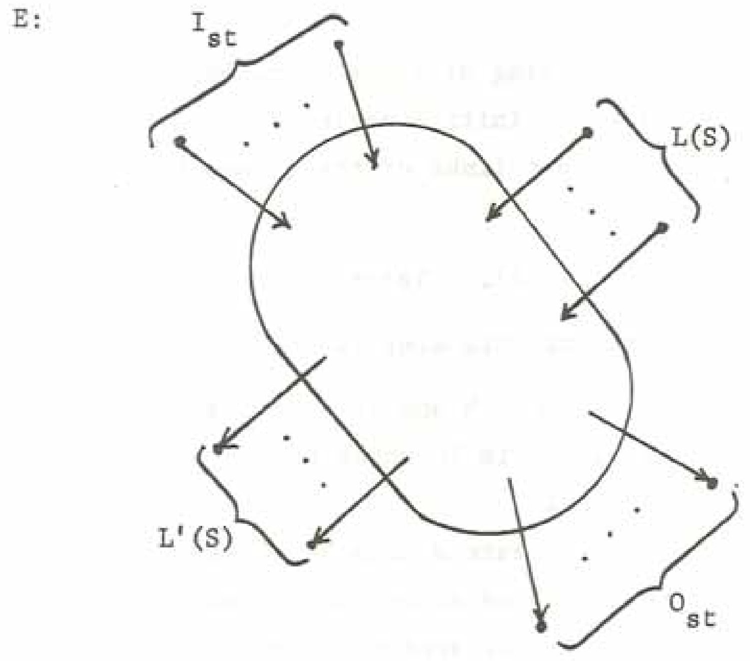


Figure 4.15 The schema E

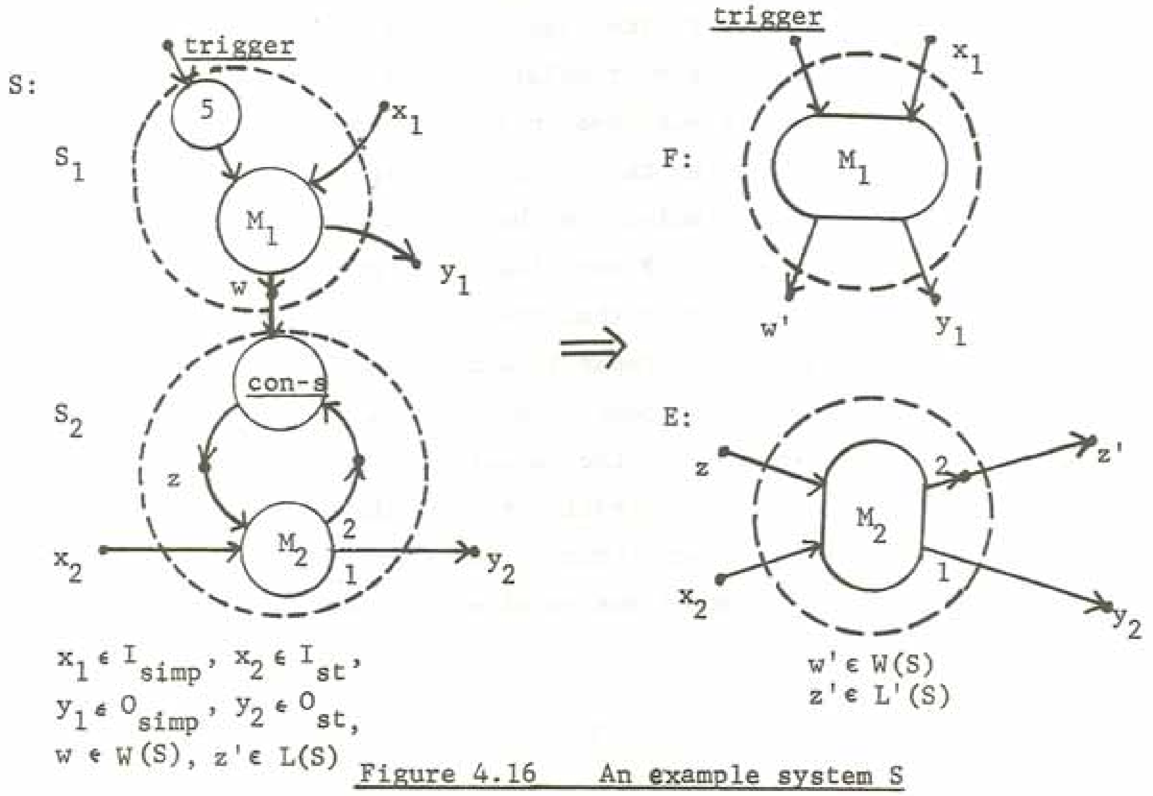
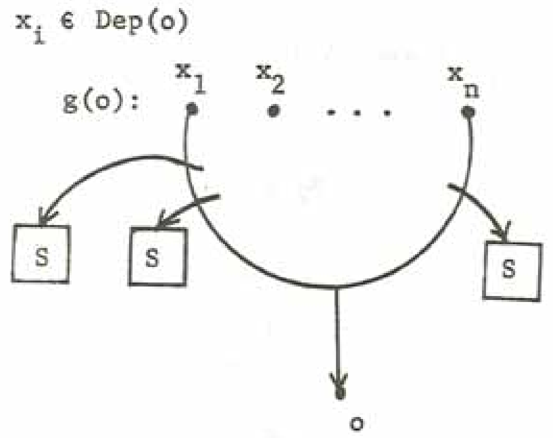
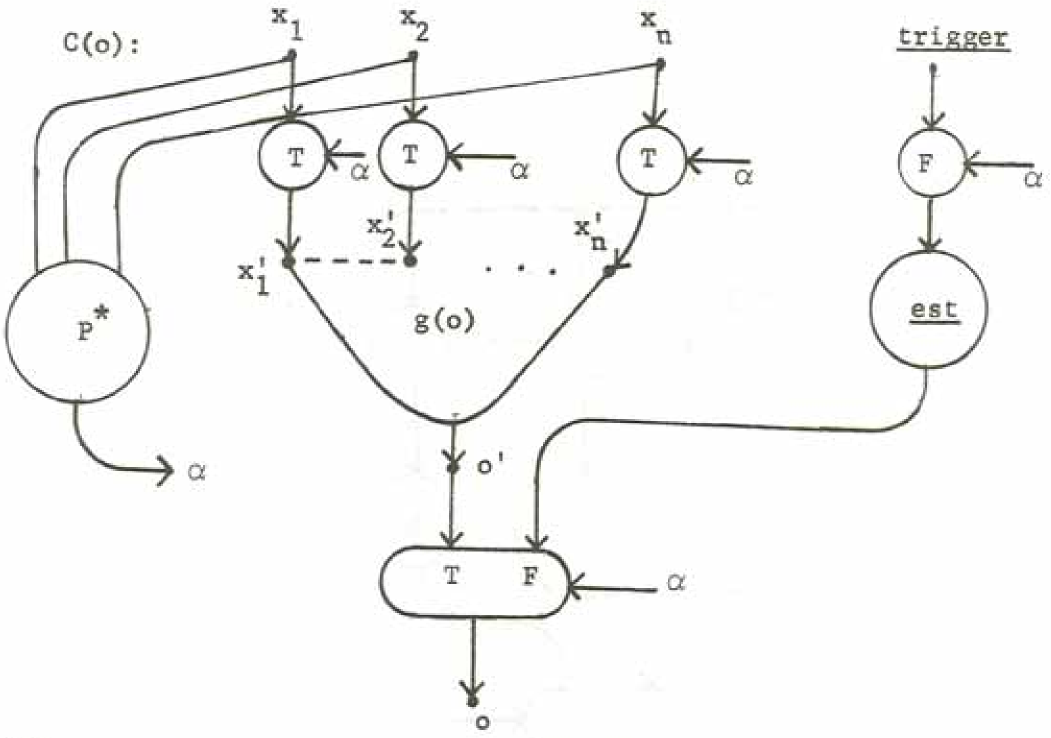


Figure 4.16 An example system S

(a) a subschema g(o)



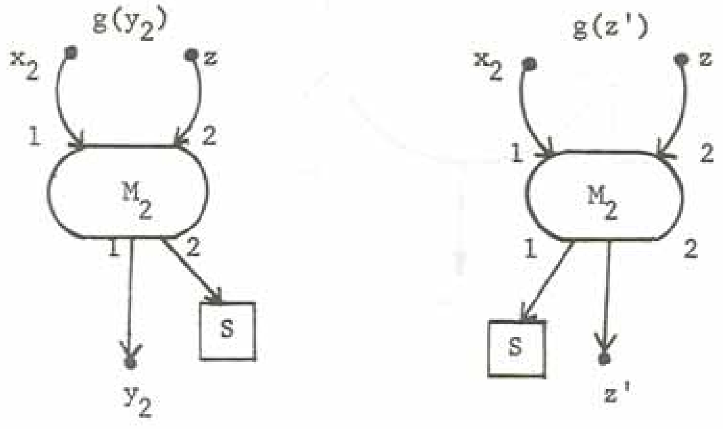
(b) embedded subschema



(*) P produces a true value if none of x_i's is the est token, otherwise a false value is produced.

Figure 4.17 Embedding of a subschema g(o)

(a) using the example shown in figuer 4.16
In= $\{x_2, z\}$, Out= $\{y_2, z'\}$



(b) embedding of $g(y_2)$ and $g(z')$

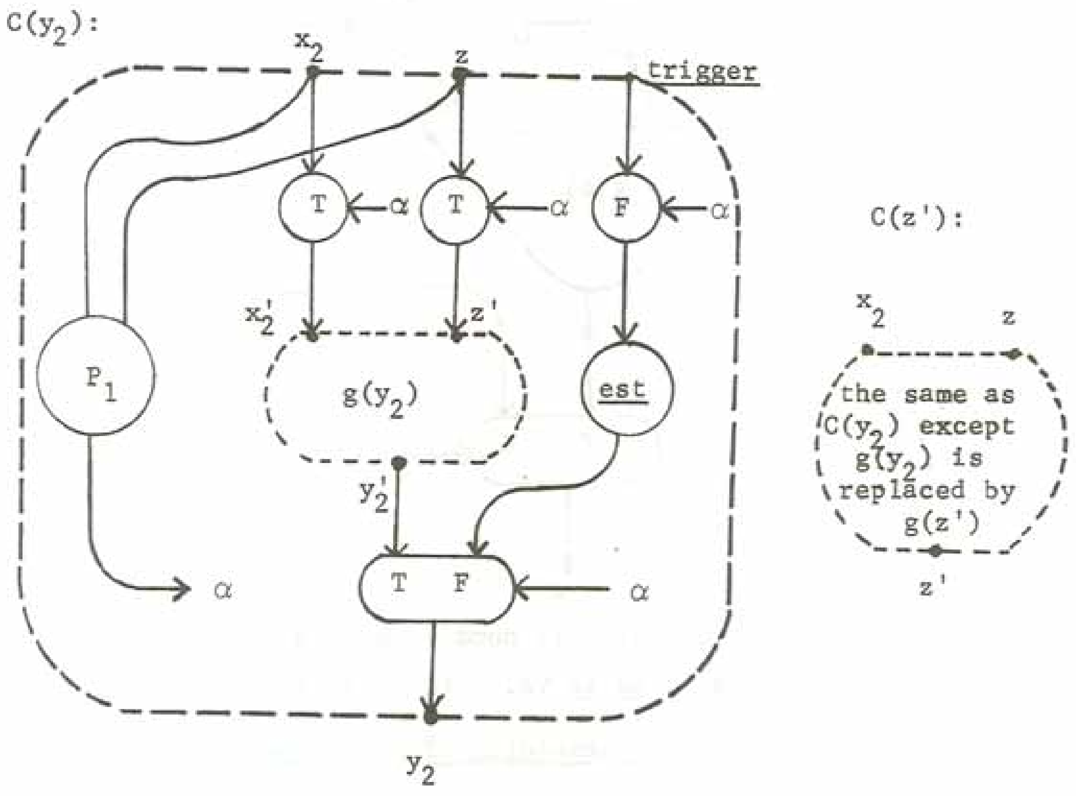


Figure 4.18 Examples for step (ii)

(c)

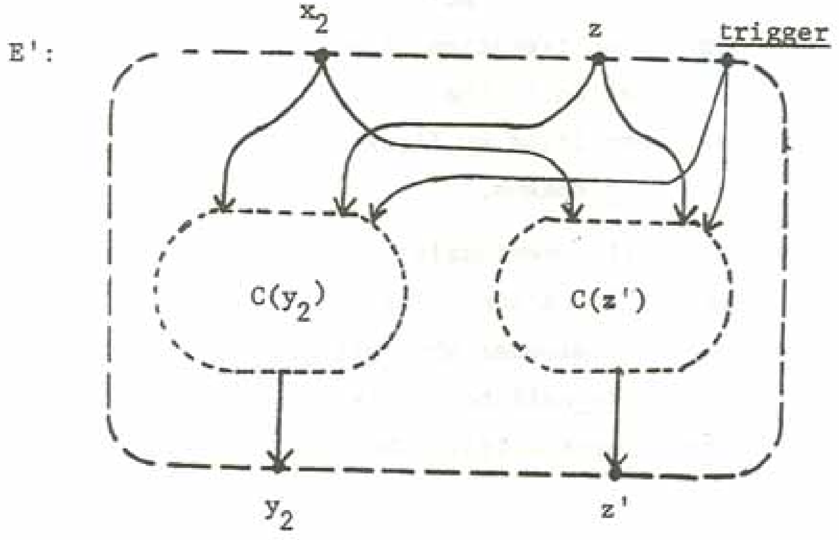


Figure 4.18 (continued)

c. The schema E' merges all input links of C(o) for $o \in$ Out, which has the same identifiers as labels. The sets of input and output links of E' are simply In and Out, respectively. This step is demonstrated by figure 4.18(c). The reader should note that the computation carried out by E' actually performs the computation required to complete an iteration (i.e., the process of reestablishing initial configuration in the sense of initial markings of marked graphs as exemplified by figure 4.11) in S, if the modules do not receive any est tokens.

(iii) As described informally previously, the schema E' performs the computation required in one iteration. The schema R simply repeates recursively iterations and terminates when all outputs are completed (i.e., when the outputs of the module_call to E' yields $o_1, \ldots, o_k$ = est) as tested by the conditional A (see figure 4.19). The recursive schema R is defined as follows:

$$\begin{aligned} \text{Let } & i_1, \ldots, i_m \in \text{Ist}(S) \\ & o_1, \ldots, o_k \in \text{Ost}(S) \\ & \ell_1, \ldots, \ell_n \in L(S) \\ & \ell'_1, \ldots, \ell'_n \in L'(S), \end{aligned}$$

then we define a module E' from the schema E':

E' : module ($i_1, \ldots, i_m, \ell_1, \ldots, \ell_n; o_1, \ldots, o_k, \ell'_1, \ldots, \ell'_n$)
< body > specifying the schema E'
mend.

The module for the schema R is expressed in TDFL by allowing est as a constant value and eos as a predicate, as shown in figure 4.19. Note that the function of the group of statements in B is to "absorb" one more token from each input in Ist, and that the function of the group C is to put one more token on each output in Ost whenever one iteration is completed.

```
R : rmodule (i_1,...,i_m, ℓ_1,...,ℓ_n; o_1,...,o_k, ℓ'_1,...,ℓ'_n)

      ⎧ if empty (i_1) then est → I_1;  [ ] → r_I_1
      ⎪                else get (i_1) → I_1, r_I_1
    B ⎨                end;
      ⎪        :
      ⎪  (similarly for i_2,...,i_m)
      ⎩        :

     E'(I_1,...,I_m, ℓ_1,...,ℓ_n; O_1,...,O_k, L_1,...,L_n);

      ⎧ if  eos(O_1) ∧...∧ eos(O_k)
      ⎪     then   [ ] → o_1,...,o_k;
      ⎪            est → ℓ'_1,...,ℓ'_n
      ⎪     else
    A ⎨         R(r_I_1,..., r_I_m, L_1,...,L_n;
      ⎪              r_o_1,..., r_o_k, ℓ'_1,...,ℓ'_n);
      ⎪
      ⎪        ⎧if eos(O_1) then r_o_1 → o_1
      ⎪      C ⎨            else con-s(O_1, r_o_1) → o_1
      ⎪        ⎪            end
      ⎪        ⎪     :
      ⎪        ⎩  (similarly for O_2,...,O_k)
      ⎩     end          :

mend
```

Figure 4.19   The schema R

(iv) Lastly,

let $j_1,\dots,j_a \in$ Isimp, $p_1,\dots,p_b \in$ Osimp, and $w_1,\dots,w_t \in W(S)$,

then the module F is defined as follows:

F : <u>module</u> ($j_1,\dots,j_a$; $p_1,\dots,p_b$, $w_1,\dots,w_t$)
&lt; body &gt; specifying the schema F
<u>mend</u>.

The module T below is the result of the translation of S.

```
T : module (list of input id's; list of output id's)
    F(j_1,...,j_a; p_1,...,p_b, w_1,...,w_t):
    R(i_1,...,i_m, w_1,...,w_t;
            o_1,...,o_k, ℓ'_1,...,ℓ'_n)
    mend
```

The module T performs the computation corresponding to that of S_1 first and the outputs $w_1,\dots,w_t$ are used as initial values to S_2.

4.4 <u>Extensions</u>

As described in section 4.2, a perform_group may be named as a module. We did not, however, allow recursive perform_groups. This extension provides more generality, and may be embedded in the language. We have restricted the kind of assignments allowable in a perform_group. Some extensions such as the inclusion of <u>first</u>, <u>get</u> and stream valued constants can be useful. It is an open issue whether conditionals should be allowed in a perform_group.

Chapter 5

Summary and Conclusion

In this thesis we presented a parallel programming language which is inherently determinate. The semantics of the language is given by providing rules for translation into recursive data flow schemas. The language incorporates important features which contribute to the semantic simplicity of the language: single assignment, explicit declaration of inputs and outputs of a module, stream-oriented computation, and constructs for defining a system of inter-communicating modules. In section 5.1, we shall discuss some additional issues related to the above features. We have avoided these discussions thus far in order to prevent readers from being side-tracked and with the hope that we can provide a more complete and coherent point of view. Section 5.2 points out several issues on data flow schemas and their implementation. Extensions of the language and areas for further research are suggested in section 5.3.

5.1 Summary and discussion

In chapter 2 we introduced data flow schemas and the basic structure of TDFL. The class of rwf data flow schemas excludes iteration schemas as a result of several considerations. First, the semantics of iteration schemas involves the update of an identifier and therefore does not conform with single assignment. Second, we feel that efficiency arguments against recursion are not justified in general and that the recursive form of describing iterative processes in an asynchronous system may result in faster completion of the computation. Third, in the computer architecture proposed by Dennis, Misunas [14] iteration schemas need to be modified by adding gates to prevent the arbitration network from possibly hanging up (Misunas [35]). The elimination of non-local identifiers results in simple translation rules for TDFL and avoids the question of whether non-local identifiers should have static or dynamic binding.

The explicit distinction of inputs and outputs of a module is a natural consequence of the single assignment rule and circumvents the semantic complexity introduced by defining parameter passing conventions.

We may relax the requirement that all identifiers be well-defined by allowing references to identifiers to precede their definitions.

Chapter 3 introduced new primitives which provide for computations over streams. By eliminating end of stream tokens (est) and the predicate end of stream (eos) from the language TDFL one can guarantee that ill-formed sequences of tokens will not occur.

In chapter 4 we summarized some of the results in the theory of parallel computation on determinate systems. The behavior of interconnected data flow modules is shown to be properly abstracted in terms of lattice theoretical functions, which suggests the possibility of unifying the semantics of the language within the framework of the Scott-Strachey [40] mathematical approach. (As a side issue we must point out that in forming the lattice of the partial ordering for streams one mus include the est token as part of the signal history.)

The concepts of proper initialization and termination were discussed. We believe that any program should either be expressed in the subset of the language guaranteeing that these properties exist or else are specifically proved to satisfy these properties. We defined a subclass of determinate systems composed of extended well-behaved modules and gave necessary and sufficient conditions for determination of proper initialization. Translation into properly terminating recursive data flow schemas is also given for the subclass.

5.2 Related Issues on Data Flow Schemas

The firing rule of the module_application actor requires a token be present on each input arc for the actor to be enabled. This requirement actually makes a computation less asynchronous since all inputs are guaranteed to arrive if there is no non-terminating computation which does not generate any outputs. The feasibility of relaxing the firing rule for the module_application actor should be considered seriously, expecially if it should result in higher degree of parallelism than what would be otherwise. As a result the semantics of module_call in a perform group can be greatly simplified.

Consider, for example, the program shown in figure 5.1. The activation of the module generates a snapshot as shown in figure 5.2, where only get and con-s operators are shown. The long chains of get and con-s operators cause substantial inefficiency if the implementation for data flow schemas actually generates these chains, since tokens must travel through the chains whose lengths keep on increasing as new activations are invoked. This need not be the case if the implementation simulates the effect of these chains. For instance, arrays may be employed or some mechanism for dynamically shortening the chains could be devised.

```
Chopper : module (x : st int ; y : st int)
          if empty (x) then [ ] → y
                       else get (x) → x_head, x_rest;
                            if x_head < 0 then [ ] → y
                                          else chopper (x_rest) → y_rest;
                                               con-s (x_head, y_rest) → y
                                          end
                       end
            mend
```

Figure 5.1

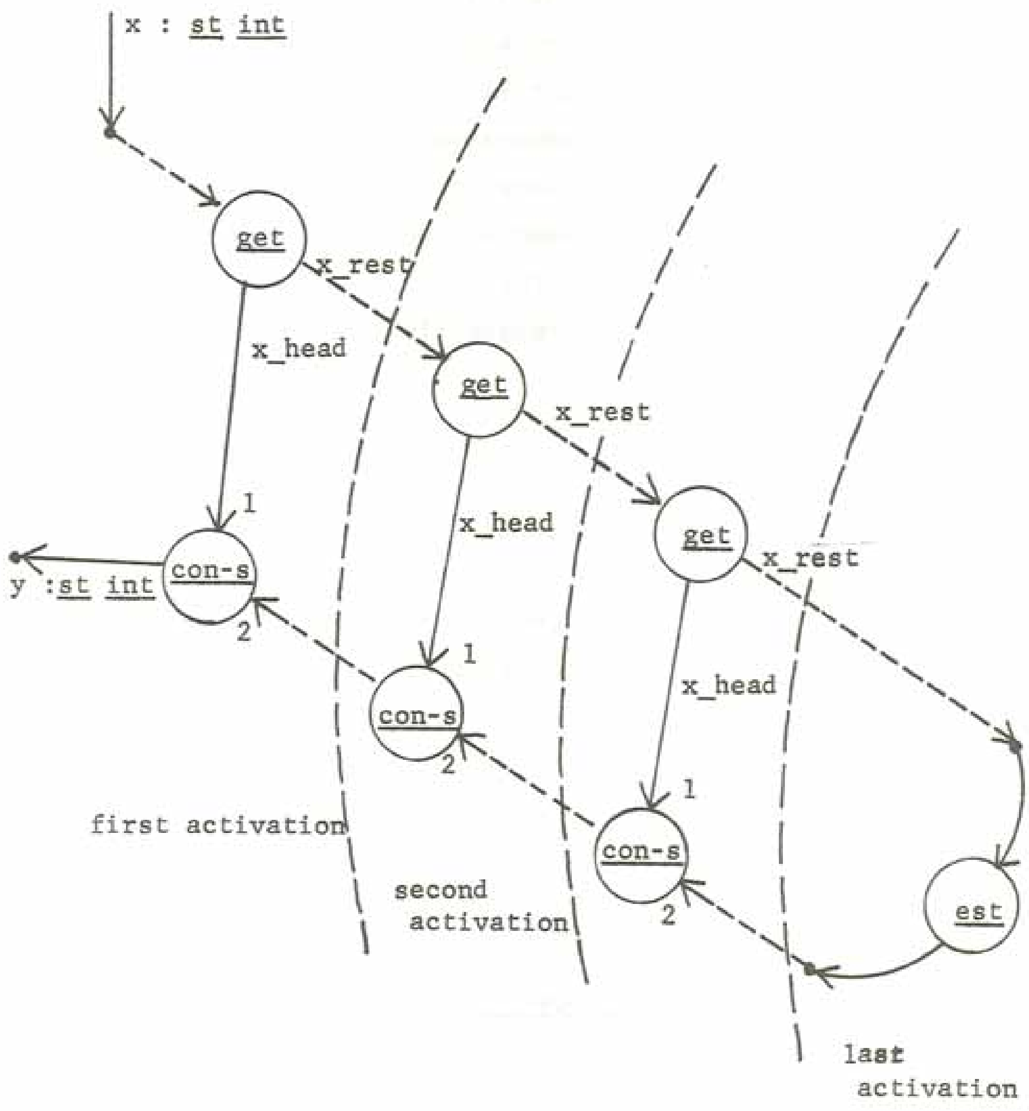


Figure 5.2 Chains of stream operators

5.3 Further Work and Research

The language TDFL is minimal in the sense that many desirable features are not included for simplicity and can be added if so chosen. For example, new data type such as string and data structures based on the acyclic structure of CBL (Dennis [10]) may be included, and common occurences of nested conditionals may be reduced by case statements.

Computations involving data structures often exhibit high degree of parallelism. An example is the simultaneous activation of processes operating on each component of a data structure. A for_all construct may be defined for expressing this kind of parallelism. Primitives for converting a data structure to a stream comprised of substructures and vice versa may also prove useful.

It is still an open area of research to define a set of primitives for non-determinate computations. When a data structure is shared the "Monitor" concept advocated by Hoare [21] may prove adequate.

Techniques for optimization and transformations which are applicable to data flow schemas to gain efficiency and more parallelism still need to be investigated.

BIBLIOGRAPHY

1. Adams, D.A. A Computation Model with Data Flow Sequencing. Technical Report CS 117, Computer Science Dept., School of Humanities and Sciences, Stanford Univ., Stanford, Calif., Dec. 1968.

2. Anderson, D.N., F.J. Sparacio, and R.M. Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction Handling. IBM Journal of Research and Development, 11, 1 (Jan. 1967), 8-24.

3. Anderson, J.P. Program Structures for Parallel Programming. Comm. ACM, 8 (Dec. 1965), 786-788.

4. Bährs, A. Operation Patterns (An extensible model of an extensible language). Symp. on Theoretical Programming, Novosibirsk, U.S.S.R., August 1972 (preprint).

5. Balzer, R.M. Port: A Method for Dynamic Interprogram Communication and Job Control. Proc. AFIPS Conf., 39.

6. Barnes, G.H., et al. The Illiac IV Computer. IEEE Transactions on Computers, C-17, 8 (Aug. 1968), 746-757.

7. Conway, M.E. A Multiprocessor System Design. Proc. FJCC, 23, (1963), 139-146.

8. Dahl, O.-J., et al. SIMULA 67. Norweigian Computing Center, Oslo, Norway.

9. 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, 362-376.

10. Dennis, J.B. On the Design and Specification of a Common Base Language. Proc. of the Symp. on Computers and Automata, Polytechnic Press of the Polytechnic Institute of Brooklyn, 1971, 47-74.

11. Dennis, J.B. Coroutines and Parallel Computation. Proc. Princeton Conference, 1971.

12. Dennis, J.B. and J.B. Fosseen. Introduction to Data Flow Schemas. CSG Memo 81-1, Project MAC, M.I.T., Cambridge, Mass.

13. Dennis, J.B., and D.P. Misunas. A Computer Architecture for Highly Parallel Signal Processing. Proc. of the ACM 1974 National Conference, N.Y., Nov. 1974, 402-409.

14. Dennis, J.B., and D.P. Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. Proc. of the Second Annual Symp. on Computer Architecture, IEEE, N.Y., Jan. 1975, 126-132.

15. Dennis, J.B., and E.C. Van Horn, Programming Semantics for Multiprogramming Computations. Comm. ACM, 9 (Mar. 1966), 143-155.

16. Ellis, D.J. Semantics of Data Structures and References. Report TR-134, Project MAC, M.I.T., Cambridge, Mass., Aug. 1974.

17. Dijkstra, E.W. Notes on Structured Programming. Structured Programming, Academic Press, 1972.

18. Dijkstra, E.W. Co-operating Sequential Processes. Programming Languages, F. Genuys, Ed., Academic Press, 1968.

19. Fosseen, J.B. Representation of Algorithms by Maximally Parallel Schemata. S.M. Thesis, Dept. of E.E., M.I.T., Cambridge, Mass.

20. Hack, M.H.T. Analysis of Production Schemata by Petri Nets. Report TR-94, Project MAC, M.I.T., Cambridge, Mass., 1972.

21. Hoare, C.A.R. Monitors: An Operating System Structuring Concept. CS73-401, Dept. of Computer Science, Stanford Univ., 1973.

22. Hoare, C.A.R. Towards a Theory of Parallel Programming. Operating System Techniques (ed. C.A.R. Hoare and R.N. Perrott), Academic Press, London, 1972.

23. Hoare, C.A.R. Parallel Programming: An Axiomatic Approach. Memo AIM 219, Stanford Artificial Intelligence Lab., Stanford Univ., Calif., 1973.

24. Kahn, G. A Preliminary Theory for Parallel Programs. Internal Memo, Institut de Rech. d'inform. et d'Automat., Rocquencourt, France, 1973.

25. Kahn, G. The Semantics of a Simple Language for Parallel Programming. Proc. IFIP Cong. Conference, 1974.

26. Karp, R.M., and R.E. Miller. Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing. SIAM Journal of Applied Mathematics, 14 (Nov. 1966), 1390-1411.

27. Knuth, D.E. Fundamental Algorithms. The Art of Programming, Vol. 1, Addison-Wesley.

28. Knuth, D.E. Structured Programming with GOTO Statements. Computing Surveys of the ACM, 6, No. 4, Dec. 1974.

29. Kogge, R.M., and H.S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. STAN-CS-72-298, Dept. of Computer Science, Stanford Univ., Calif., 1972.

30. Kosinski, P.R. A Data Flow Language for Operating Systems Programming. Proc. of the ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices, 8, 9 (Sept. 1973), 89-94.

31. Krieg, B. A Class of Recursive Coroutines. Proc. IFIP Cong. Conference, 1974.

32. Lamport, L. The Parallel Execution of Do Loops. Comm. of the ACM, 17, No. 2, Feb. 1974.

33. Lauesen, S. A Large Semaphore Based Operating System. Comm. of the ACM, 18, No. 7 July 1975.

34. Levitt, K.N. The Application of Program Proving Techniques to the Verification of Synchronization Processes. Proc. FJCC, 1972.

35. Misunas, D.P. A Computer Architecture for Data Flow Processor. S.M. Thesis, Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., 1974.

36. Patil, S.S. Closure Property of Interconnected Systems. Record of the Project MAC Conference of Concurrent Systems and Parallel Computation, ACM, N.Y., 1970.

37. Ramamoothy, C.V., and M.J. Gonzalez. A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs. Proc. FJCC, 1969, 1-15.

38. Rodriguez, J.E. A Graph Model for Parallel Computation. Report TR-64, Project MAC, M.I.T., Cambridge, Mass., Sept. 1969.

39. Rumbaugh, J.E. A Parallel Asynchronous Computer Architecture for Data Flow Programs. Report TR-150, Project MAC, M.I.T., Cambridge, Mass., 1975.

40. Scott, D., and C. Strachey. Toward a Mathematical Semantics for Computer Languages. Proc. of the Symp. on Computers and Automata, Polytechnic Institute of Brooklyn, Apr. 1971, 19-46.

41. Strachey, C., and C.P. Wadsworth. Continuations -- A Mathematical Semantics for Handling Full Jumps. Technical Monograph RPG-11, Oxford Univ. Computing Lab., Programming Research Group, Jan. 1974.

42. Tesler, L.G. and H.J. Enea. A Language for Concurrent Processes. Proc. SJCC, 1968.

43. Thornton, J.E. Parallel Operations in Control Data 6600. AFIPS Conference Proc., 26, Part II, AFIPS Press, Montvale, N.J., 1964, 33-41.

44. Tomasulo, R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. IBM Journal of Research and Development, 11, 1 (Jan. 1967), 25-33.

45. Traub, J.F., (ed.). Complexity of Sequential and Parallel Numerical Algorithms. Academic Press, N.Y., 1973.

46. Vuillemin, J.F. Proof Techniques for Recursive Programs. Memo AIM-218, Stanford Artificial Intelligence Lab., Stanford Univ., Calif., 1973.

47. Zahn, C.T. A Control Statement for Natural Top Down Programming. Presented at Symp. on Programming Languages, Paris, 1974.