A Formal Model of Non-determinate Dataflow Computation

by

Jarvis Dean Brock

A. B., Duke University
1974

S. M., E. E., Massachusetts Institute of Technology
1979

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

August, 1983

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science
August 23, 1983

Certified by [illegible]
Jack B. Dennis
Thesis Supervisor

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

A Formal Model of Non-determinate Dataflow Computation

by

Jarvis Dean Brock

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 1983 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

Almost ten years ago, Gilles Kahn used the fixed point theory of Dana Scott to define a formal and elegant model of computation for *determinate* dataflow graphs, networks of determinate processes communicating asynchronously through unbounded channels. Kahn viewed each process as a function mapping each tuple of *streams*, or sequences of values, received through its input channels to the tuple of streams produced at its output channels. Determinacy was defined as the requirement that the mapping be functional — that for each input stream tuple there be only one possible output stream tuple. Although most useful computation can be accomplished with only determinate processes, there are many important, inherently non-determinate application areas to which Kahn's theory cannot be applied.

In this thesis, a formal model of computation for non-determinate networks is presented in which each possible computation of a network is represented by a *scenario*. A scenario is a pair consisting of an input stream tuple and an output stream tuple, together with a *causality order* relating each element of the input and output streams to those elements which played a role in its creation. A non-determinate network is represented by a set of scenarios, just as a determinate network is represented by a set of pairs of input and output stream tuples. Scenario sets contain but a little more information than the most straightforward extension of Kahn's theory, the representation of graphs as relations on tuples of streams. We justify the addition of this causality information by demonstrating that the relational representation is inadequately detailed for describing non-determinate computation.

A formal algebra for deriving the scenario set of a graph from the scenario sets of its components is also presented. Our scenario set composition rules are very simple. Only an elementary knowledge of partial orders is required in order to understand them. We prove the correctness of our model by showing its consistency with the standard operational model of dataflow computation.

Thesis Supervisor: Jack B. Dennis
Title: Professor of Computer Science and Engineering

Keywords: Programming language semantics, concurrent computation, non-determinate computation, data flow.

Acknowledgments

I'd like to thank my thesis supervisor, Jack Dennis, for encouraging my efforts in both dataflow computation and formal semantics. I have learned much during my long association with him.

My second thesis reader, Arvind, has been generous with his time and interest in my work. We have had many long discussions covering all aspects of dataflow and applicative languages.

For me, it was very fortunate that Joe Stoy came from Oxford to spend the summer at MIT. In spite of the little warning, he agreed to be the third thesis reader. In addition to battling the then (and perhaps now) abominable grammar, he provided many thought-provoking comments about the relation of this work to similar ones.

At least three other people contributed directly to this thesis. Bill Ackerman posed the initial challenge that lead to this research and worked with me in the development of many of its concepts. Gordon Plotkin suggested the simplification of the scenario composition rules presented in Chapter 5. Joan Curry carefully read drafts of this thesis and corrected numerous errors.

I've enjoyed the company of the many friends I have made among the graduate students in Jack Dennis' Computation Structures Group and Arvind's Functional Languages and Architectures Group. Two have been especially helpful throughout the last few weeks of this effort. Andy Boughton is a CSG old-timer who has never failed to provide help to a friend. Keshav Pingali is an FLA old-timer with whom I have spent many hours evolving solutions to the problems of dataflow computation and to the crosswords of the *New York Times*.

And finally, I would like to thank Ruth Brock for sharing the adventure, and I wish her luck as she now pursues her own education at the Technical College of Alamance.

This research was supported by the National Science Foundation under grant number MCS-7915255.

Table of Contents

1. Introduction

Dataflow is a model of concurrent computation whose program schema is a graph, or network, of concurrent operators. Although the dataflow graph is most often envisioned microscopically as the representation of a dataflow machine language program, the macroscopic view of the graph as a collection of communicating agents distributed over several machines is equally valid. Viewed thus, dataflow graphs may represent the "large" problems, such as transaction systems, that generate so much interest at a time when computation has moved from the computer room to the teller's desk.

However, as the locus of computation moves in this direction, many of its characteristics change. Formerly most computation was determinate; that is, for any given set of input data, only one possible output result was possible (or, at least, desired). But now the nature of the input data has changed: input was once a deck of cards and, perhaps, a few disk files, but today it often consists of characters typed from widely-dispersed terminals and records transmitted from remote computers, and the ultimate output may depend on the relative timing of these input activities. One consequence of the dependence of output on relationships of time is non-determinacy: performing a computation on a set of input data may result in any of many possible sets of output data.

Presently, non-determinate computation is undoubtedly the most significant area of research in programming language semantics, the study and development of formal models to describe the behavior of programs. Although Scott's [39] fixed point semantics provided a basis from which all aspects of determinate computation could be described, the proposed extensions of this work to non-determinate computation have generally suffered from one of the four following flaws: (1), they are unable to cope with the most important type of non-determinism, that resulting from "races" of independent concurrent processes; (2), they are so *operational*, or permeated with the particulars of implementation, that the representation of a program is almost as detailed as its syntax; (3), they are too *abstract* and fail to distinguish some computations with observably different behaviors; or (4), they can be understood only by mastering very difficult mathematical constructions, such as the derivation of co-limits within categories.

The thesis presents a new model of non-determinate computation which attempts to avoid the four faults enumerated in the preceding paragraph. The model is specialized to dataflow, a scheme for highly concurrent execution. The model is both abstract and simple. The purest input-output characterization of a parallel program or system with input and output channels would be a relation on the histories (sequences) of values produced and received through those channels. Unfortunately, such history relations do not contain sufficient information to be used as the basis for a theory of non-determinate computation. The "merge anomaly," our demonstration of this inadequacy, is accomplished by constructing two systems which have the same history relation but which cannot replace each other within a larger system without altering that system's history relation.

In our semantic model systems are represented by scenario sets.[1] A scenario is an easy extension of an input history-output history pair made by placing a partial order on the individual elements of the sequences of communicated values. This partial order embodies causality by relating each communicated value to the other values which contributed to its existence or, equivalently, which must precede it in time. The behavior of a system is the collection of all its possible scenarios.

The scenario set of a system may be derived from those of its components. In comparison to the complexity of many other non-determinate semantic models [10, 25], the scenario set

1. Scenarios were first described in a paper by this author and Ackerman [8].

composition rules are very simple, requiring only an elementary understanding of properties of partial orders as prerequisite mathematical knowledge. One other advantage of our model over these others is its ability to represent *fair* computation. The natural and straightforward relation of the scenario model to implementation-level models of dataflow computation allows us formally to prove, by induction over the structure of dataflow programs, that scenarios are a correct model of parallel computation.

The underlying programming language of our theory is the dataflow graph. The graph is a collection of concurrently executing operators, or processes. Each of the processes has a fixed, finite set of input and output ports. The links of the graph connect the ports of the processes. Each link is one-way and one-to-one: one output port is joined to one input port. Within the graph, the only means of inter-process communication is the transmission of values through these connections. We assume the connections to be FIFO queues with unbounded buffering capacity so that no two values are ever transposed and no value is ever lost in transit through the link.

It may appear that, in adopting these interconnection rules, we have prohibited three of the most popular forms of interprocess communication: the memory of conventional multi-programming languages in which processes deposit and examine shared data; the synchronized transmission of Hoare's [19] *Communicating Sequential Processes* in which the sending and receiving processes of a value must act simultaneously; and the connection networks of distributed programming languages such as Argus [27] in which values, addressed to the intended recipient, are deposited and arrive arbitrarily interleaved with other "mail." Ultimately, the universality of our model's programming language, the dataflow graph, depends on either the ease with which programs incorporating these three forms of communication can be translated into dataflow or the ease with which "interesting" non-determinate applications can be programmed in dataflow. The translation route to universality is not difficult: a shared memory and a distributed connection network can be modeled by either processes or dataflow graphs, and the synchronized transmission of a value can be simulated with a pair of dataflow links. Although the programming of large applications in dataflow graphs would certainly be a tedious and unpleasant task, Arvind, Gostelow, and Plouffe [6], Dennis [13], and Jayaraman and Keller [21] have developed high-level programming languages which are trivially mapped into dataflow graphs and have used them to write programs for the most difficult of non-determinate applications, resource control. The abilities to model many forms of non-determinate communication and to program many kinds of non-determinate applications combine to make dataflow a very general model of non-determinate computation.

Between the introductory and concluding chapters of this thesis, we make two passes over the problem of formalizing non-determinacy and our solution to it. Each pass will consist of two chapters — the first will discuss the programming language, dataflow graphs, and the second will discuss the semantic model, scenario sets.

The first pass will be informal and somewhat tutorial. The next chapter, Chapter 2, is a thorough introduction to dataflow, with particular emphasis on presenting the operational semantics of dataflow and introducing a few dataflow operators useful in the specification of non-determinate computation. *Token-pushing*, the process of moving tokens, or values, through a dataflow graph by the *firing* of operators, forms the operational semantics of dataflow. This mechanism plays an important role in the thesis by providing a "target" for the more abstract semantics of scenarios. Token-pushing is "controlled" by individual operator *firing rules* specifying when each operator accepts values from its input ports and produces values at its output ports. Most dataflow operators (especially those corresponding to elementary arithmetic and logical operations) have simple firing rules: they fire when values are present at each input port and produce values at each output port. However, special *stream* operators with non-trivial firing rules are needed to specify inter-process communication. We will use only one non-determinate stream operator, the **merge** which interleaves two input streams into one output stream, in our examples. In Chapter 2, we briefly explain how these stream operators may be used to solve resource control problems.

Chapter 3 is a survey of previous semantic models for dataflow computation and an introduction to scenarios. Kahn [22] has shown that a determinate operator can be characterized as a function mapping the sequences received at its input ports to the sequences produced at its output ports and that the computation of a network of determinate operators can be derived as the least fixed point of a set of functional equations inferred from its operators. We will examine the shortcomings often encountered in applying fixed point semantics to non-determinacy and will show that the natural extension of Kahn's representation to non-determinate computation, the characterization of operators as relations, is insufficiently detailed for the development of a semantic theory consistent with the dataflow firing rules. By augmenting this relation with a *causality order*, scenarios are obtained. The scenario model of computation is the rule for constructing the scenarios of a network from the scenarios of its components.

The second pass will be decidedly more formal: in it our methodology and notation will be algebraic. In Chapter 4, we develop an algebra for dataflow graphs. The algebra is very simple. Stripped of its algebraic terminology, it is nothing more than a syntax for describing dataflow graphs. The dataflow graph algebra has three operators: the first joins two independent graphs into one, the second relabels a graph port, and the third connects a graph output port to a graph input port. Many of the ideas of this chapter were inspired by the flowgraph algebra of Milne and Milner [30].

Chapter 5 presents an algebra for scenarios. First, a formal definition of scenarios is given with emphasis placed on stating the restrictions on the causality order. The scenario algebra has three composition operators similar to those of the dataflow graph algebra. By showing that the scenario set model of computation is consistent with the operational model of Chapter 2, we prove that scenario sets are an abstract representation of non–determinate dataflow networks.

In the conclusion of this thesis, we discuss Pratt's [37] recent generalization of the scenario model and also enter the ongoing fairness "controversy" by observing the apparent immunity of the scenario composition rules to the "problem" of the fair merge. Finally, some open problems for future research are stated.

2. Dataflow Graphs and Languages

Dataflow graphs were developed as an instruction set for dataflow machines. The goal of this machine language was to provide a highly concurrent representation of computation in which the needless constraints on concurrent instruction execution implied by sequential machine languages are eliminated and only those constraints implied by the dependence of one instruction on the data produced by another are retained.

The nodes of a dataflow graph are called *operators*. The simplest operators perform elementary arithmetic and logical operations and correspond one-to-one with dataflow machine language instructions. When an operator is *fired*, *i. e.* executed, it removes *tokens*, *i. e.* values, from its input ports and produces tokens at its output ports. Each operator has a set of *firing rules* (conditions on the presence of input tokens) specifying when it is enabled for firing.

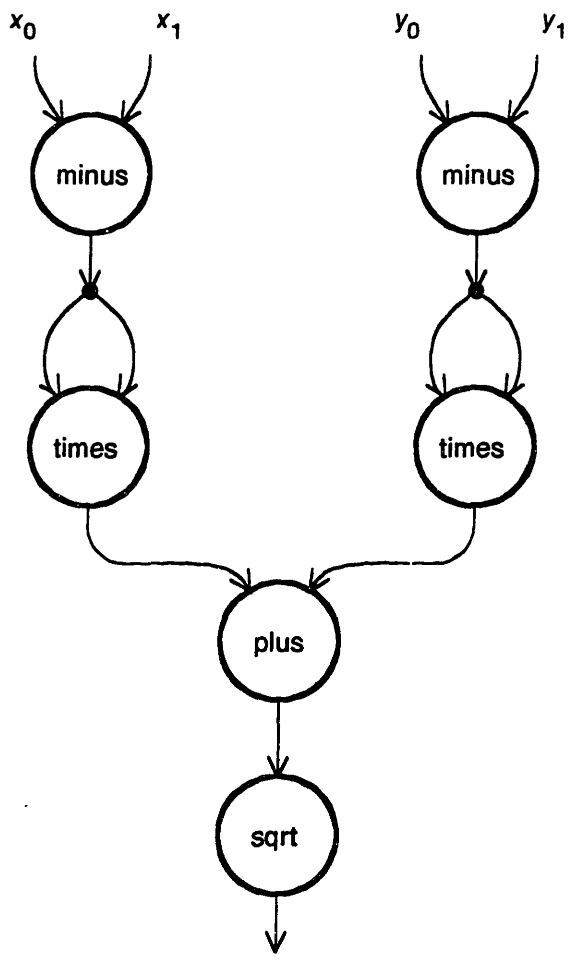
The links of the dataflow graph connect operator output ports to operator input ports. During the execution of a dataflow program values *flow* through these links from one operator to another. Not all operator ports of the graph should be connected: the unconnected operator ports become the ports of the graph itself. By considering graphs themselves to be operators, large graphs can be built in a modular fashion from smaller ones.

In Figure 2.1, a dataflow graph for computing the distance:

$$\mathbf{sqrt}(\,(x_0 - x_1)^2 + (y_0 - y_1)^2\,)$$

between two points (x_0, y_0) and (x_1, y_1) is illustrated. When this graph is executed, concurrency is obtained by the simultaneous execution of the difference and product operators. Dataflow researchers believe that in very large computations hundreds and thousands of operations may be performed in parallel and have proposed machines capable of exploiting such massive parallelism. Furthermore, note that the distance dataflow graph need not compute only one distance at a time. Additional concurrency may be obtained by pipelining several computations

Figure 2.1. An example dataflow graph



$\mathbf{sqrt}((x_0 - x_1)^2 + (y_0 - y_1)^2)$

---

through the graph.

Many high-level, algorithmic languages [2, 5, 12, 29] have been designed for the specific task of programming dataflow computers. These languages are quite conventional in appearance and would not frighten the average programmer. Perhaps the most "radical" action of dataflow language designers has been the banishment of side effects, an action increasingly appreciated for its role in simplifying programming language semantics [42]. Readers interested in learning more about dataflow languages should read Ackerman's [1] tutorial on the subject.

2.1 Dataflow Streams

Weng [44] introduced the stream data type into a side effect-free dataflow language to allow the programming of history-sensitive computation. At the graph level, a stream is the sequence, or history, of values passing through a graph link during the course of a dataflow computation. At the language level, a stream is a data type with operations similar to those of LISP lists. In this thesis we will write a stream by the juxtaposition of its components: for example, 5 6 will denote the two-element stream whose first value is 5 and second value is 6. The zero-element empty stream will be denoted Λ.[1]

We will use only three determinate stream operators in our discussion:

(1), **first**, when applied to a stream yields the first value of the stream;

(2), **rest**, when applied to a stream yields the stream with its first value removed;

(3), **cons**, when applied to a simple value and a stream yields the stream formed by concatenating the value onto the beginning of the stream.

Thus, for a value *x* and a stream *S*:

$$x = \textbf{first}(\,\textbf{cons}(x, S)\,)$$
$$S = \textbf{rest}(\,\textbf{cons}(x, S)\,)$$

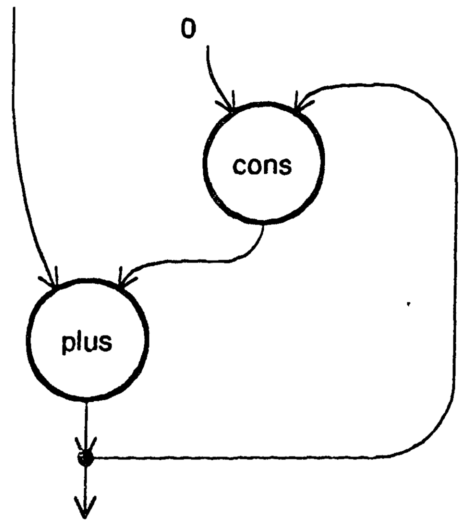
Stream operators have special firing rules that ensure that they do not wait for entire streams to appear on their input links before firing. Instead, as soon as an individual value of an input stream arrives, it is routed to the appropriate output. For example, as soon as a value appears on its leftmost port, the **cons** operator fires, absorbing that input value and producing it as output. Its firing rule then changes to pass all the rightmost inputs through the output port.

1. Our empty stream is different from Weng's. In his implementation all finite streams are terminated with a special end-of-stream value, and, consequently, one can test for the emptiness of a stream. It should be noted that it is not difficult to implement Weng's streams with ours.

Dataflow graphs exhibiting history-sensitive behavior may be constructed by joining stream operators together with feedback, that is, in cycles. In Figure 2.2, a graph which receives an input stream and produces an output stream whose *n*'th value is the sum of the first *n* input values is drawn. Interconnections with feedback can be adapted to a wide range of history-sensitive computations, including those as complicated as the interaction of a computer system with terminal input and output streams.

Although we have only discussed how streams are used in dataflow languages, conventional programs with input/output primitives causing side effects can also be considered to generate streams. The first use of streams for parallel computation was Kahn's [22] "simple language for parallel programming." In this language, processes were written in an Algol-like language with two primitives, **get** and **put**, for reading input from and writing output to process channels. The **cons** stream operator may be written in language very similar to Kahn's as:

Figure 2.2. A history-sensitive dataflow graph



```
process cons in (IN1, IN2) out (OUT)
    put get(IN1) on OUT
    while (true) do
        put get(IN2) on OUT
    end
end cons
```

Kahn also developed a notation for connecting processes together to form networks. Because the interconnection syntax only mentions processes externally, *i. e.* at the level at which they may be considered stream receivers and producers, it is equally adaptable to the task of constructing dataflow graphs from operators or subgraphs. For example, the cyclic graph of Figure 2.2 could be specified as:

```
network Sum in (IN) out (OUT) internal (X)
    X ← cons(0, OUT)
    OUT ← plus(IN, X)
end Sum
```

Kahn carefully defined his language so that all processes and networks written in it would be determinate, *i. e.* always produce the same tuple of output streams in response to a given input stream tuple. Determinacy was guaranteed by disallowing two kinds of potential races: (1), tokens from different channels racing to be the first read — avoided by allowing only one output channel to be connected to an input channel and by allowing the **get** primitive to wait on only one input channel; and (2), tokens on a channel overtaking or overwriting each other — avoided by requiring all channels to be implemented as FIFO queues of unbounded length.

Kahn's language predated and, in many ways, inspired dataflow streams. However, our primary motivation for presenting his language here is not to provide a historical perspective on streams, but rather to indicate that dataflow graphs and, consequently, our semantic theories are relevant to many styles of expressing parallel computation. Readers uncomfortable (or unimpressed) with the doctrines of dataflow are encouraged to interpret our results consistent with a more conventional creed of concurrency.

2.2 A Non-determinate Dataflow Operator

There are many good reasons for making a language determinate. In addition to the considerable "semantic profit" examined in Chapter 3, there is a benefit which would be appreciated by anyone who has ever tried to debug a parallel program: a determinate computation has no time-dependent behavior and is consequently always repeatable. However, there are some applications in which the time-independent reproducibility of determinate computation would be disastrous, and for this reason we must introduce non-determinate operators into our dataflow language.

Consider a determinate computer system with one printer, but several terminals. The users of such a system may be considered as competing for the use of the printer. If user Alice requests the printing of a file one week before user Zane, we expect Alice's request to be honored first. However, since the computer system is completely determinate, the time difference between the two requests must not affect the order in which the requests are serviced: Alice must always go first. Had Zane actually entered his request first, he must wait indefinitely, perhaps eternally, for Alice to make her request so that it could be honored before his own.

The behavior we really want of our system is non-determinate. We do want the system to service the requests of both users, but we do not want the order of service to be determined *a priori*. Our language can be extended to include non-determinate behavior by the addition of a single operator, the **merge**. The **merge** receives two input streams and non-deterministically interleaves them into a single output stream. In its intended implementation, the **merge** fires as soon as a value appears on either of its input ports and produces that value at its output port. There are three possible output stream responses when **merge** is applied to the input streams 5 6 and 7:

merge(5 6, 7) = 5 6 7 *or* 5 7 6 *or* 7 5 6

Thus, although **merge** has a very simple firing rule, its behavior is obviously not a simple function of its input data.

By merging the requests of users competing for a resource into a single stream and then "passing" that stream to a resource controller, a large class of resource allocation problems can be programmed with non-determinate dataflow languages. Furthermore, if each request is tagged to indicate its originator, the resource controller can direct its responses to the proper recipients. Arvind, Gostelow, and Plouffe [6] have developed dataflow *managers* which incorporate the **merge** operator implicitly and have solved a large number of the "popular" readers-and-writers problems in their dataflow language, *Id*. In a survey of non-determinate dataflow programming (recommended for readers interesting in language aspects of this issue), Arvind and this author [3] present a shared printer manager which gives priority to short print requests.

2.3 Operational Semantics

The dataflow graphs described in this chapter will be the programming language on which our model of non-determinate computation is based. The operational semantics of dataflow graph execution may be inferred from the firing rules of dataflow operators. The operational model of execution is called *token-pushing*: the performance of a series of state transitions corresponding to the firings of graph operators. The state of a graph is the product of the states of its component operators and the states, *i. e.* contents, of its links. Each transition of the graph is the firing of a dataflow operator. In firing, an operator can remove tokens (values) from its input ports and add tokens to its output ports. Because an operator may represent something as complex as a subgraph or a process, the firing rules and the states of operators may be arbitrarily intricate. It should also be noted that many firings taken by these "large" operators will be internal and will not affect the contents of any of the graph links.

The operational semantics of dataflow graphs is a *global state* model of computation. Because operators may be widely distributed physically and because firings may occur concurrently, it will in general be impossible actually to observe an executing graph performing a discrete state transition or even to determine the state of a graph. However, although firings may be concurrent, they may never conflict (because operator input links are disjoint), and for that reason token-pushing is a faithful representation of dataflow execution.

In the succeeding chapters of this thesis, we will use token-pushing as a standard to measure other, more abstract, models of non-determinate computation. We will only invoke token-pushing informally — to derive the result of executing a dataflow graph so that the actual result may be compared with those predicted by other models. Although formal properties of token-pushing will never be used, nonetheless an intuitive understanding of an operational semantics of parallel computation is essential for the appreciation of the more abstract ones.

3. Scenarios: A Model of Non-determinate Computation

The first goal of any semantic theory is to give computational meaning to programs. Other goals may depend upon the intended audience of the formal specification. In *operational* semantic specifications, programs are translated into the code of an abstract state machine. In theory, an operational semantic definition should be of use to the implementor of a programming language; in practice, this usefulness may be greatly limited by the distance from the abstract machine to existing computing machinery. In *denotational* semantic specifications, the syntactic elements of a program (expressions, statements, *etc.*) are mapped directly into mathematical entities denoting their meanings. Thus a program can be understood directly in terms of its syntactic structure without reference to its compilation to either an abstract or a real machine. Another very important goal of a denotational semantic theory should be to provide a usable methodology for analyzing programs. The test of this methodology is greatest when applied to large programs.

We are primarily interested in giving a denotational semantics for dataflow graphs. We will assign a meaning to each dataflow graph without reference to how it will be executed. Nevertheless, we are very interested in implementation-level operational[1] models of dataflow computation because we must take great care to ensure that our denotational semantics is consistent with real-world dataflow graph execution.

The earliest attempts to characterize the semantics of parallelism were formalizations of a *global state model* of computation [23] very similar to token-pushing, our operational model of dataflow computation. As we saw in the preceding chapter, in these models the state of a parallel computation is the states of its constituent operators (or processes) plus the values in transit

1. Our use of the word "operational" here is somewhat incorrect. See Stoy's [41] book, for a more precise presentation of the different flavors of semantic theory.

between these operators, and the state transitions of the computation are the state transitions of its operators. The sequence of state transitions performed by the system during a computation is called an *execution sequence.*

In the global state model, most pairs of state transitions are independent, *i. e.*, they occur at different operators and have no real causal relationship. Consequently, their positions within the execution sequence may be interchanged without affecting the result (produced output values) of the computation. In general, as the size of the parallel program increases, not only the number of execution sequences, but also the number corresponding to the "same" computation, increases exponentially. Furthermore, there is an increase in the detail of the execution sequence, because each of its transitions corresponds to the action of an operator playing a very small role within a very large computation. For these reasons, the analysis of large systems with the global state model is a torment for both semanticist and programmer.

The global execution sequences of a parallel program do reveal the meaning of the program. They do give programs a meaning other than their own syntax. However, when a program is itself used as a component of another, larger program, there is very little advantage to viewing the component program as a collection of execution sequences as opposed to a collection of interconnected processes. The useful semantic quality which the global execution model lacks is *abstraction*, the removal of externally uninteresting details of program syntax from the input–output behavioral specification of a program.

In this chapter we will discuss several previously proposed models of parallel computation and will present one model of our own. In each of these models we will look for three important qualities: faithfulness, abstractness, and simplicity.

3.1 Fixed Point Semantics for Determinate Networks

History functions are an adequate computational model for a very important class of dataflow graphs, those composed of determinate operators. During a graph computation, each input port receives and each output port transmits a *history* (sequence) of values. Consequently, every graph and operator has a *history relation* associating vectors of input histories, one history for each input port, with vectors of output histories, one history for each output port.

A graph, or operator, is *determinate* if there is only one possible output history vector for each input history vector. Note that this concept of determinacy, by virtue of being defined without reference to timing, is time-independent. Neither absolute differences in the arrival times of input values at a single port nor relative differences in the arrival times of input values at different ports can influence the output of a determinate system. A system producing an integer value equal to the spacing in micro-seconds between its first two input values is *not* determinate. Neither is a system producing as its only output the first input value to arrive at either of two input ports.

The time-independent property of determinate systems has one very important practical significance: determinate computations are easily repeatable. This means that determinate dataflow graphs may be executed with great concurrency without the dangers of non-reproducibility that may occur in parallel shared-memory computer architectures. One of the important early achievements of dataflow research [12] was the development of a set of determinate dataflow operators which could be used to generate graphs implementing all programs of a general-purpose (though applicative) programming language.

Patil [35] proved that determinate systems are closed under interconnection. Therefore, a dataflow graph is determinate if its constituent operators are. In Chapter 2 we introduced Kahn's [22] "simple language for parallel programming." Any process written using Kahn's **get** and **put** operators is determinate, and consequently any network of such processes is also determinate. Kahn [22] used the fixed point methods of Scott [39, 41, 42] to define semantically the results of executing the networks generated with his programming language.

Any determinate process (operator, network, or even graph) P may be modeled by its history function $\mathfrak{F}[\![P]\!]$. When X is the tuple of histories presented on the input ports of P, $\mathfrak{F}[\![P]\!](X)$ will be the tuple of histories produced at the output ports of P.

In Chapter 2, we presented a network Sum whose n'th output was the sum of its first n inputs. In Kahn's language Sum was defined as:

```
network Sum in (IN) out (OUT) internal (X)
    X ← cons(0, OUT)
    OUT ← plus(IN, X)
end Sum
```

The two component processes of Sum have the following very simple history functions:

$\mathfrak{F}[\![\mathbf{cons}]\!](\Lambda, Y) = \Lambda$
$\mathfrak{F}[\![\mathbf{cons}]\!](x\ X, Y) = x\ Y$

$\mathfrak{F}[\![\text{plus}]\!](\Lambda, Y) = \Lambda$
$\mathfrak{F}[\![\text{plus}]\!](X, \Lambda) = \Lambda$
$\mathfrak{F}[\![\text{plus}]\!](x\ X, y\ Y) = x+y\ \ \mathfrak{F}[\![\text{plus}]\!](X, Y)$
Λ is the empty history
x, y are single values
X, Y are arbitrary (possibly empty) sequences of values

By replacing each process name within a network definition with its history function, a set of simultaneous equations is constructed:

$\text{X} = \mathfrak{F}[\![\mathbf{cons}]\!](0, \text{OUT})$
$\text{OUT} = \mathfrak{F}[\![\text{plus}]\!](\text{IN}, \text{X})$

When the value of the input history IN is fixed, the least fixed point (solution) of the set of equations gives, as the value of OUT, the output history generated by executing Sum with input history IN. Thus may the history function of Sum be determined.

This method of finding the result of executing a network computation may be applied to any determinate dataflow graph. The existence of the least fixed point solution depends on three conditions:

> (1), The range and domain of any history function are complete partial orders, each with a least element. The prefix order $\sqsubseteq$ is used to satisfy this condition. Given histories X and Y, $X \sqsubseteq Y$ if and only if $XZ = Y$ for some sequence Z. History vectors are ordered by the natural extension of $\sqsubseteq$. In a complete partial order, every increasing chain $X_0 \sqsubseteq X_1 \sqsubseteq \ldots$ has a least upper bound $\sqcup X_n$. $\sqsubseteq$ is used to determine the *least* fixed point.
>
> (2), For any determinate system S, $\mathcal{F}[\![S]\!]$ is *monotonic*; *i. e.* $X \sqsubseteq Y$ implies $\mathcal{F}[\![S]\!](X) \sqsubseteq \mathcal{F}[\![S]\!](Y)$.
>
> (3), For any determinate system S, $\mathcal{F}[\![S]\!]$ is *continuous*; *i. e.* given an increasing chain $X_0 \sqsubseteq X_1 \sqsubseteq \ldots$, $\mathcal{F}[\![S]\!](\sqcup X_n) = \sqcup \mathcal{F}[\![S]\!](X_n)$.

Monotonicity and continuity are two common-sense axioms of computation behavior. The physical implication of monotonicity is that the more input a system receives the more output it may produce. The physical implication of continuity is that a system's response to an infinite input is the "limit" of its response to finite prefixes of that input. Formally, the continuity of $\mathcal{F}[\![S]\!]$ implies the monotonicity of $\mathcal{F}[\![S]\!]$.

History functions are a well-suited semantic model for determinate dataflow computation. They are "maximally" abstract, containing no more information than the system input-output behavior. They are also simple to use in the derivation of the semantic characterization of large systems. Furthermore, Faustini [16] has shown history functions to be faithful to the underlying global-state operational model of dataflow computation by formally proving the equivalence of the two models.

3.2 Fixed Point Semantics for Non-determinate Computation

Not all useful dataflow computation is determinate. In Chapter 2, we discussed the need for non-determinate, time-dependent computation and introduced a single "universal" non-determinate operator, **merge**. The **merge** receives two input streams and indeterminately interleaves them, roughly in order of time of arrival, into a single output stream.

Fixed point semantics "work" for determinate computation because the dataflow operators are monotonic and continuous over the domain of history vectors under the prefix ordering. However, non-determinate dataflow computations are not functions over this domain. At the least, a non-determinate computation is a function mapping each tuple of input histories into a set of possible output histories or, equivalently, is a *history relation* associating tuples of input histories with tuples of output histories. In order easily to apply Scott's [39] fixed point theory to non-determinate computation represented by history relations, an ordering (a *complete partial* ordering) must be defined on the domain of history relations, the powerset of history tuples, so that all determinate and non-determinate dataflow operators and all graphs constructed from them are both monotonic and continuous.

However, no such ordering exists for history relations. This can be easily seen by considering two data systems constructed using the **merge** operator. The first system we call EitherStream. Applied to two input streams, EitherStream indeterminately chooses one of them to become its output stream. As a history relation, EitherStream may be specified as:

EitherStream(X, Y) = $\{X, Y\}$

Using the stream operators of Weng [44] introduced in Chapter 2 along with the **merge** operator, EitherStream may be programmed as:

```
EitherStream(X, Y) =
   if first(merge(true, false)) then X else Y endif
```

Applied to the input history vector $\langle \Lambda, 5 \rangle$ consisting of the empty stream and the stream consisting of the single value five, EitherStream yields either the empty stream or the single-value stream. Applied to yet more input, in particular the input history tuple $\langle 5, 5 \rangle$, EitherStream yields the stream consisting of the single value five. Consequently, if monotonicity holds for all history relations, $\{\Lambda, 5\}$ must be less than $\{5\}$.

The second dataflow system we call FirstFiveOnTwo. This system internally merges its two input streams and passes the merged inputs to an internal process FirstFiveOnOne which will produce a single output value five if its first input value is five and will otherwise produce no output. In Kahn's [22] language, the process FirstFiveOnOne may be programmed as:

```
process FirstFiveOnOne in (IN) out (OUT)
    if get(IN) = 5 then
        put 5 on OUT
    end
end FirstFiveOnOne
```

and FirstFiveOnTwo may be considered the "functional" composition FirstFiveOnOne $\circ$ **merge**. When FirstFiveOnTwo is applied to the input history tuple $\langle \Lambda, 5 \rangle$, FirstFiveOnOne will receive a single input value five and transmit it as its output. Therefore:

$$\text{FirstFiveOnTwo}(\langle \Lambda, 5 \rangle) = \{5\}$$

However, when FirstFiveOnTwo is applied to more input, in particular $\langle 6, 5 \rangle$, FirstFiveOnOne may be applied to either 6 5 or 5 6 and, consequently, its output stream will either be empty or consist of the single value five. Therefore:

$$\text{FirstFiveOnTwo}(\langle 6, 5 \rangle) = \{\Lambda, 5\}$$

Now we have a dilemma for fixed point theory: with FirstFiveOnTwo monotonicity forces us to consider $\{5\}$ to be less than $\{\Lambda, 5\}$, but with EitherStream monotonicity forces $\{\Lambda, 5\}$ to be less than $\{5\}$. This contradiction implies that we cannot use a straightforward extension of Kahn's [22] fixed point theory on non-determinate dataflow computations *represented as* history relations.[1]

In the remainder of this section, we will look at some proposed fixed point theories for non-determinate computation. Most of these theories have been designed for models of computation other than dataflow. Of these, the dataflow-oriented ones are based on underlying mathematical domains much more complicated than history relations.

Most orderings for non-determinate computation are based on the Egli-Milner [14, 31] order. This order is defined on the powerset of program values. For sets X and Y of values:

$$X \sqsubseteq_M Y \leftrightarrow \forall x \in X\ \exists y \in Y \text{ such that } x \sqsubseteq y \text{ and } \forall y \in Y\ \exists x \in X \text{ such that } x \sqsubseteq y.$$

Hence, $X \sqsubseteq_M Y$ implies that for every element of the lesser set there is a corresponding greater element in the greater set and *vice versa*. This ordering is best suited for use with languages where non-determinacy results from a choice operator **or**. The expression "exp_0 **or** exp_1" is evaluated by *first* choosing one of exp_0 and exp_1 to evaluate and then returning the result of that evaluation. The intuitive meaning of the Egli-Milner ordering is that as a computation progresses, every possible non-determinate branch is increased.

While the **or** operator is monotonic and continuous under the Egli-Milner order, a simple "enhancement" of it, McCarthy's [28] ambiguity operator **amb** is not. The expression "exp_0 **amb** exp_1" is evaluated by evaluating exp_0 and exp_1 simultaneously and returning the earlier to terminate. This operator is rather like the composition **first** $\circ$ **merge**. Letting $\perp$ denote the result of the non-terminating computation, the least element of our underlying domain of values, **amb** may be defined as:

1. In fact, it will be shown later in this chapter that in no semantic theory for non-determinate dataflow computation may dataflow graphs be adequately modeled by history relations.

$\perp$ **amb** $y = \{y\}$
x **amb** $\perp = \{x\}$
x **amb** $y = \{x, y\}$, if $x \neq \perp$ and $y \neq \perp$

Because 5 **amb** $\perp$, or $\{5\}$, is not less than 5 **amb** 6, or $\{5, 6\}$, under the Egli-Milner order, **amb** is not monotonic. Consequently, this ordering cannot be used to define semantically languages containing the **amb** operator or, in fact, any operator which "races" concurrent computations to produce its result.

Another problem arises when the Egli-Milner order is applied to the value domain of streams ordered by prefix inclusion. In this case, we find that both $\{\Lambda, 5\ 6\} \sqsubseteq_M \{\Lambda, 5, 5\ 6\}$ and $\{\Lambda, 5, 5\ 6\} \sqsubseteq_M \{\Lambda, 5\ 6\}$. The Egli-Milner order is a true partial order only when applied to value domains that are flat or discrete, that is, where $x \sqsubseteq y$ implies $x = y$ or $x = \perp$. Plotkin [36] resolved this problem by considering $\sqsubseteq_M$ to be an ordering of the *powerdomain* of equivalence classes defined by:

$$X \equiv Y \leftrightarrow X \sqsubseteq_M Y \wedge Y \sqsubseteq_M X$$

Under this equivalence a set X of streams is equivalent to the set $Con(Cl(X))$. The set $Cl(X)$ is the *closure* of X. It contains the limit points of all sequences that can be constructed from elements of X; that is, if there exist elements $x_0, x_1, \ldots$ in X such that $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ then $\sqcup x_n \in Cl(X)$. $Con(Cl(X))$ is the set which makes $Cl(X)$ *concave*; that is, if $x, z \in Cl(X)$ and $x \sqsubseteq y \sqsubseteq z$ then $y \in Con(Cl(X))$. Thus the set of all streams consisting totally of an even number of fives would be considered equivalent to the set of all streams, even the infinite one, consisting of only fives. Two systems producing these two different sets of possible output streams cannot be distinguished with this fixed point theory.

The semantic constructions of Lehmann [26] and Smyth [40] avoid equivalence classes of powersets by considering the powerdomain to be a category of multisets (sets in which elements may occur several times) instead of a partially ordered set of sets. This powerdomain construction is particularly obscure; however, it generalizes to one half of the Egli-Milner ordering:

$$X \sqsubseteq Y \leftrightarrow \forall y \in Y \; \exists x \in X \text{ such that } x \sqsubseteq y.$$

This order provides a means by which branches of a non-determinate computation may fail as the computation progresses. However, many useful non-determinate operators such as the previously discussed **amb** remain non-monotonic (more precisely, are not functors) under this weaker ordering, and for this reason it cannot be used to describe non-determinate dataflow computation.

Park [33] has developed a fixed point theory for a somewhat restricted class of non-determinate dataflow computation. He uses two fixed points, the least *and* the greatest, to define the result of the non-determinate operator called the **fairmerge**. The **fairmerge** agrees with our **merge** only on infinite input histories. It may "deadlock" when one of its input histories is finite and ignore values of the other input history.

In a later paper [34] Park defines the semantics of unrestricted non-determinate dataflow computation by considering streams to contain *hiatons*, non-existent values that cannot be manipulated by programs but merely serve as semantic place holders. Hiatons may be considered as the tokens produced by a system whenever it needs to mark time. All the fixed point equations of this theory have unique solutions, thus avoiding much of the complexity of Park's earlier work. However, this uniqueness results from requiring all functions to be *>-functions*, *i. e.* functions which produce output histories longer than their input histories. Any function may be easily made into a >-function by the introduction of hiatons into its output. This explicit introduction of time into the representation of computation does lead to some loss of abstraction, even for determinate graphs. For example, a single dataflow identity operator and two concatenated identity operators will not share the same hiatonized semantic characterization.

Recently, Broy [10] has proposed an elaborate fixed point theory for dataflow computation. The derivation of a dataflow graph execution following his techniques is quite difficult, involving the solution of three fixed point equations (one obtained by the replacement of some graph operators) under three different orderings of the set of streams. Furthermore, this theory is incapable of modeling the **merge** used in this thesis. Our **merge** is *fair*[1] in that it will eventually accept any of its input tokens. The fair **merge** is necessary for any resource management application [3] in which a response is to be guaranteed for any allocation request. Broy's theory can only be used with the *unfair* **merge** which admits the possibility of one input "hogging" the operator's output.

Kosinski [25] has defined a denotational semantics in which values are "tagged" with the sequence of non-determinate **merge** "decisions" which caused their generation. Networks are then defined as functions mapping tagged input history tuples into sets of tagged output history tuples. The derivation of a network's semantic characterization from those of its components is the solution of a fixed point equation which includes the matching of value tags to determine consistent input histories for processes with multiple input ports.

Clinger [11] has pointed out a significant technical shortcoming in Kosinski's ordering of sets of tagged histories. Even if this problem were overcome, Kosinski's formal characterization of dataflow graphs would still contain many details of internal operator communication together with the information concerning externally observable behavior contained in history relations. For large graphs these internal details, most of them irrelevant, could easily overwhelm the useful information.

Not all research in the semantics of non-determinate computation occurs within the context of dataflow. Perhaps the most active research in non-determinacy is concerned with formalizing

1. Again, this is not the *fairmerge* operator of Park's earlier paper [33] which interleaves all its input tokens only if both input streams are infinite. The word "fair," at least in the context of dataflow, is yet another example of a British-American difference in word usage.

the execution of concurrent processes which communicate by simultaneous and co-operative action. Most of the programming languages utilizing this type of communication are akin to Hoare's [19] *CSP* (*Communicating Sequential Processes*) or Milner's [32] *CCS* (*Calculus of Communicating Systems*). In these languages when two processes wish to communicate both must do so synchronously: one process will receive a value while the other simultaneously sends it. Although the communication itself will only involve two processes, a process may offer to interact with several processes even if it will eventually succeed in communicating with at most one. The non-determinacy of synchronized communication results from the competition of these potential communicators.

Viewed externally, the processes of these languages appear to be performing a series of message transmissions and receptions. Consequently, it is only natural that their semantic models should have a similar structure. Two well-known semantic models for this form of computation are the *synchronization trees* of Milner [32] and the *failure sets* of Hoare, Brookes, and Roscoe [9, 20].

In Milner's synchronization tree model a process is represented by a tree whose nodes correspond to possible states of the process. The links of the tree correspond to state changes accomplished through communication. However, not all links represent interaction with external processes. A process may itself be composed of several subprocesses which engage in internal communication. A synchronization tree may have links corresponding to *silent* transitions resulting from these internal transactions. Obviously, many of these silent transitions represent an internal detail of process computation that could never be uncovered by external manipulation of the process. Milner prevents the potential loss of abstraction that could result from the incorporation of silent transitions in synchronization trees by defining a notion of *observation equivalence* in which two processes are considered equivalent if their behavior is indistinguishable in every context, that is, if they can be substituted for each other as subprocesses of any system without noticeably changing that system's externally observable behavior. Milner adopts as his model of non-determinate computation the quotient set of synchronization trees under observation equivalence.

In the failure set model of Hoare, Brookes and Roscoe [9, 20], a process is represented as a set of pairs of the form $\langle s, X \rangle$ where s is a *trace* (sequence of communications) which the process may perform, immediately after which it cannot perform any communication within the *refusal set* X. Although the refusal set, being a statement of what the process cannot do, may seem unnatural, its role in the failure set model is similar to that of silent transitions in the synchronization tree model. Refusal sets may be used to distinguish those instances in which internal transitions which immediately follow the performance of a trace affect the ability of a process to participate in future external communication.

A very lucid and concise summary and comparison of these two semantic models has been given by Brookes [9]. Rather than elaborate any further on the differences between the two theories, we will restrict out attention to their most apparent commonality, the representation (more or less) of a concurrent process as a set of totally ordered sequences of external communications, and examine the consequences of applying this representation to dataflow, a computational model admittedly outside the intended application area of both semantic formalisms.

The relevance of synchronized communication models to dataflow computation is not at all obvious. Dataflow communication is not only asynchronous but is even queued through unbounded buffers. However, it is not hard to transform a dataflow graph into a synchronized process: all that is required is the placement of synchronized unbounded-buffer processes on all its ports. Each buffer process must always be ready to receive values at its input port, and, as long as the buffer is non-empty, it must be ready to transmit values at its output port.

This transformation once performed allows the use of synchronization trees and failure sets in the analysis of dataflow graphs. However, these models when applied to dataflow will retain the underlying sequentiality of synchronous communication. In particular, dataflow graphs will appear to communicate in totally ordered sequences although at the implementation level values may be received and produced concurrently on different ports. In both the CCS and CSP models, concurrent communication must be represented as a collection of all the alternative interleavings of the communication. Milner [31] admits that synchronization trees do not represent "true concurrency" but argues for his sequential approach on the grounds that it is simple and observable (after all, only one event can be seen at a time). Furthermore, Milner demonstrates that he can analyze with little difficulty many truly concurrent systems.

However, the simplicity of a formal model must be measured relative to the programming constructs to which it is to be applied. Some very important dataflow concepts change when graphs are viewed as performing sequential synchronous communication. One such concept is determinacy. Dataflow graphs once considered determinate may now appear to have an ability to non-determinately "choose" the order in which to produce output values at different ports.

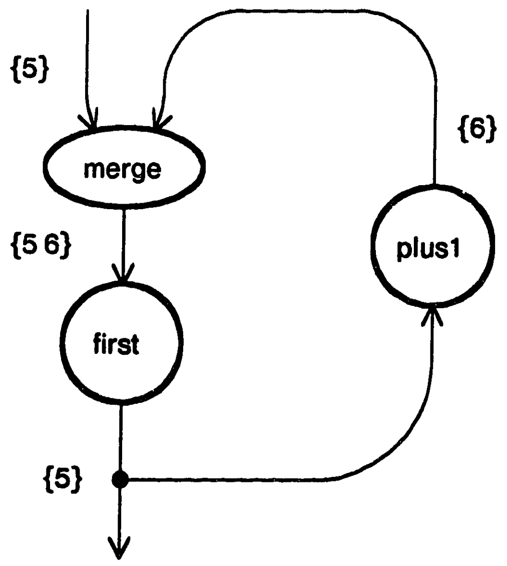
Admittedly, synchronization trees and failure sets could be applied to dataflow, and the resultant theory would certainly be adequate and even simple when compared to many of the previously-mentioned theories of non-determinate dataflow computation. However, one goal in this thesis is to retain, as far as possible, the history-orientation of Kahn [22] and to construct a semantic theory in the spirit of the asynchrony of dataflow.

3.3 The Incompleteness of History Relations

By comparison to many of the preceding models of non-determinate dataflow computation, history relations are quite simple. But unfortunately history relations lack sufficient detail to describe the behavior of interconnected graphs. In the remainder of this chapter, we will show this incompleteness and will describe a natural extension of history relations that is adequate for non-determinate computation. These results have been previously reported in a paper of this author and Ackerman [8]. The incompleteness of history relations will be demonstrated by exhibiting two dataflow graphs S_1 and S_2 which have the same history relation, but which cannot be substituted for each other as components of a larger network without changing that network's history relation. Consequently, there can be no "correct" interconnection rules for networks represented by history relations. Formally interpreted within the algebraic terminology employed by Milne and Milner [30], the incompleteness result shows that the function mapping each dataflow graph into its history relation cannot be extended to a morphism of the algebra of dataflow graphs.

It should be noted that our demonstration of the incompleteness of history relations differs fundamentally from that of Keller [24]. His demonstration uses a graph similar to that of Figure 3.1. Beside each graph arc is written all its possible histories with graph input history 5. When computation begins, the value five passes through the leftmost input port of the **merge** and causes six to appear on the rightmost input port. Eventually, the **merge** will produce the output history 5 6. However, Keller, "pursuing the incremental approach" to semantics, assumes that, in any semantic theory in which non-determinate processes are represented as history relations, the output of the **merge** operator could be any element of **merge**(5, 6), including the clearly physically impossible 6 5; given that if "these inputs were presented externally, this would certainly be the case." From this merge anomaly he concludes that history relations must be extended to include causality relations between individual elements of different histories. We do,

Figure 3.1. Keller's Example (with slight modification)



of course, accept Keller's ultimate conclusion; however, we believe that his anomaly demonstrates at most that history relation interconnection rules which ignore causality fail. It does not show that the necessary causality relations cannot be inferred from history relations by, for example, requiring that later output of a **merge** does not "rewrite" earlier output.[1] Keller shows there are no *easy* interconnection rules for networks characterized by history relations. We show there are *no* interconnection rules.

1. There are several subalgebras of data flow graphs which exhibit Keller's merge anomaly but which are amenable to analysis by history relations. A very simple one, consisting of only two-input two-output graphs, may be constructed using a single non-determinate dataflow operator computing **plus1** ∘ **first** ∘ **merge** at both output ports (always the same value on both, that is, the output values are not separately computed) and a single graph interconnection rule of composing two graphs by placing them side-by-side and then connecting the rightmost ports of the left graph to the leftmost ports of the right graph.

Let S_1 and S_2 be the graphs shown in Figure 3.2. Syntactically, S_1 and S_2 may be written:

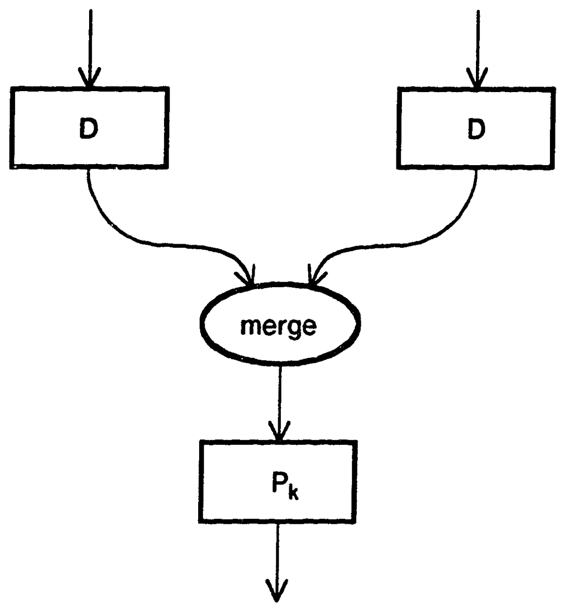
$$S_k(X, Y) = P_k(\text{merge}(D(X), D(Y)))$$

D, P_1, and P_2 are all determinate processes which produce at most two output values. Process D produces two copies of its first input value. In Kahn's [22] language, it may be written as:

```
process D in (IN) out (OUT)
    x ← get(IN)
    put x on OUT
    put x on OUT
end D
```

Both P_1 and P_2 allow their first two input values to pass through themselves as their first two output values. However, P_1 will produce its first output as soon as it receives its first input, while P_2 will not produce any output until it has received two input values. In Kahn's language P_1 and P_2 may be written as:

Figure 3.2. S_k for $k \in \{1, 2\}$



```
process $P_1$ in (IN) out (OUT)
    x ← get(IN)
    put x on OUT
    y ← get(IN)
    put y on OUT
end $P_1$

process $P_2$ in (IN) out (OUT)
    x ← get(IN)
    y ← get(IN)
    put x on OUT
    put y on OUT
end $P_2$
```

As history functions these three processes may be specified as:

$D(\Lambda) = \Lambda$
$D(x\ Z) = x\ x$

$P_1(\Lambda) = \Lambda$
$P_1(x) = x$
$P_1(x\ y\ Z) = x\ y$

$P_2(\Lambda) = \Lambda$
$P_2(x) = \Lambda$
$P_2(x\ y\ Z) = x\ y$

Λ is the empty history
x and y are single values
Z is an arbitrary (possibly empty) sequence of values

Despite the difference between P_1 and P_2, networks S_1 and S_2 have the same history relation representation. Neither network produces any output unless it receives some input. Suppose S_k receives the input stream $x\ X$ at its leftmost input port and no input at its rightmost port. Then the leftmost D process will produce the output history $x\ x$, while the rightmost D will produce nothing. The streams $x\ x$ and the empty stream will be merged into the stream $x\ x$. Regardless of whether S_k is S_1 or S_2, process P_k (P_1 or P_2) will receive $x\ x$, *two* input values, and will produce $x\ x$ as its, and hence S_k's, output. A similar situation occurs if input is received on the rightmost port only.

If S_k receives input at both input ports, in particular, the stream $x\ X$ on the rightmost and the stream $y\ Y$ on the leftmost, it will internally merge the two input streams $x\ x$ and $y\ y$ produced by the D processes and communicate the merged stream to P_k. Although there are six possible ways of interleaving $x\ x$ and $y\ y$, we (and the P_k) are only interested in the first two elements of these interleavings, and of these there are only four possibilities: $x\ x$, $x\ y$, $y\ x$, and $y\ y$. The output of P_k, and consequently S_k, may be any of these four sequences.

Because the D processes guarantee the generation of at least two input values for P_k as long as S_k receives any input, the differences between P_1 and P_2, and consequently S_1 and S_2, are avoided, at least at the history relation level of behavioral specification. S_1 and S_2 have the same history relation:

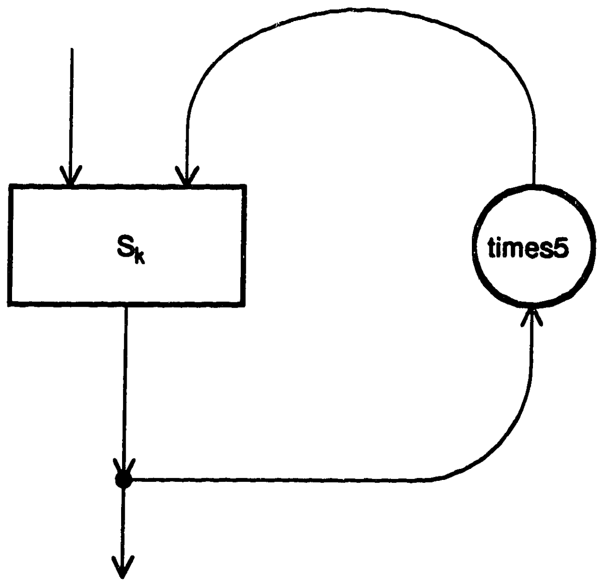
$$S_k(\Lambda, \Lambda) = \{\Lambda\}$$
$$S_k(x\ X, \Lambda) = \{x\ x\}$$
$$S_k(\Lambda, y\ Y) = \{y\ y\}$$
$$S_k(x\ X, y\ Y) = \{x\ x, x\ y, y\ x, y\ y\}$$

Λ is the empty history
x, y are single values
X, Y are arbitrary (possibly empty) sequences of values

However, there is a subtle difference in the behaviors of the two networks: S_2 will not produce its first output until its second output has been determined. These networks can be placed within a larger network which uncovers this difference.

Let T_1 and T_2 be the networks of Figure 3.3. T_k is a cyclic network containing the non-determinate process (subgraph) S_k and the determinate process **times5**, which multiplies each of its input values by five. T_k's single input port is the leftmost input port of S_k. The outputs of S_k are routed to two destinations: to the single output port of T_k and, through the **times5**

Figure 3.3. T_k **for** $k \in \{1, 2\}$

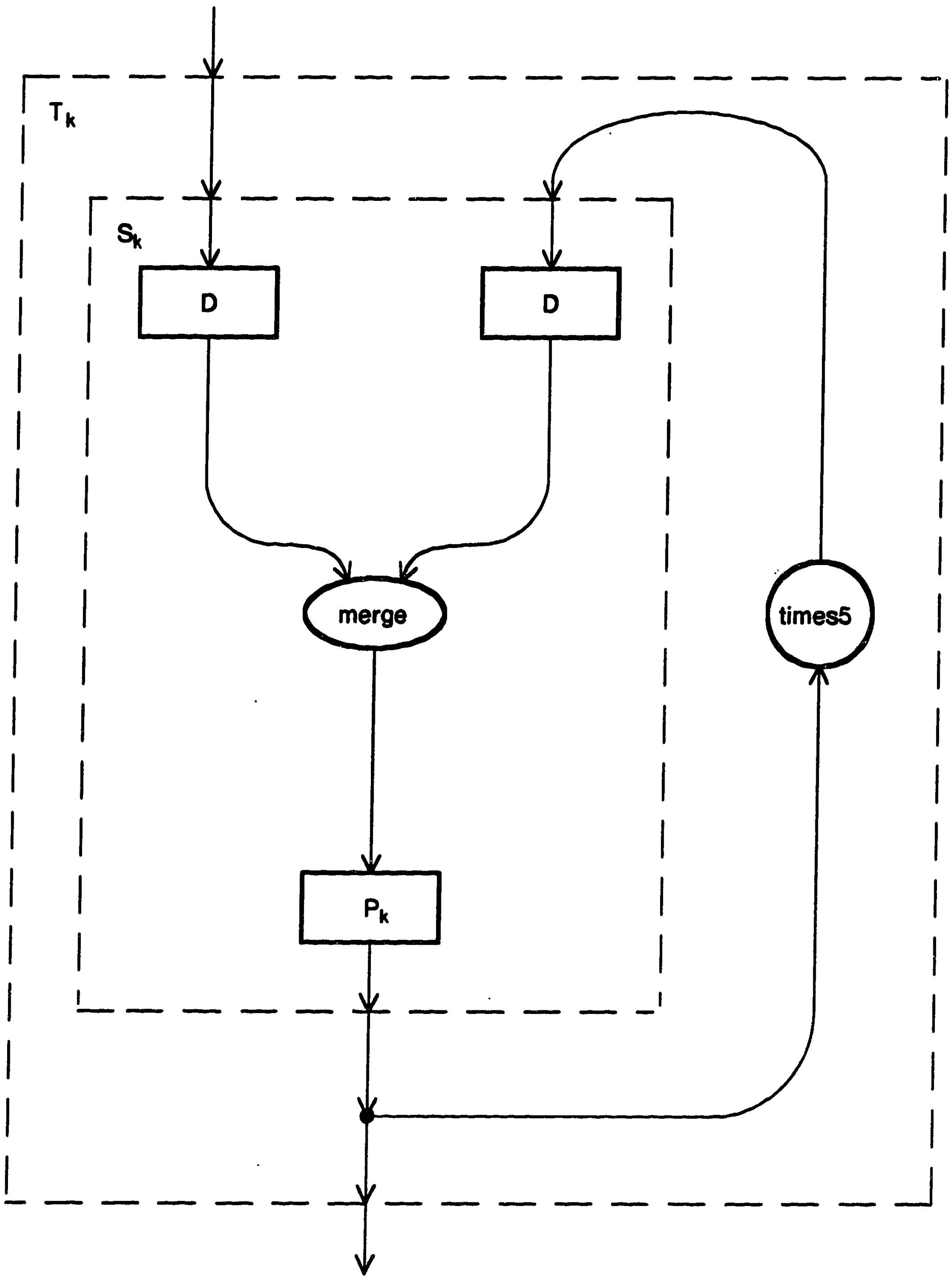


operator, to the rightmost input port of S_k. In Kahn's language, this interconnection may be specified as:

```
network T_k in (IN) out (OUT) internal (X)
    X ← times5(OUT)
    OUT ← S_k(IN, X)
end T_k
```

If history relations are an adequately detailed model of non-determinate dataflow computation, then networks T_1 and T_2 should have the same history relation as all their corresponding components do. However, this is not the case, as can be seen by simulating the execution of these two networks on the input history consisting of the single input one. In Figure 3.4 we have "removed the cover" from S_k and have reproduced T_k with the internal components of S_k clearly shown. Let us now examine all possible computations of first T_1 and

Figure 3.4. T_k for $k \in \{1, 2\}$ (exploded)



then T_2 with input history 1.[1]

Suppose T_1 receives the single input value one. The value one passes through the leftmost D process which then produces the output stream 1 1. At this point the **merge** has no "choice" as to its next action. It must remove the first value from its leftmost input port and produce it at its first output. That value, a one, then passes through P_1 to become the first output of T_1. The value is multiplied by five by the **times5** operator, causing the value five to appear at the rightmost input port of S_1. The value five is duplicated by the rightmost D process producing the output stream 5 5. Now the **merge** does have a choice: it may either accept its second leftmost input, the value one, or its first rightmost input, the value five, to become its second output. The **merge** may have already taken the one before the five arrives, but nevertheless a possibility for choice does exist. P_1 will pass its second input, whichever of one or five was chosen, through as the second output value of T_1. There may be further firings of the **merge** and **times5** operators, but these cannot affect the output of T_1. Therefore, applied to the input stream 1, T_1 has two possible output streams, 1 1 and 1 5.

$T_1(1) = \{1\ 1, 1\ 5\}$

Now suppose T_2 receives the input sequence 1. The value one enters the leftmost D process which produces the output stream 1 1. As in the T_1 case, the **merge** has no choice but to remove the first one and produce it as its own first output and as the first input of P_2. But here, we encounter a difference between the two systems. P_2 will not produce any output until its has received two input values, while P_1 could produce output upon immediate receipt of input. Eventually, the **merge** must remove the second value, another one, from its leftmost input port

1. Readers interested in a more meaningful investment in this simulation are advised to dig deep into their pockets to obtain thirteen cents, three pennies and two nickels, to use as data values before proceeding.

and make it the second input to P_2. P_2 will now produce output, in particular 1 1, which will become the eventual output of T_2. Again, computation may proceed internally within T_2, but no further output will be produced. Unlike T_1, T_2 has but one possible output sequence, 1 1. Therefore:

$$T_2(1) = \{ 1\ 1 \}$$

By taking two networks with the same history relation and showing that they are not substitutable as components of a larger network, we have demonstrated that history relations incompletely specify the behavior of non-determinate networks. Although we have derived the history relations of our example networks using the somewhat informal operational semantics of token-pushing discussed in Chapter 2, more formal derivations could be made.

3.4 Scenarios: An Informal Introduction

In this section we introduce *scenario sets*, our own model of non-determinate dataflow computation, to overcome the shortcomings of history relations. A history relation is a set of pairs of input and output history tuples. Each pair represents one possible, potentially eternal, computation of a modeled process. A single *scenario* is one of these pairs augmented with a partial ordering on the individual data items of the history tuples showing causality. As such the scenario has three components: an input history vector, an output history vector, and a *causality* relation on the individual tokens of the input and output histories. Two tokens are causally related if the event of producing one *must* precede the event of producing the other. Consequently, the tokens of one input (or one output) history are totally ordered under the causality relation, since early tokens of a stream *must* be produced before later ones. If an input token contributes, through a series of indivisible primitive actions, to the production of an output token, it is causally related to that output token. Note that, when a system is viewed as a "black box" with unconnected input and output ports, no input token can cause an input token on another port, no output token can cause an output token on another port, and no output token can cause an input token. Consequently, the only possible causality relations that can exist are those between elements of a single history and those from an input token to an output token. In some ways the ordering of the scenario resembles the "combined order" of Hewitt and Baker [18]. A system, itself, is represented by the *scenario set* of all its possible behaviors.

A scenario can be drawn graphically with each input or output history shown as a column of values and with arrows drawn between causally related values. Because every history is totally ordered causally, arrows must be drawn the length of all columns. The only other arrows will be from input values to the output values they cause. The requirement that the causality order be partial may be restated as the requirement that arrows do not form directed cycles; that is, that no value "causes" itself. There are three possible behaviors of the **merge** operator when receiving the input history tuple <5 6, 7>. The corresponding three scenarios are illustrated in Figure 3.5. The entire specification of the **merge** by scenarios consists of an infinite set of scenarios with many, perhaps infinitely many, scenarios for each input history tuple.

Before describing how scenario sets of networks can be derived from those of their components, let use use our intuition concerning causality to construct scenario sets for systems S_1 and S_2 (Figure 3.2). Because at most one value from either input port can contribute to the

---

Figure 3.5. Scenarios for merge(5 6, 7)

$(5\,6, 7) \rightarrow 5\,6\,7$

$$\begin{tikzcd}
5 \arrow[d] \arrow[rr, bend left] & 7 \arrow[rdd] & 5 \arrow[d] \\
6 \arrow[rr] & & 6 \arrow[d] \\
 & & 7
\end{tikzcd}$$

$(5\,6, 7) \rightarrow 5\,7\,6$

$$\begin{tikzcd}
5 \arrow[d] \arrow[rr, bend left] & 7 \arrow[rd] & 5 \arrow[d] \\
6 \arrow[rrd, bend right] & & 7 \arrow[d] \\
 & & 6
\end{tikzcd}$$

$(5\,6, 7) \rightarrow 7\,5\,6$

$$\begin{tikzcd}
5 \arrow[d] \arrow[rrd, bend right] & 7 \arrow[r] & 7 \arrow[d] \\
6 \arrow[rrd, bend right] & & 5 \arrow[d] \\
 & & 6
\end{tikzcd}$$

output of S_k, we will only consider input histories of length zero or one. In Figure 3.6, the five easy cases are illustrated. The scenarios for these cases are common to both S_1 and S_2. The most trivial case is that in which no input is received, and, consequently, no output is produced. In this case, the causality relation is, of necessity, empty. The remaining four trivial cases (shown as scenario "schemas") occur when only one input value contributes to, *i. e.* causes, both output values of S_k. In each of these cases, a D module accepts this input value and transmits two copies of it to the **merge** which in turn transmits the copies to P_k (as its first *two* input values) which in turn transmits them to the output port of S_k. Any other input value to S_k could cause, at best, the third output of the **merge** and consequently can play no role in the generation of S_k's output.

The non-trivial cases are the interesting ones, for it is in these cases that scenarios reveal the difference between S_1 and S_2. Suppose both input values contribute to the output value of S_k. We have drawn the scenario schemas for this case in Figure 3.7 for S_1 and in Figure 3.8 for S_2. For now assume that the leftmost input of S_k "becomes" its first output. (This is the leftmost scenario in both figures.) The symmetric case may be analyzed similarly. Whether in S_1 or in S_2,

Figure 3.6. Scenarios in Both S_1 and S_2

$(\Lambda, \Lambda) \rightarrow \Lambda$

$$\begin{tikzcd} \Lambda & \Lambda & \Lambda \end{tikzcd}$$

$(x, \Lambda) \rightarrow x\,x$

$$\begin{tikzcd} x \arrow[rr, bend left] & \Lambda & x \arrow[d] \\ & & x \end{tikzcd}$$

$(\Lambda, y) \rightarrow y\,y$

$$\begin{tikzcd} \Lambda & y \arrow[r] & y \arrow[d] \\ & & y \end{tikzcd}$$

$(x, y) \rightarrow x\,x$

$$\begin{tikzcd} x \arrow[rr, bend left] & y & x \arrow[d] \\ & & x \end{tikzcd}$$

$(x, y) \rightarrow y\,y$

$$\begin{tikzcd} x & y \arrow[r] & y \arrow[d] \\ & & y \end{tikzcd}$$

Figure 3.7. Scenarios for S_1 Only

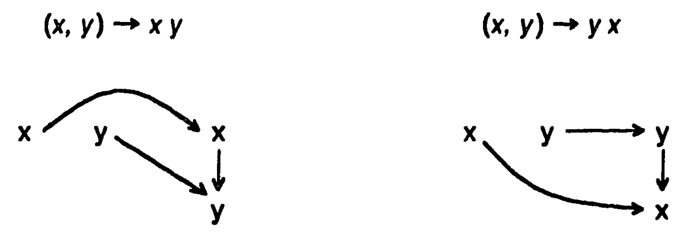
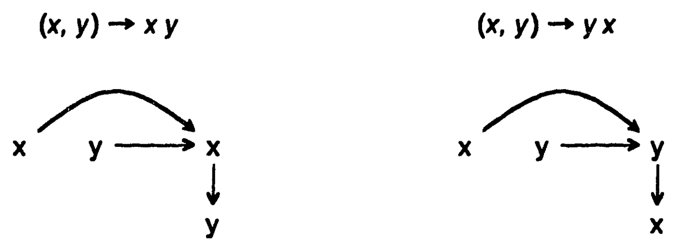


Figure 3.8. Scenarios for S_2 Only



the merge will receive a stream x x on its left and a stream y y on its right and will transmit a stream beginning with x y to P_k. P_k will, of course, produce x y as the output of S_k. However, only P_1's first input is required to produce the first output. Consequently, only the leftmost input of S_1 causes its first output. But with P_2 both input values, including that resulting from the rightmost input of S_2, are required for the production of the first output. Consequently, both input values of S_2 cause its first output value. So, although the history relation specifications of these computations are identical, their scenarios reveal the subtle difference in behavior that becomes apparent when the systems are used in the construction of T_1 and T_2.

Given the scenario sets for the components of a network, we wish to derive scenario sets for the network. The method for performing this derivation is the *scenario composition rule*. In the remainder of this chapter, we will present, somewhat informally, the scenario composition rule for two component networks and illustrate it by deriving scenarios for networks T_1 and T_2. In

Chapter 5, we will develop a more formal algebra of scenario set composition and will "prove" the faithfulness of the scenario composition rule to the operational semantics of dataflow computation.

Given the scenario sets of the components of a network of two systems, the scenario set of the interconnection is obtained by a four-step procedure. First, enumerate the Cartesian product of the scenario sets of the two systems. The remaining three steps will filter and refine this set of scenario pairs.

Second, discard all scenario pairs whose data values do not agree on ports that are linked to each other, and "merge" each pair into one scenario, identifying the columns of linked ports. Obviously, two scenarios can co-exist as part of the same computation only if they specify the same histories for connected ports. These scenario pairs we call *value-consistent*. They correspond to the fixed points of determinate dataflow computation.

Third, if a directed cycle exists within a merged value-consistent pair, discard the pair. If there is a directed cycle, we have a "spontaneous" value causing itself and, furthermore, our causality relation is not a partial order. The remaining scenario pairs we call *causality-consistent*. They correspond to the least fixed points of determinate dataflow computation.

Fourth, and finally, from each of the causality-consistent pairs remove the columns of linked ports. If a chain of arrows is broken by the removal of a column, draw a new "direct" arrow to replace that chain; that is, extend the partial order across the identified columns. The linked ports are an internal feature of the interconnection and no reference should be made to them in an external specification of the network's behavior. However, if an input to the network causes an output by acting through one of the internally connected ports, this causal relationship must be shown in the network's scenario by extension of the partial order.

Although we derived the scenario sets for S_1 and S_2 through "common-sense" causal reasoning, the same result could be obtained by use of the scenario composition rule, and the skeptical reader is invited to verify this claim. We will illustrate the composition rule by applying it to the systems T_1 and T_2 with special reference to the input history of the single value one which resulted in the history relation merge anomaly.

T_k is an interconnection of two systems: S_k and **times5**. For both T_1 and T_2, there are only two value-consistent scenario pairs for the input history 1. In Figure 3.9, the value-consistent pairs for T_1 are illustrated both with and without the merging of the connected ports. Both of the merged value-consistent pairs are causality-consistent, and each can be made into a scenario by the removal of the internal column. Thus the scenario composition rule is consistent with our earlier observation that T_1 could produce either 1 1 or 1 5 in response to the input history 1.

Figure 3.9. Value-consistent Pairs for $T_1(1)$

T_1: $1 \rightarrow 1\ 1$
S_1: $(1, 5\ 5) \rightarrow 1\ 1$
times5: $1\ 1 \rightarrow 5\ 5$

T_1: $1 \rightarrow 1\ 5$
S_1: $(1, 5\ 25) \rightarrow 1\ 5$
times5: $1\ 5 \rightarrow 5\ 25$

without port merging

$$\begin{tikzcd} 1 \arrow[rr, bend left] & 5 \arrow[d] & 1 \arrow[d] \\ & 5 & 1 \end{tikzcd} \qquad \begin{tikzcd} 1 \arrow[r] \arrow[d] & 5 \arrow[d] \\ 1 \arrow[r] & 5 \end{tikzcd} \qquad \begin{tikzcd} 1 \arrow[rr, bend left] & 5 \arrow[d] \arrow[dr] & 1 \arrow[d] \\ & 25 & 5 \end{tikzcd} \qquad \begin{tikzcd} 1 \arrow[r] \arrow[d] & 5 \arrow[d] \\ 5 \arrow[r] & 25 \end{tikzcd}$$

with port merging

$$\begin{tikzcd} 1 \arrow[rr, bend left] & 5 \arrow[d] & 1 \arrow[l] \arrow[d] \\ & 5 & 1 \arrow[l] \end{tikzcd} \qquad \begin{tikzcd} 1 \arrow[rr, bend left] & 5 \arrow[d] \arrow[dr] & 1 \arrow[l] \arrow[d] \\ & 25 & 5 \arrow[l] \end{tikzcd}$$

In Figure 3.10, the value–consistent pairs for T_2 are illustrated both with and without the merging of the connected ports. However, note that now only one of the merged value–consistent pairs is causality-consistent. In the other we see a cycle (between the one in S_k's output and the five in S_k's rightmost input). The scenario composition rule correctly reflects the fact that 1 1 is the only response of T_2 to the input history 1.

In the beginning of this chapter, we mentioned faithfulness, abstraction, and simplicity as three goals of a semantic theory. With scenarios, faithfulness to the underlying operational model results from the manner in which the scenario causality relation reflects the causality implied by the firing sequences of the operational model. In Chapter 5 we shall make a more formal assertion of this claim. We have shown that no semantic theory for non–determinate dataflow computation can be as abstract as history relations, the pure input/output behavioral specification for this class of computation. However, scenarios are not so far from this

---

Figure 3.10. Value–consistent Pairs for $T_2(1)$

T_2: 1 → 1 1
S_2: (1, 5 5) → 1 1
times5: 1 1 → 5 5

T_2: 1 → 1 5
S_2: (1, 5 25) → 1 5
times5: 1 5 → 5 25

without port merging

$$\begin{tikzcd}
1 \arrow[rr, bend left] & 5 \arrow[d] & 1 \arrow[d] \\
 & 5 & 1
\end{tikzcd}
\qquad
\begin{tikzcd}
1 \arrow[r] \arrow[d] & 5 \arrow[d] \\
1 \arrow[r] & 5
\end{tikzcd}
\qquad
\begin{tikzcd}
1 \arrow[rr, bend left] & 5 \arrow[r] \arrow[d] & 1 \arrow[d] \\
 & 25 & 5
\end{tikzcd}
\qquad
\begin{tikzcd}
1 \arrow[r] \arrow[d] & 5 \arrow[d] \\
5 \arrow[r] & 25
\end{tikzcd}$$

with port merging

$$\begin{tikzcd}
1 \arrow[rr, bend left] & 5 \arrow[d] & 1 \arrow[l] \arrow[d] \\
 & 5 & 1 \arrow[l]
\end{tikzcd}
\qquad
\begin{tikzcd}
1 \arrow[rr, bend left] & 5 \arrow[d] \arrow[r, bend right] & 1 \arrow[l] \arrow[d] \\
 & 25 & 5 \arrow[l]
\end{tikzcd}$$

unattainable goal. They are just history relations augmented with a notion of causality. The most striking advantage of the scenario model when compared to other proposed models of non-determinate dataflow computation must be the simplicity of its composition rule. Scenario composition does not require the solution of complicated fixed point equations over complicated mathematical domains. Scenarios can be composed with no more complicated mathematical machinery than the knowledge of partial orders.

4. The Dataflow Graph Algebra

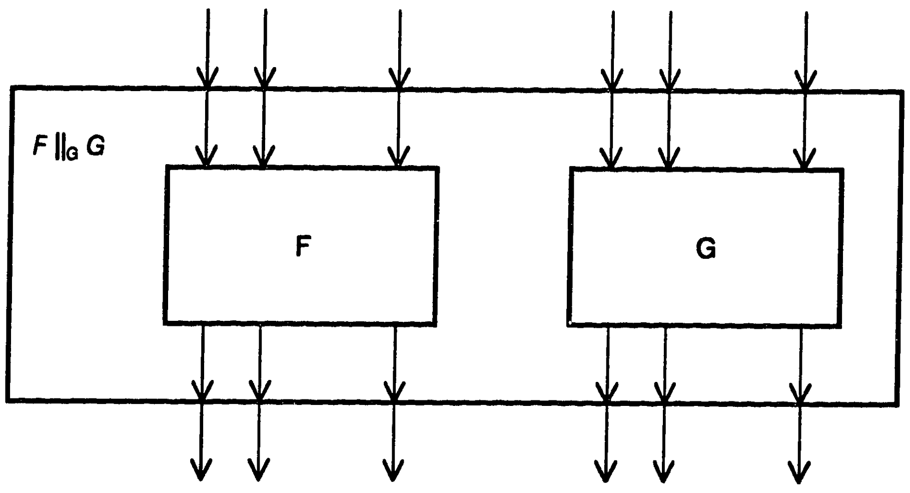
In this brief chapter we define an algebra for dataflow graphs. This algebra may be considered as the syntax for an "assembly" language for specifying dataflow graphs, but certainly not as a serious programming language (as will soon be apparent). We present this language to provide a base language for the more complete and formal definition of the scenario model of dataflow computation which appears in the next chapter.

Our graph algebra will resemble the initial flowgraph algebra of Milne and Milner [30], but there are, of course, some important differences. While the ports of flowgraphs are bi-directional, the ports of dataflow graphs are either input or output ports. Also, while several ports of a flowgraph may be connected together, the only port connection we allow is of a single output port to a single input port. For this reason we do not need the "restriction" operator.

We assume as generators or base objects of the algebra a set A of dataflow actors. The actual members of A will not be crucial to our discussion. However, a "useful" A would contain the necessary operators to implement a high-level dataflow programming language such as Id [5] or VAL [2, 29]. The A we will use will be composed of all determinate processes written in Kahn's [22] language plus the non-determinate **merge** operator. Associated with each operator o of A are two disjoint sets, a set $Inport(o)$ of input port labels and a set $Outport(o)$ of output labels. For convenience, $Port(o)$ denotes $Inport(o) \cup Outport(o)$. These sets will play an important role in determining the applicability of the combining operators of the dataflow graph algebra to particular graphs.

The dataflow graph algebra has only three operators: graph union, port relabeling, and port connection. These three are sufficient to describe any dataflow graph. The graph union operator $\|_G$ is binary. Graph union is illustrated in graphic detail in Figure 4.1. Graph union is merely the association of two disjoint graphs with no implied interconnection of ports. The labels of the

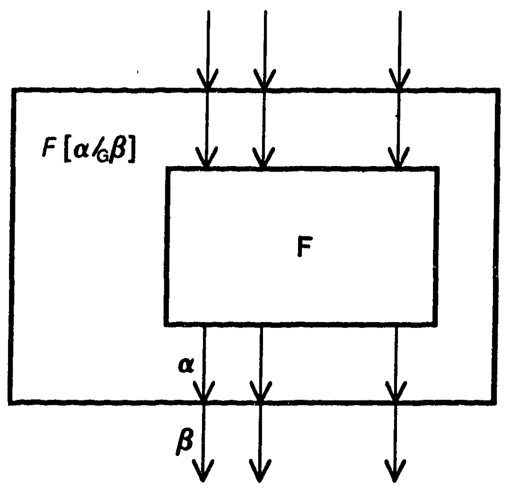
Figure 4.1. Graph union $\|_G$



constituent graphs are unchanged by graph union. In order to avoid problems of duplicate port labels, the graph union operator is restricted to graphs G and H with disjoint port labels, *i. e.* $G \|_G H$ is defined only when $Port(G) \cap Port(H) = \emptyset$. Consequently, the input ports, $Inport(G \|_G H)$, of the joined graphs are $Inport(G) \cup Inport(H)$, and the output ports, $Outport(G \|_G H)$, are $Outport(G) \cup Outport(H)$.

Because graph union is limited to graphs with disjoint labels, it will often be necessary to relabel ports before graphs are joined. The port relabeling operator $[\bullet/_G\bullet]$ is used for this purpose. Actually, $[\bullet/_G\bullet]$ represents an operator schema: for each pair $\langle\alpha, \beta\rangle$ of port labels, there is a relabeling operator $[\alpha/_G\beta]$ which renames port α to be port β. Port relabeling is illustrated in Figure 4.2. Port relabeling cannot be used to introduce duplicate port labels. For this reason, $G\ [\alpha/_G\beta]$ is defined only when β is not a port label of G. For convenience, we also require that α must be a port label of G. If α is an input port of G, then $Inport(G\ [\alpha/_G\beta]) = (Inport(G) - \{\alpha\}) \cup \{\beta\}$, and $Outport(G\ [\alpha/_G\beta]) = Outport(G)$. Similarly, if α is an output port of G, then $Outport(G\ [\alpha/_G\beta]) = (Outport(G) - \{\alpha\}) \cup \{\beta\}$, and $Inport(G\ [\alpha/_G\beta]) = Inport(G)$.

Figure 4.2. Port relabeling $[\bullet /_G \bullet]$

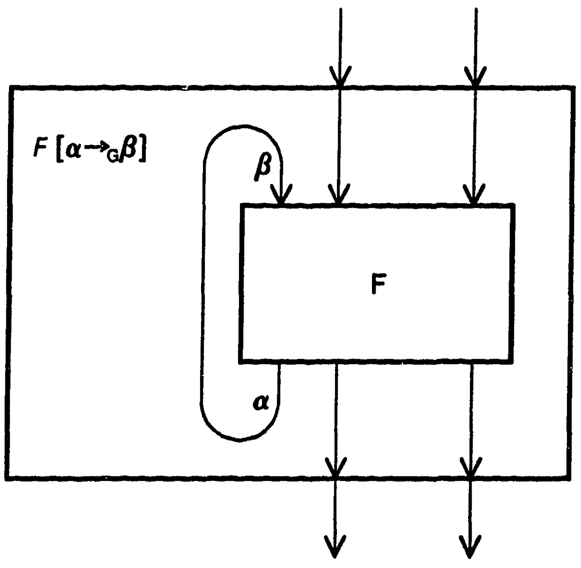


The third, and final, dataflow graph operation is port connection. Port connection $[\bullet \rightarrow_G \bullet]$, like port relabeling, is a unary operator and is also an operator schema. The port connection operator $[\alpha \rightarrow_G \beta]$ can be applied only to graphs G with an output port α and an input port β. The graph $G\ [\alpha \rightarrow_G \beta]$, illustrated in Figure 4.3, is formed by connecting output port α to input port β. The connected ports become internal[1] to the new graph and may never play any role in future graph interconnections. Thus $Inport(G\ [\alpha \rightarrow_G \beta]) = Inport(G) - \{\beta\}$, and $Outport(G\ [\alpha \rightarrow_G \beta]) = Outport(G) - \{\alpha\}$. The connection of an output port to two or more input ports is accomplished by the explicit use of determinate fan-out operators.

Obviously it is quite tedious to build up a graph with these operators. In Figure 4.4 a simple three-operator dataflow graph for computing the dot product of two two-element vectors, (x_0, y_0) and (x_1, y_1), is shown and described in our algebraic notation. We assume that each of the three

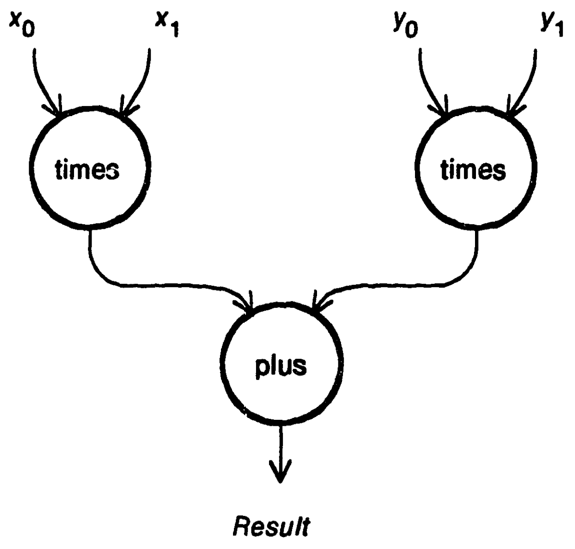
1. "Restricted" in the Milne-Milner [30] terminology.

Figure 4.3. Port connection [$\bullet \rightarrow_G \bullet$]



dataflow operators has input ports labeled In_1 and In_2 and an output port labeled Out_1 and that each input port of the assembled graph should be labeled by the appropriate variable name and the graph output port should be labeled *Result*. This rather straightforward interconnection requires no less than thirteen applications of our operators. However, since we are only concerned with developing a basis for the formalism of the succeeding chapter, the wordiness of our notation is of little concern.

Figure 4.4. Two-Element Dot Product



$$\begin{array}{l}
(\ \text{times}\ [In_1 /_G\ x_0]\ [In_2 /_G\ x_1]\ [Out_1 /_G\ Op_1Out_1]\ \|_G \\
\ \ \text{times}\ [In_1 /_G\ y_0]\ [In_2 /_G\ y_1]\ [Out_1 /_G\ Op_2Out_1]\ \|_G \\
\ \ \text{plus}\ [In_1 /_G\ Op_3In_1]\ [In_2 /_G\ Op_3In_2]\ [Out_1 /_G\ Result]\ ) \\
\ \ \ [Op_1Out_1 \rightarrow_G Op_3In_1]\ [Op_2Out_1 \rightarrow_G Op_3In_2]
\end{array}$$

5. The Scenario Set Algebra

In this chapter we will construct a formal model for the scenario model of computation introduced informally in Chapter 3. This model consists of two parts: (1), a representation of scenarios, and hence of scenario sets, and (2), a formal definition of the scenario composition rules. We will also use this model to assert the faithfulness of the scenario model of computation to token-pushing, the operational model of dataflow computation presented in Chapter 2.

Like graphs, scenario sets have labeled input and output ports. Given disjoint sets *In* of input port labels and *Out* of output port labels, a scenario is a triple $\langle E, V, C\rangle$ where:

> E is the set of scenario *events*. Each element of E is an ordered pair consisting of a port label and an integer greater than zero. If $\langle \alpha, n\rangle$ is in E, it represents the event of receiving (or producing) the n'th value at input (or output) port α.
>
> V is the scenario *valuation* function mapping the events of E into the set of elementary dataflow token values. Consequently, $V(\langle \alpha, n\rangle)$ is the value of the n'th token received (or produced) at port α.
>
> C is the scenario *causality* relation on events. If $\langle \alpha, m\rangle\ C\ \langle \beta, n\rangle$ then the event of receiving (or producing) the m'th element at port α contributes to the event of receiving (or producing) the n'th element at port β.

Furthermore, the three components of a scenario must obey the same scenario restrictions stated somewhat informally in Chapter 3. In particular:

> If $\langle \alpha, n\rangle$ is an element of E and m is an integer greater than zero and less than n, then $\langle \alpha, m\rangle$ is also an element of E. This restriction prevents port histories from having "holes."
>
> If $\langle \alpha, m\rangle$ and $\langle \alpha, n\rangle$ are elements of E and m is less than or equal to n, then $\langle \alpha, m\rangle\ C\ \langle \alpha, n\rangle$. That is, early elements of a history must precede later ones.
>
> If α and β are distinct port labels and $\langle \alpha, m\rangle\ C\ \langle \beta, n\rangle$ then α is an input port label and β is an output port label. Therefore, the only inter-port causalities are from input events to output events.
>
> C is a partial order: two events do not cause each other.

In Figure 5.1, the scenario for the **merge** with input history tuple <5 6, 7> leading to the production of the output sequence 5 7 6 is illustrated. Assuming that the **merge** has input port labels In_1 and In_2 and output port label Out_1, the scenario of Figure 5.1 is represented by the triple $\langle E, V, C \rangle$ where:

$$E = \{\langle In_1, 1\rangle, \langle In_1, 2\rangle, \langle In_2, 1\rangle, \langle Out_1, 1\rangle, \langle Out_1, 2\rangle, \langle Out_1, 3\rangle\}$$

$V(\langle In_1, 1\rangle) = 5$ $\quad V(\langle In_2, 1\rangle) = 7$ $\quad V(\langle Out_1, 1\rangle) = 5$
$V(\langle In_1, 2\rangle) = 6$ $\quad V(\langle Out_1, 2\rangle) = 7$
$V(\langle Out_1, 3\rangle) = 6$

C is represented in Figure 5.2 as a Hasse diagram, the usual pictorial representation of a partial order. When this relation is enumerated as a subset of $E \times E$, it contains sixteen ordered pairs.

Figure 5.1. One Scenario for merge(5 6, 7)

$$\begin{tikzcd}
5 \arrow[d] \arrow[rr, bend left] & 7 \arrow[dr] & 5 \arrow[d] \\
6 \arrow[drr, bend right] & & 7 \arrow[d] \\
& & 6
\end{tikzcd}$$

Figure 5.2. Hasse Diagram for a merge Causality Relation

$$\begin{tikzcd}
\langle In_1, 1\rangle \arrow[d] \arrow[rr, bend left] & \langle In_2, 1\rangle \arrow[dr] & \langle Out_1, 1\rangle \arrow[d] \\
\langle In_1, 2\rangle \arrow[drr] & & \langle Out_1, 2\rangle \arrow[d] \\
& & \langle Out_1, 3\rangle
\end{tikzcd}$$

5.1 Operators of the Scenario Set Algebra

For consistency with our dataflow graph algebra, *Inport*(S) denotes the set of input port labels and *Outport*(S) denotes the set of output port labels of a scenario set S. Like the dataflow graph algebra, the scenario algebra has three operations, $\|_S$, $[\bullet /_S \bullet]$, and $[\bullet \rightarrow_S \bullet]$. Applications of the scenario operators must satisfy the same constraints on input and output port labels that the corresponding applications of dataflow graph operators must satisfy.

Scenario set union is little more than taking the Cartesian product of two scenario sets. Like $\|_G$, if S and S' are scenario sets, then $S \|_S S'$ is defined only if S and S' have disjoint port labels. For each scenario $\langle E, V, C\rangle$ of S and $\langle E', V', C'\rangle$ of S', $\langle E \cup E', V \cup V', C \cup C'\rangle$ is a scenario of $S \|_S S'$. The set unions used to define this scenario reflect the underlying independence of two paired, but unconnected, systems. The events of the new scenario are the events of either of the two component scenarios. It may seem unusual to take unions of functions and relations, but because the underlying event domains are disjoint it really works. The union of the valuation functions and the causality relations may be expressed more conventionally as:

$$\begin{aligned}(V \cup V')(\langle\alpha, n\rangle) &= V(\langle\alpha, n\rangle), \text{ if } \langle\alpha, n\rangle \in E\\ &= V'(\langle\alpha, n\rangle), \text{ if } \langle\alpha, n\rangle \in E'\end{aligned}$$

$$\begin{aligned}\langle\alpha, n\rangle\,(C \cup C')\,\langle\beta, m\rangle &= \langle\alpha, n\rangle\, C\, \langle\beta, m\rangle, \text{ if } \langle\alpha, n\rangle, \langle\beta, m\rangle \in E\\ &= \langle\alpha, n\rangle\, C'\, \langle\beta, m\rangle, \text{ if } \langle\alpha, n\rangle, \langle\beta, m\rangle \in E'\\ &= \textbf{false}, \text{ otherwise}\end{aligned}$$

It is pretty easy to see that $C \cup C'$ satisfies the restrictions on causality relations stated at the beginning of this chapter and thus to conclude that $\langle E \cup E', V \cup V', C \cup C'\rangle$ is indeed a scenario.

Scenario set port relabeling is an even simpler operation. If S is a scenario set with port label α and without port label β, then $S\,[\alpha /_S \beta]$ is defined. For every scenario $\langle E, V, C\rangle$ of S there is a scenario $\langle E', V', C'\rangle$ of $S\,[\alpha /_S \beta]$ formed from $\langle E, V, C\rangle$ by replacing all occurrences of α with β. By "replace all occurrences" we mean:

$$E' = \{\langle\gamma, m\rangle \mid \langle\gamma, m\rangle \in E \text{ and } \gamma \neq \alpha\} \cup \{\langle\beta, m\rangle \mid \langle\alpha, m\rangle \in E\}$$

$$V'(\langle\gamma, m\rangle) = V(\langle\gamma, m\rangle), \text{ if } \langle\gamma, m\rangle \in E \text{ and } \gamma \neq \alpha$$
$$V'(\langle\beta, m\rangle) = V(\langle\alpha, m\rangle), \text{ if } \langle\alpha, m\rangle \in E$$

$$\langle\gamma, m\rangle\, C'\, \langle\delta, n\rangle = \langle\gamma, m\rangle\, C\, \langle\delta, n\rangle, \text{ if } \gamma \neq \alpha \text{ and } \delta \neq \alpha$$
$$\langle\gamma, m\rangle\, C'\, \langle\beta, n\rangle = \langle\gamma, m\rangle\, C\, \langle\alpha, n\rangle, \text{ if } \gamma \neq \alpha$$
$$\langle\beta, m\rangle\, C'\, \langle\delta, n\rangle = \langle\alpha, m\rangle\, C\, \langle\delta, n\rangle, \text{ if } \delta \neq \alpha$$
$$\langle\beta, m\rangle\, C'\, \langle\beta, n\rangle = \langle\alpha, m\rangle\, C\, \langle\alpha, n\rangle$$

Once again it is easy to show that *C'* satisfies the requirements of a causality relation.

The final scenario set operator is port connection, and it is by far the most difficult to formalize. Let *S* be a scenario set with output port α and input port β. We derive the scenario set $S\ [\alpha \rightarrow_S \beta]$ by the same filtering process described informally in Chapter 3.

Definition: A scenario $\langle E, V, C\rangle$ is α–β *value-consistent* if and only if:
(1), $\langle\alpha, m\rangle \in E$ if and only if $\langle\beta, m\rangle \in E$, and
(2), $V(\langle\alpha, m\rangle) = V(\langle\beta, m\rangle)$ for all $\langle\alpha, m\rangle \in E$.

Value-consistency characterizes those scenarios in which the histories at output port α and input port β are identical.

Definition: A scenario $\langle E, V, C\rangle$ is α–β *causality-consistent* if and only if:
(1), $\langle E, V, C\rangle$ is α–β value-consistent, and
(2), for all $\langle\beta, m\rangle \in E$, it is *not* the case that $\langle\beta, m\rangle\, C\, \langle\alpha, m\rangle$.

Satisfaction of causality-consistency ensures that, when output port α is connected to input port β, no value passing through the connection causes itself. Consequently, all α–β causality-consistent scenarios represent feasible computations for systems in which the α–β connection has been made. This port connection, once made, will, of course, introduce extra causality relationships.

Definition: For an α–β causality-consistent scenario $\langle E, V, C\rangle$, the α–β *connection relation* on *E* is the relation C^* such that $\langle\gamma, m\rangle\, C^*\, \langle\delta, n\rangle$ if and only if:
(1), $\langle\gamma, m\rangle\, C\, \langle\delta, n\rangle$, or
(2), there exists $\langle\alpha, p\rangle$ in *E* such that $\langle\gamma, m\rangle\, C\, \langle\alpha, p\rangle$ and $\langle\beta, p\rangle\, C\, \langle\delta, n\rangle$.

The suitability of the α–β connection relation for constructing scenarios is established by the following lemma:

> **Lemma**: The α–β connection relation of an α–β causality–consistent scenario is a partial order.
>
> **Proof**: Let C^* be the α–β connection relation of an α–β causality–consistent scenario $\langle E, V, C\rangle$. For C^* to be a partial order, it must be reflexive, antisymmetric, and transitive.
>
> C^* is trivially reflexive as it is an extension of the reflexive relation C.
>
> We prove antisymmetry by contradiction. Suppose $\langle\gamma, m\rangle$ and $\langle\delta, n\rangle$ are *distinct* elements of E such that $\langle\gamma, m\rangle\ C^*\ \langle\delta, n\rangle$ and $\langle\delta, n\rangle\ C^*\ \langle\gamma, m\rangle$. From the definition of C^*, we know there are four cases which must be considered. The four cases are the product of the following two: (1), either $\langle\gamma, m\rangle\ C\ \langle\delta, n\rangle$ or there is an element $\langle\alpha, p\rangle$ in E such that $\langle\gamma, m\rangle\ C\ \langle\alpha, p\rangle$ and $\langle\beta, p\rangle\ C\ \langle\delta, n\rangle$; and (2), either $\langle\delta, n\rangle\ C\ \langle\gamma, m\rangle$ or there is an element $\langle\alpha, q\rangle$ in E such that $\langle\delta, n\rangle\ C\ \langle\alpha, q\rangle$ and $\langle\beta, q\rangle\ C\ \langle\gamma, m\rangle$.
>
> The first case is trivial. Because C itself is antisymmetric, it cannot be that $\langle\gamma, m\rangle\ C\ \langle\delta, n\rangle$ and $\langle\delta, n\rangle\ C\ \langle\gamma, m\rangle$.
>
> In only two of the remaining three cases is either of $\langle\gamma, m\rangle$ or $\langle\delta, n\rangle$ directly related through C. Both cases may be argued similarly. Suppose $\langle\gamma, m\rangle\ C\ \langle\delta, n\rangle$ and there is an $\langle\alpha, q\rangle$ in E such that $\langle\delta, n\rangle\ C\ \langle\alpha, q\rangle$ and $\langle\beta, q\rangle\ C\ \langle\gamma, m\rangle$. This situation is illustrated in Figure 5.3. As $\langle\beta, q\rangle\ C\ \langle\gamma, m\rangle$, $\langle\gamma, m\rangle\ C\ \langle\delta, n\rangle$, and $\langle\delta, n\rangle\ C\ \langle\alpha, q\rangle$, using the transitivity of C we may conclude that $\langle\beta, q\rangle\ C\ \langle\alpha, q\rangle$, a contradiction of the second condition of the definition of α–β causality–consistency.
>
> In the remaining case there exist $\langle\alpha, p\rangle$ and $\langle\alpha, q\rangle$ in E such that $\langle\gamma, m\rangle\ C\ \langle\alpha, p\rangle$ and $\langle\beta, p\rangle\ C\ \langle\delta, n\rangle$, and $\langle\delta, n\rangle\ C\ \langle\alpha, q\rangle$ and $\langle\beta, q\rangle\ C\ \langle\gamma, m\rangle$. Assume q is at least as large as p, as illustrated in Figure 5.4. Consequently, as C is a causality relation, $\langle\beta, p\rangle\ C\ \langle\beta, q\rangle$. Therefore, as $\langle\beta, p\rangle\ C\ \langle\beta, q\rangle$, $\langle\beta, q\rangle\ C\ \langle\gamma, m\rangle$, and $\langle\gamma, m\rangle\ C\ \langle\alpha, p\rangle$ it follows from the transitivity of C that $\langle\beta, p\rangle\ C\ \langle\alpha, p\rangle$, a contradiction. The case in which p is at least as large as q leads to a similar contradiction.
>
> As all four cases have resulted in contradiction, we may conclude that C^* is an antisymmetric relation.

Figure 5.3. Antisymmetry: Cases 2 and 3

$$\begin{tikzcd}
\langle \gamma, m \rangle \arrow[rrr] & & & \langle \delta, n \rangle \arrow[dl] \\
& \langle \beta, q \rangle \arrow[ul] & \langle \alpha, q \rangle &
\end{tikzcd}$$

Figure 5.4. Antisymmetry: Case 4

$$\begin{tikzcd}
\langle \gamma, m \rangle \arrow[drr] & & & \langle \delta, n \rangle \arrow[ddl, bend left] \\
& \langle \beta, p \rangle \arrow[urr] \arrow[d] & \langle \alpha, p \rangle \arrow[d] & \\
& \langle \beta, q \rangle \arrow[uul, bend left] & \langle \alpha, q \rangle &
\end{tikzcd}$$

Now only the proof of transitivity remains to establish that C^* is a partial order. Suppose $\langle \gamma, m \rangle$, $\langle \delta, n \rangle$, and $\langle \epsilon, o \rangle$ are elements of E such that $\langle \gamma, m \rangle\ C^*\ \langle \delta, n \rangle$ and $\langle \delta, n \rangle\ C^*\ \langle \epsilon, o \rangle$. Once again, from the definition of C^*, we have four cases, the product of the following two: (1), either $\langle \gamma, m \rangle\ C\ \langle \delta, n \rangle$ or there is an element $\langle \alpha, p \rangle$ in E such that $\langle \gamma, m \rangle\ C\ \langle \alpha, p \rangle$ and $\langle \beta, p \rangle\ C\ \langle \delta, n \rangle$; and (2), either $\langle \delta, n \rangle\ C\ \langle \epsilon, o \rangle$ or there is an element $\langle \alpha, q \rangle$ in E such that $\langle \delta, n \rangle\ C\ \langle \alpha, q \rangle$ and $\langle \beta, q \rangle\ C\ \langle \epsilon, o \rangle$.

If $\langle \gamma, m \rangle\ C\ \langle \delta, n \rangle$ and $\langle \delta, n \rangle\ C\ \langle \epsilon, o \rangle$ then $\langle \gamma, m \rangle\ C\ \langle \epsilon, o \rangle$, and thus $\langle \gamma, m \rangle\ C^*\ \langle \epsilon, o \rangle$, by the transitivity of C.

Once more, we shall look at only one of the two remaining cases in which the events of one pair are directly related through *C*. Suppose ⟨γ, *m*⟩ *C* ⟨δ, *n*⟩ and there is an ⟨α, *q*⟩ in *E* such that ⟨δ, *n*⟩ *C* ⟨α, *q*⟩ and ⟨β, *q*⟩ *C* ⟨ε, *o*⟩. This case is illustrated in Figure 5.5. As ⟨γ, *m*⟩ *C* ⟨δ, *n*⟩ and ⟨δ, *n*⟩ *C* ⟨α, *q*⟩ then, by transitivity, ⟨γ, *m*⟩ *C* ⟨α, *q*⟩ and, from the definition of *C**, ⟨γ, *m*⟩ *C** ⟨ε, *o*⟩. Although not required for our proof, it's worth noting that the restrictions on inter-port causality relations imposed by scenarios imply that δ must be either γ or α.

For the last case of the last property of a partial order, assume that there exist events ⟨α, *p*⟩ and ⟨α, *q*⟩ such that ⟨γ, *m*⟩ *C* ⟨α, *p*⟩ and ⟨β, *p*⟩ *C* ⟨δ, *n*⟩, and ⟨δ, *n*⟩ *C* ⟨α, *q*⟩ and ⟨β, *q*⟩ *C* ⟨ε, *o*⟩. Furthermore, without loss of generality, assume that *q* is at least as great as *p* and, consequently, that ⟨α, *p*⟩ *C* ⟨α, *q*⟩ as shown in Figure 5.6. Then from ⟨γ, *m*⟩ *C* ⟨α, *p*⟩, it follows that ⟨γ, *m*⟩ *C* ⟨α, *q*⟩ and, in turn, that ⟨γ, *m*⟩ *C** ⟨ε, *o*⟩. Again, it's worth noting that the inter-port causality relation restrictions force δ to be either α or β.

Having proven that *C** is a reflexive, antisymmetric, and transitive relation, we have established that *C** is a partial order.

Figure 5.5. Transitivity: Cases 2 and 3

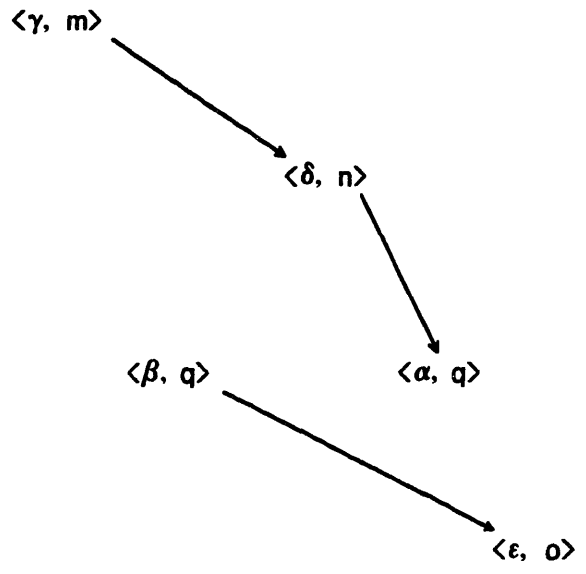


Figure 5.6. Transitivity: Case 4

$$\begin{tikzcd}
\langle\gamma, m\rangle \arrow[drrr] & & & & \\
& \langle\beta, p\rangle \arrow[dr] & & \langle\alpha, p\rangle \arrow[dd] & \\
& & \langle\delta, n\rangle \arrow[dr] & & \\
& \langle\beta, q\rangle \arrow[drrr] & & \langle\alpha, q\rangle & \\
& & & & \langle\varepsilon, o\rangle
\end{tikzcd}$$

The α–β connection relation is not only a partial order but also the smallest partial order which extends C and relates events at output port α to corresponding events at input port β. This fact is important to note as it makes our formal definition of scenario composition consistent with the informal one of Chapter 3.

Given an α–β connection relation C^* of a scenario set S, it is straightforward to construct a corresponding scenario for $S\ [\alpha \rightarrow_S \beta]$ by "removing" the histories for α and β. The following theorem, an easy consequence of the preceding lemma, states more precisely how this is accomplished.

Theorem: If S is a scenario set with output port label α and input port label β, then for every α–β causality-consistent scenario $\langle E, V, C\rangle$ with α–β connection relation C^*, the triple $\langle E', V', C'\rangle$ defined by:

$$E' = \{\langle\gamma, m\rangle \mid \langle\gamma, m\rangle \in E \text{ and } \gamma \neq \alpha \text{ and } \gamma \neq \beta\}$$

$$V'(\langle\gamma, m\rangle) = V(\langle\gamma, m\rangle), \text{ for all } \langle\gamma, m\rangle \in E'$$

$$C' = C^* \cap (E' \times E')$$

is a scenario.

Proof: By the preceding lemma, C^* is a partial order. Once this is established, it is trivial to show that $\langle E', V', C'\rangle$ satisfies the other scenario restrictions stated at the beginning of this chapter. Clearly, C' is a total order on each component history and C' is a partial order on E'. The only remaining problem is asserting that C' contains no inter-port causality relations other than from input events to output events. However, it is easy to see that C^* contains no such relations except those involving α and β, the two ports removed from C^* to form C'.

The set $S\ [\alpha \rightarrow_S \beta]$ contains a scenario, generated according to the above theorem, for each α–β causality relation of S.

5.2 Generators of the Scenario Set Algebra

Now that we have defined the three scenario set composition rules, we direct our attention to the task of defining scenario sets for the elementary dataflow operators. In Chapter 3, the "common-sense" derivation of operator scenarios was effected by following the principle of considering an output event to be causally preceded by the input events required for its generation.

In Chapter 2, Kahn [22] processes were introduced. These are sequential programs that read input values with **get** operations and write output values with **put** operations. We have chosen as the generators of our dataflow graph algebra these determinate Kahn processes plus the non-determinate **merge** operator.

Operationally, the computation of a Kahn process may be considered to be a sequence (total ordering) of *firings* corresponding to the execution of either a **get** or **put** operation. For every such firing sequence there is a scenario formed by removing all inter-port precedences except those from input ports to output ports. Thus, an input event corresponding to a **get** on port α is causally preceded only by the previous **get**'s on α, while an output event corresponding to a **put** on port β is causally preceded by the previous **put**'s on β *plus all* the previous **get**'s on any input channel. Because several firing sequences may yield the same scenario, scenario sets are an abstraction of the firing sequence characterization of processes.

During a dataflow computation, the **merge** repeatedly performs a series of two firings. On the first firing an input is accepted, and on the second an output of the same value is produced. The scenario corresponding to a **merge** firing sequence is obtained by removing from the total ordering implied by the sequence all dependencies between events occurring at different input ports.

Given any two input streams X and Y, the set of all possible **merge** scenarios may be more formally described by use of an *oracle* [33] function d. Letting $|X|$ denote the length of X (which may be infinite), d is a function mapping the integers from one to $|X| + |Y|$ to the two-element set $\{1, 2\}$. If $d(n)$ is one, then the n'th element of the output is chosen from input stream X. Otherwise, it is chosen from Y. The set of integers which d maps to one must equal X in size, and similarly the set it maps to two must equal Y in size.

For any oracle d satisfying these constraints, there is a scenario corresponding to a possible execution of **merge** with input streams X and Y in which d "controls" the choices of the **merge**. The set E of events of this scenario is defined by the following union:

$$\begin{aligned} E = {} & \{\langle In_1, n\rangle \mid 1 \leq n \leq |X|\} \cup \{\langle In_2, n\rangle \mid 1 \leq n \leq |Y|\} \\ & \cup \{\langle Out_1, n\rangle \mid 1 \leq n \leq |X| + |Y|\} \end{aligned}$$

With d we associate a function p such that:

$$p(n) = |\{i \mid 1 \leq i \leq n \text{ and } d(i) = d(n)\}|$$

That is, $p(n)$ is the position of the n'th output token within its original input history. Using p the valuation function V may be specified as:

$$\begin{aligned} V(\langle In_1, n\rangle) &= X_n \\ V(\langle In_2, n\rangle) &= Y_n \\ V(\langle Out_1, n\rangle) &= X_{p(n)}, \text{ if } d(n) = 1 \\ &= Y_{p(n)}, \text{ if } d(n) = 2 \end{aligned}$$

and the causality relation C, as:

$$\begin{aligned} C = \{ & (\langle In_1, n\rangle, \langle In_1, m\rangle) \mid n \leq m\} \\ \cup \{ & (\langle In_2, n\rangle, \langle In_2, m\rangle) \mid n \leq m\} \\ \cup \{ & (\langle Out_1, n\rangle, \langle Out_1, m\rangle) \mid n \leq m\} \\ \cup \{ & (\langle In_1, n\rangle, \langle Out_1, m\rangle) \mid \exists\, o \text{ such that } p(o) = n, d(o) = 1, \text{ and } o \leq m\} \\ \cup \{ & (\langle In_2, n\rangle, \langle Out_1, m\rangle) \mid \exists\, o \text{ such that } p(o) = n, d(o) = 2, \text{ and } o \leq m\} \end{aligned}$$

5.3 Operational Consistency

With scenario sets defined for elementary dataflow operators and scenario set composition rules defined for the dataflow graph–building operators, it is now possible to generate scenario sets for all dataflow graphs. To convince the reader of the usefulness of the scenario model of computation, we wish to conclude this chapter by arguing, perhaps somewhat informally, that scenario sets are a faithful representation of dataflow computation.

In Chapter 2, we introduced token-pushing, the obvious operational model of dataflow computation in which a graph changes state through the firings of its constituent operators. The state of the graph is both the state of its operators and the state of its arcs, that is, the queues of tokens "stored" on its arcs. In token-pushing a graph may be characterized as a set of sequences of operator firings. Each firing changes the internal state of a single operator and changes a single graph link by either removing (reading) or adding (writing) a value. For the determinate operators, all Kahn processes, a firing is the execution of either a **get** or **put** operation. Although the **merge** cannot be written in Kahn's language, its firings resemble executions of **get** or **put** operations (although under non-determinate control) and will be considered as such throughout the remainder of this section.

The proof of operational faithfulness will be a demonstration that for any given dataflow graph the possible computations represented by its scenarios are precisely the same as those represented by its firing sequences. We must show that scenario sets predict all possible computations, that is, for every firing sequence a corresponding scenario must be found; and we must show that scenario sets do not predict any impossible computations, that is, for every scenario a corresponding firing sequence must be found. Stated formally, except for the details of the meaning of firing sequence–scenario correspondence which we shall discuss in the proof, our theorem of the operational faithfulness of the scenario model of non-determinate dataflow computation is:

> **Theorem:** If G is a dataflow graph constructed from the set of operators consisting of the determinate Kahn processes plus the non-determinate **merge** operator by use of the graph-building operators $\|_G$, $[\bullet \looparrowleft_G \bullet]$, and $[\bullet \rightarrow_G \bullet]$ and S is the corresponding scenario set constructed from the scenario sets of these determinate processes and the **merge** operator by use of the scenario set operators $\|_S$, $[\bullet \looparrowleft_S \bullet]$, and $[\bullet \rightarrow_S \bullet]$, then for every firing sequence of G there is a corresponding scenario of S and for every scenario of S there is a corresponding firing sequence of G.

The essence of this theorem is: whether one constructs the scenario set of a system directly from the firing sequences of its dataflow graph or by use of the scenario set composition rules, the result is the same. Our first task in proving this theorem is to define the "correspondence" between firing sequences and scenarios. Given a dataflow graph G and one of its firing sequences f_1f_2 ..., the input and output histories generated by f_1f_2 ... are simply the values read by get's on graph input ports and written by put's on graph output ports.[1]

In the preceding section, we obtained scenarios from firing sequences of elementary dataflow operators by dropping all inter-port causalities implied by the firing sequences except those from input to output ports. A similar tactic will be used here to find scenarios corresponding to graph firing sequences. For each operator of the graph, there is within $f_1 f_2 \ldots$ a subsequence of that operator's own firings. From each of these subsequences, the scenario followed by that operator during the *graph* computation prescribed by $f_1 f_2 \ldots$ may be uncovered. By combining the causality relationships of all these operators, a scenario can be generated for the entire graph executing the firing sequence $f_1 f_2 \ldots$. In this scenario, a graph output will be causally related to all those input values which contributed to its existence by a direct chain of operator firings.

Operational faithfulness may be proven by induction on the set of dataflow graphs as generated from the set of elementary dataflow operators by use of the graph-building operators. The correspondence of firing sequences and scenarios for elementary operators follows from the definitions of the previous section. It is also easy to see that the correspondence is preserved by the two "trivial" graph operators: graph union $\|_G$ and port relabeling $[\bullet/_G\bullet]$. Consequently, we shall direct our attention to port connection $[\bullet \rightarrow_G \bullet]$, the remaining graph operation.

Let G be a dataflow graph with output port α and input port β, and let S be the scenario set characterization of G. Our first case will be that of finding a scenario of $S\ [\alpha \rightarrow_S \beta]$ for each firing sequence of $G\ [\alpha \rightarrow_G \beta]$. Suppose $f_1 f_2 \ldots$ is a firing sequence of $G\ [\alpha \rightarrow_G \beta]$ which causes the history X to pass through the α–β connection. Then $f_1 f_2 \ldots$ will also be a firing sequence for a

1. During some computations an operator will fail to read all its input. Such cases are easily handled by augmenting the firing sequence execution model with sequences of untouched input values.

computation of G in which history X is both received at input port α and produced at output port β.

By our inductive hypothesis, there will exist a scenario of S corresponding to this firing sequence of G. This scenario will also be α-β causality-consistent, as it ultimately originates from a firing sequence of $G\ [\alpha \rightarrow_G \beta]$, and consequently it may be used to construct a scenario of $S\ [\alpha \rightarrow_S \beta]$. Thus, there is a scenario in $S\ [\alpha \rightarrow_S \beta]$ for every firing sequence in $G\ [\alpha \rightarrow_G \beta]$.

The remaining case is the only difficult one for it is here that we must run counter to the abstraction from firing sequences to scenarios. Suppose $\langle E, V, C \rangle$ is a scenario of $S\ [\alpha \rightarrow_S \beta]$. Then there is an α-β causality-consistent scenario $\langle E', V', C' \rangle$ of S from which $\langle E, V, C \rangle$ is derived. By our induction hypothesis, there is a firing sequence $f_1 f_2 \ldots$ of G that corresponds to this α-β causality-consistent scenario. Unfortunately, $f_1 f_2 \ldots$ need not be a firing sequence of $G\ [\alpha \rightarrow_G \beta]$. As long as the α-β connection remains open, output values at α may be "delayed" arbitrarily long without contradicting the scenario's causality relation. (Remember, scenarios have no output-to-input causalities.) The firing sequence $f_1 f_2 \ldots$ must be some linearization of the α-β causality-consistent scenario $\langle E', V', C' \rangle$ and, although it is never the case that $\langle \beta, m \rangle\ C'\ \langle \alpha, m \rangle$ or the m'th value read at β causes the m'th value written at α, it is possible, at some point in this particular linearization, that the receiving of the m'th value at β does precede the producing of the m'th value on α. Of course, if this were so, $f_1 f_2 \ldots$ could not, without rearrangement, be made into a firing sequence of $G\ [\alpha \rightarrow_G \beta]$. However, $f_1 f_2 \ldots$ is but one linearization of $\langle E', V', C' \rangle$, and there are alternate linearizations, and consequently legitimate firing sequences, in which firing $\langle \alpha, m \rangle$ does precede $\langle \beta, m \rangle$.

Such a firing sequence can be constructed by placing in it first all those firings needed to produce $\langle \alpha, 1 \rangle$ and then all those needed to produce $\langle \alpha, 2 \rangle$ and so on. Because it is never the case that $\langle \beta, m \rangle$ causally precedes $\langle \alpha, m \rangle$, firing $\langle \beta, m \rangle$ will never be placed before $\langle \alpha, m \rangle$. This new firing sequence, a rearrangement of the original, will be a possible computation of

$G\ [\alpha \to_G \beta]$. Thus there are firing sequences in $G\ [\alpha \to_G \beta]$ for every scenario in $S\ [\alpha \to_S \beta]$. With both cases satisfied, the operational faithfulness of scenario sets is established.

The operational faithfulness of scenarios rests in their ability to incorporate physical causality in system representation. Firing sequences also represent this causality, but in doing so they include inter-port causalities other than those from input to output ports even though such causalities cannot be detected when dataflow graphs are connected through unbounded, time-independent communication channels. By omitting these unobservable causalities, we are able to develop an abstract, but nonetheless faithful, model of non-determinate dataflow computation.

6. Conclusion

In Chapter 3 we discussed three characteristics of a useful semantic theory, abstractness, mathematical tractability, and operational faithfulness. In this thesis we have developed a semantic theory for non-determinate computation in which dataflow graphs are represented by sets of scenarios, each corresponding to a particular computation which that graph may perform. The *purest* input-output specification, and hence the most abstract useful semantic representation, of dataflow computation would be a relation from the histories of values received at graph input ports to the possible histories of values produced at graph output ports. However, we have conclusively demonstrated that such history relations are inadequately detailed to capture all nuances of computational behavior which may become manifest when graphs are interconnected.

Scenario sets are a straightforward extension of history relations which do contain sufficient information to determine the result of graph interconnection. The scenario, the constituent of the scenario set, is nothing more that an input history-output history pair, the constituent of history relations, augmented by a causality relation placed on the individual elements of the histories. Although scenario sets are not a pure input-output semantic specification of non-determinate dataflow computation, they are not very far from that most desirable, though unobtainable, level of abstraction.

Along with scenario sets we have presented composition rules for deriving the scenario set representation of a dataflow graph from those of its components. Compared to many proposed semantic theories, be they based on global-state models [23] or on fixed point theory [10, 25], our composition rules are strikingly simple. The most sophisticated mathematical technique required to understand and utilize the scenario model of computation is the extension of a partial order. On account of its mathematical tractability, this theory should serve as a useful basis from which other results, such as the certification of verification methodologies, may be derived.

The remaining goal of a useful semantic theory was operational faithfulness. We have formally proven this property for scenario sets by demonstrating their consistency with token-pushing, the standard global-state operational model of dataflow computation. Furthermore, unlike most proposed fixed point theories [10, 25] of dataflow computation, scenario sets are able to model the *fair* **merge**, the **merge** described in detail in Chapter 5 which interleaves all its input tokens into one output stream even if one input stream is infinite. The continuity requirements of fixed point theories seem to restrict their application to the *unfair* **merge** in spite of its very limited usefulness in important application areas such as resource control and management [3]. Most of the semantic theories (with the notable exception of Park's [34]) that are capable of representing fair computation abstract very little from program execution models. For example, Clinger's [11] theory represents computation by "event diagrams" incorporating all internal, as well as external, elements of program execution.

There are several ways in which the research of this thesis could be extended. One is the extension of the semantic model to other application areas. Pratt [37] has already defined a "repackaging" of scenario set theory called the *process model*. The programs of the process model are *nets*. Like dataflow graphs, nets consist of processes connected by fixed channels of unbounded capacity. However, these channels are not FIFO queues: nets are dataflow graphs in which values may overtake others in transit between processes. The semantic representation of a net is a set of *traces*. Traces are a generalization of scenarios in which all the elements of the same port are no longer required to be totally ordered. The semantics of process composition using traces is similar to that of process composition using scenarios: consistent net traces are generated for all constituent processes and then the partial order implied by all process traces is restricted to the net ports. Besides having obvious application in distributed computing where inter-process communication is generally considered unordered [27], the process model (or perhaps even a simplification of it) could be applied to the U-interpreter for dataflow computation developed by Arvind and Gostelow [4] in which streams may contain "holes," a condition implemented by "tagging" otherwise unordered values with their stream positions.

The extension of scenarios to even more generalized versions of streams, in particular those containing other streams as "elementary" values, seems quite feasible. Streams of streams are already supported by the previously-mentioned U-interpreter [4] and can be used quite advantageously in many programming situations. A similar data structure currently receiving much attention is the sequence of Backus' [7] FP. Though FP is presently a purely determinate language and thus too "simple" for scenario theory, an extension of the language to important inherently non-determinate application areas would be well suited for some scenario-like semantic theory.

In this thesis we have constructed a semantic theory for *static* dataflow graphs, that is, graphs which cannot change in structure during a computation. However, *dynamic* dataflow graphs are a more appropriate implementation-level model of languages with functions and procedures because they allow function calls to extend graphs. Dynamic graphs can also support some very important non-determinate systems such as resource allocators for varying numbers of users. The development of a new scenario composition rule for recursively generated dynamic graphs seems a natural point in which to incorporate some aspects of fixed point theory into the scenario model. Thus far we have developed our theory without reference to monotonicity or continuity even though related properties must hold for the scenario sets of any realizable system. For example, it is obvious that the scenario for a large input sequence must be an extension of some scenarios of its input prefixes and that the scenario for an infinite input sequence must be the limit of some scenarios of its finite input prefixes. The statement of monotonicity and continuity requirements for scenarios seems a natural first step toward the development of a fixed point theory based on scenario sets.

Although we have consistently presented scenarios as partial orders, it is possible to assume a functional viewpoint that should be more palatable to fixed point theorists. The causality information of every scenario may be represented by a function from input histories to output histories in which each input history prefix is mapped into the output values it generates without the aid of those input values which follow it. Such associations will only produce partial functions; however, the partial functions associated with classes of scenarios following the same computational "route" may be combined to form total functions. Thus it is possible to move from characterization of dataflow graphs by scenario sets to characterization by sets of functions. Such a transition should allow the more fruitful application of the tools and techniques of fixed point theory to scenarios.

Clearly one of the main hopes for any formal semantics is that its machinery for describing programming language features and concepts will aid programmers in the development of correctly functioning software. There are many ways in which scenarios may be used to accomplish this goal. Most obviously, using scenario theory it is possible both to specify the desired result of a system as a scenario set and to prove the equivalence of that specification to a dataflow program or graph by deriving its own scenario set. However, even if such proofs are possible, in general they would be neither trivial nor enjoyable and often they would not even be feasible. Both the specification and the verification end of this process may be improved. First, there are undoubtedly many kinds of specification more appropriate for proving program correctness than scenarios. Scenario sets were developed to reveal all properties of behavior which must be known in order to determine the result of arbitrary interconnections of operators and graphs. This level of detail is very rarely required in the formal specification of a desired system: generally, the end-users of the system will be interested only in its input-output behavior or history relation and usually in only a few relevant input cases of that. Their needs will be meet with these more abstract specifications, even if two programs satisfying their requirements exhibit subtle differences in behavior when made to interact with other programs. If a problem were specified as a history relation or even as some abstraction thereof, any program whose scenario set was consistent with that specification could be considered to implement it.

Nevertheless, as long as the verification of consistency is performed by proof of semantic identity using scenario sets, formally proving programs correct will often be difficult. Easier proofs could result from the development an axiomatic theory of program correctness, perhaps by attaching assertions to the links of dataflow graphs rather like Floyd [17] attaches assertions to the links of flowcharts. By use of the techniques of Ellis [15] or Wadge [43], graphs may be shown to satisfy many useful properties, *e. g.* absence of deadlock, that are often an important part of system specification.

Although axiomatic theories might permit the direct verification of programs without reference to scenario theory, scenario theory would play a very important role in the development of these verification methodologies because it would allow the certification of the formalisms of the methodologies. Since we have already demonstrated the consistency of our semantic theory with a detailed operational one, the correctness of a verification methodology for non-determinate dataflow computation may be demonstrated by proving its consistency with the scenario model of computation. This proof should be much easier than one showing direct consistency with the significantly less abstract operational model.

One final way, perhaps the most important, in which formal semantics should aid the programmer's task is by influencing the design of programming languages. It is well known [42] that difficult-to-control programming language features, such as side effects and *goto*'s, lead to complicated programming language semantics and proof rules. Thus far, research in semantics appears to have reinforced our judgments of programming language features rather than to have guided our design of them. Perhaps this has happened because most of our popular programming constructs were developed before our formal methodologies were. With non–determinate computation the situation is quite different: high–level constructs for the problems of this area have only recently begun to appear. Non–determinate dataflow languages seem to offer some advantages relative to non–determinate languages with a more conventional shared–memory multi–processing orientation; for example, with streams it is possible to write programs which exhibit state without side effects. However, through the merge anomaly we have already shown that the unconstrained interconnection of processes can result in unexpected semantic complexity even in the seemingly simple dataflow approach to inter–process communication. Maybe this particular complexity is unavoidable; maybe it is not. A semantic theory may not reveal how to avoid complexity, but at least it will reveal complexity.

References

[1] Ackerman, W. B., "Data Flow Languages", *Computer 15*, 2(February 1982), 15–25.

[2] Ackerman, W. B., and J. B. Dennis, *VAL — A Value-Oriented Algorithmic Language: Preliminary Reference Manual*, Laboratory for Computer Science (TR-218), MIT, Cambridge, Massachusetts, June 1979.

[3] Arvind, and J. D. Brock, "Streams and Managers", *Operating Systems Engineering: Proceedings of the Fourteenth IBM Computer Science Symposium* (M. Maekawa, L. A. Belady, Eds.), *Lecture Notes in Computer Science 143*, October 1980, 452–465.

[4] Arvind, and K. P. Gostelow, "The U-Interpreter", *Computer 15*, 2(February 1982), 42–49.

[5] Arvind, K. P. Gostelow, and W. Plouffe, *An Asynchronous Programming Language and Computing Machine*, Department of Information and Computer Science (TR 114a), University of California - Irvine, Irvine, California, September 1978.

[6] Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, Monitors and Dataflow", *Proceedings of the Sixth ACM Symposium on Operating Systems Principles, Operating Systems Review 11*, 5(November 1977), 159–169.

[7] Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM 21*, 8(August 1978), 613–641.

[8] Brock, J. D., and W. B. Ackerman, "Scenarios: A Model of Non-determinate Computation", *International Colloquium on Formalization of Programming Concepts* (J. Díaz, I. Ramos, Eds.), *Lecture Notes in Computer Science 107*, April 1981, 252–259.

[9] Brookes, S. D., "On the Relationship of CCS and CSP", *Automata, Languages and Programming: Tenth Colloquium* (J. Díaz, Ed.), *Lecture Notes in Computer Science 154*, July 1983, 83–96.

[10] Broy, M., "Fixed Point Theory for Communication and Concurrency", *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts - II* (D. Bjørner, Ed.), June 1982, 125–147.

[11] Clinger, W., *Foundations of Actor Semantics*, Artificial Intelligence Laboratory (TR-633), Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1981.

[12] Dennis, J. B., "First Version of a Data Flow Procedure Language", *Programming Symposium: Proceedings, Colloque sur la Programmation* (B. Robinet, Ed.), *Lecture Notes in Computer Science 19*, April 1974, 362–376.

[13] Dennis, J. B., "A Language Design for Structured Concurrency", *Design and Implementation of Programming Languages: Proceedings of a DoD Sponsored Workshop* (J. H. Williams, D. A. Fisher, Eds.), *Lecture Notes in Computer Science 54*, October 1976, 231–242.

[14] Egli, H., "A Mathematical Model for Nondeterministic Computations", Forschungsinstitut für Mathematik, Technological University, Zürich, Switzerland, 1975.

[15] Ellis, D. J., *Formal Specifications for Packet Communication Systems*, Laboratory for Computer Science (TR-189), MIT, Cambridge, Massachusetts, November 1977.

[16] Faustini, A. A., "An Operational Semantics of Pure Dataflow", *Automata, Languages, and Programming: Ninth Colloquium* (M. Nielsen, E. M. Schmidt, Eds.), *Lecture Notes in Computer Science 140*, July 1982, 212–224.

[17] Floyd, R. W., "Assigning Meaning to Programs", *Mathematical Aspects of Computer Science* (J. T. Schwartz, Ed.), *Proceedings of Symposia in Applied Mathematics 19*, April 1977, 19–32.

[18] Hewitt, C. E., and H. Baker, "Actors and Continuous Functionals", *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts* (E. J. Neuhold, Ed.), August 1977, 367–390.

[19] Hoare, C. A. R., "Communicating Sequential Processes", *Communications of the ACM 21*, 8(August 1978), 666–677.

[20] Hoare, C. A. R., S. D. Brookes, and A. W. Roscoe, "A Theory of Communicating Sequential Processes", Programming Research Group (PRG-16), Oxford University, Oxford, England, 1981.

[21] Jayaraman, B., and R. M. Keller, "Resource Control in a Demand-driven Data-flow Model", *Proceedings of the 1980 International Conference on Parallel Processing*, August 1980, 118–127.

[22] Kahn, G., "The Semantics of a Simple Language for Parallel Programming", *Information Processing 74: Proceedings of IFIP Congress 74* (J. L. Rosenfeld, Ed.), August 1974, 471–475.

[23] Karp, R. M., and R. E. Miller, "Parallel Program Schemata", *Journal of Computer and System Sciences 3*, 2(1969), 147–195.

[24] Keller, R. M., "Denotational Models for Parallel Programs with Indeterminate Operators", *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts* (E. J. Neuhold, Ed.), August 1977, 337–366.

[25] Kosinski, P. R., "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs", *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, January 1978, 214-221.

[26] Lehmann, D. J., "Categories for Fixpoint Semantics", *Seventeenth Annual Symposium on Foundations of Computer Science*, October 1976, 122-126.

[27] Liskov, B., and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM Transactions on Programming Languages and Systems 5*, 3(July 1983), 381-404.

[28] McCarthy, J., "A Basis for a Mathematical Theory of Computation", *Computer Programs and Formal Systems* (P. Braffort and D. Hirschberg, Eds.), North-Holland, Amsterdam, 1963, 33-70.

[29] McGraw, J. R., "The VAL Language: Description and Analysis", *ACM Transactions on Programming Languages and Systems 4*, 1(January 1982), 44-82.

[30] Milne, G., and R. Milner, "Concurrent Processes and Their Syntax", *Journal of the ACM 26*, 2(April 1979), 302-321.

[31] Milner, R., "Processes: A Mathematical Model of Computing Agents", *Logic Colloquium '73* (H. E. Rose and J. C. Shepherdson, Eds.), North-Holland, Amsterdam, 1975, 157-174.

[32] Milner, R., *A Calculus of Communicating Systems, Lecture Notes in Computer Science 92*, 1980.

[33] Park, D., "On the Semantics of Fair Parallelism", *Abstract Software Specifications: 1979 Copenhagen Winter School Proceedings* (D. Bjørner, Ed.), *Lecture Notes in Computer Science 86*, February 1979, 504-526.

[34] Park, D., "The 'Fairness' Problem and Nondeterministic Computing Networks", *Proceedings of the Fourth Advanced Course on Theoretical Computer Science*, Mathematisch Centrum, Amsterdam, The Netherlands, 1982.

[35] Patil, S. S., "Closure Properties of Interconnections of Determinate Systems", *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, June 1970, 107-116.

[36] Plotkin, G. D., "A Powerdomain Construction", *SIAM Journal of Computing 5*, 3(September 1976), 452-487.

[37] Pratt, V. R., "On the Composition of Processes", *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982, 213-223.

[38] Rounds, W. C. and S. D. Brookes, "Possible Futures, Acceptances, Refusals, and Communicating Processes", *Proceedings of the Twenty-second Symposium on Foundations of Computer Science*, October 1981, 140–149.

[39] Scott, D. S., "Data Types as Lattices", *SIAM Journal of Computing 5*, 3(September 1976), 522-587.

[40] Smyth, M. B., "Power Domains", *Journal of Computer and System Sciences 16*, 1(February 1978), 23–36.

[41] Stoy, J. E., *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Massachusetts, 1977.

[42] Tennent, R. D., "The Denotational Semantics of Programming Languages", *Communications of the ACM 19*, 8(August 1976), 437–453.

[43] Wadge, W. W., "An Extensional Treatment of Dataflow Deadlock", *Theoretical Computer Science 13*, 1(January 1981), 3–15.

[44] Weng, K.–S., *Stream–Oriented Computation in Recursive Data Flow Schemas*, Laboratory for Computer Science (TM–68), MIT, Cambridge, Massachusetts, October 1975.

Biographic Note

Dean Brock was born on April 30, 1952 in Waynesville, North Carolina, and grew up in Durham, Little River (Transylvania County), and Statesville, North Carolina. In 1972, he married Ruth Miller Wilson of Easley, South Carolina.

In 1970, Dean Brock graduated from Easley High School in Easley, South Carolina. He received his A. B. degree *summa cum laude* in mathematics from Duke University in 1974. From 1974 to 1982, Dean Brock was a graduate student at the Massachusetts Institute of Technology. There he was associated with the Computation Structures Group of the MIT Laboratory for Computer Science. During this time he received financial support from a National Science Foundation Graduate Fellowship and from research and teaching assistantship appointments. In 1979, he received the S. M. and E. E. degrees.

In August 1982, Dean Brock returned to the South. He is presently an assitant professor in the Department of Computer Science at the University of North Carolina at Chapel Hill.

Publications

Arvind, and J. D. Brock, "Streams and Managers", *Operating Systems Engineering: Proceedings of the Fourteenth IBM Computer Science Symposium* (M. Maekawa, L. A. Belady, Eds.), *Lecture Notes in Computer Science 143*, October 1980, 452–465.

Brock, J. D., *Operational Semantics of a Data Flow Language*, Laboratory for Computer Science (TM-120), MIT, Cambridge, Massachusetts, December 1978.

Brock, J. D., "Consistent Semantics for a Data Flow Language", *Mathematical Foundations of Computer Science 1980: Proceedings of the Ninth Symposium* (P. Dembriński, Ed.), *Lecture Notes in Computer Science 88*, September 1980, 168–180.

Brock, J. D., and W. B. Ackerman, "Scenarios: A Model of Non-determinate Computation", *International Colloquium on Formalization of Programming Concepts* (J. Díaz, I. Ramos, Eds.), *Lecture Notes in Computer Science 107*, April 1981, 252–259.

Brock, J. D., and L. B. Montz, "Translation and Optimization of Data Flow Programs", *Proceedings of the 1979 International Conference on Parallel Processing* (O. N. Garcia, Ed.), August 1979, 46–54.