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 _____
Department of Electrical Engineering and Computer Science
August 23, 1983

Certified by _____
Jack B. Dennis
Thesis Supervisor

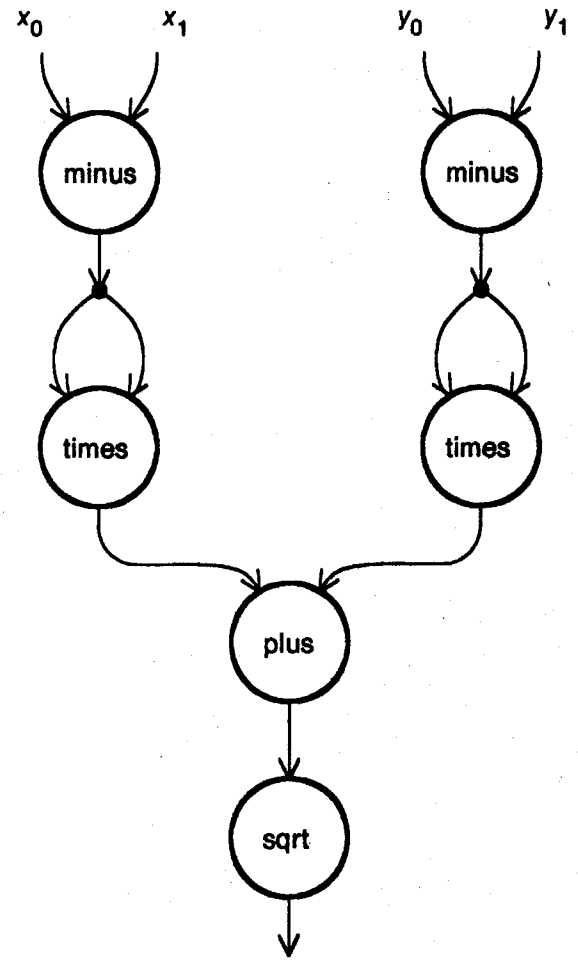
Accepted by _____
Arthur C. Smith
Chairman, Departmental Graduate Committee

Table of Contents

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.

Figure 2.1. An example dataflow graph



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

---

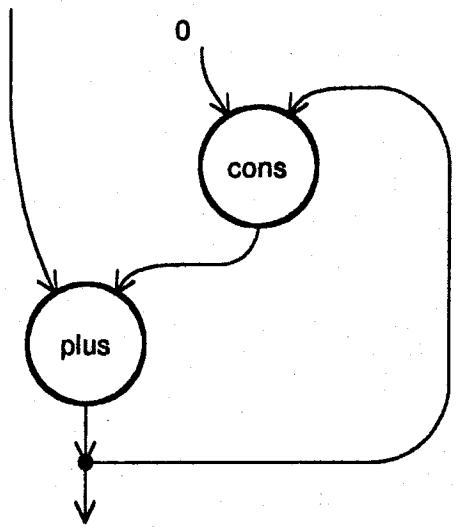
through the graph.

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

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



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.

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 $\mathcal{F}[\![P]\!]$. When X is the tuple of histories presented on the input ports of P, $\mathcal{F}[\![P]\!](X)$ 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:

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

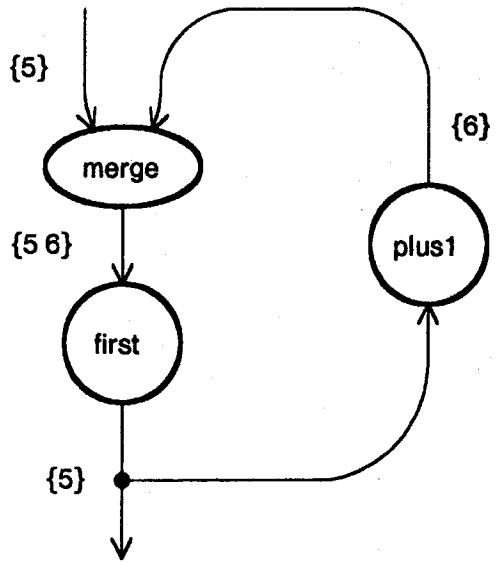
$\mathcal{F}[\![\text{plus}]\!](\Lambda, Y) = \Lambda$
$\mathcal{F}[\![\text{plus}]\!](X, \Lambda) = \Lambda$
$\mathcal{F}[\![\text{plus}]\!](x\ X, y\ Y) = x+y\ \mathcal{F}[\![\text{plus}]\!](X, Y)$
  Λ is the empty history
  x, y are single values
  X, Y are arbitrary (possibly empty) sequences of values

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

X = $\mathcal{F}[\![\textbf{cons}]\!]$(0, OUT)
OUT = $\mathcal{F}[\![\text{plus}]\!]$(IN, 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.

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:

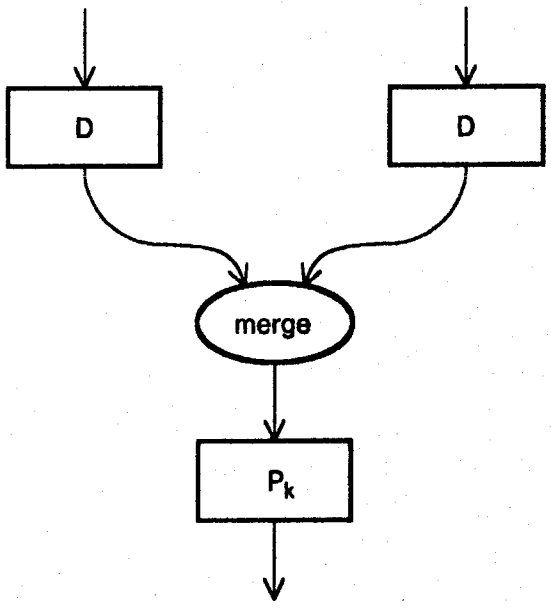
$S_k(X, Y) = P_k(\mathbf{merge}(D(X), D(Y)))$

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 P1 in (IN) out (OUT)
    x ← get(IN)
    put x on OUT
    y ← get(IN)
    put y on OUT
end P1

process P2 in (IN) out (OUT)
    x ← get(IN)
    y ← get(IN)
    put x on OUT
    put y on OUT
end P2
```

As history functions these three processes may be specified as:

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

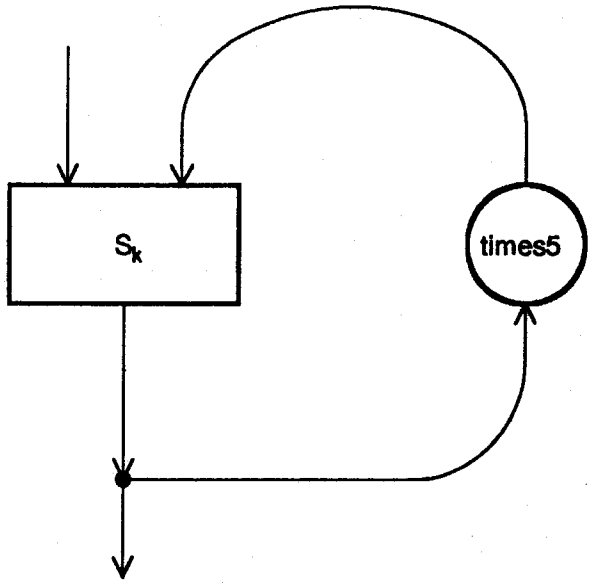
$P_1(\Lambda) = \Lambda$
$P_1(x) = x$
$P_1(x\ y\ Z) = x\ y$

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

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

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

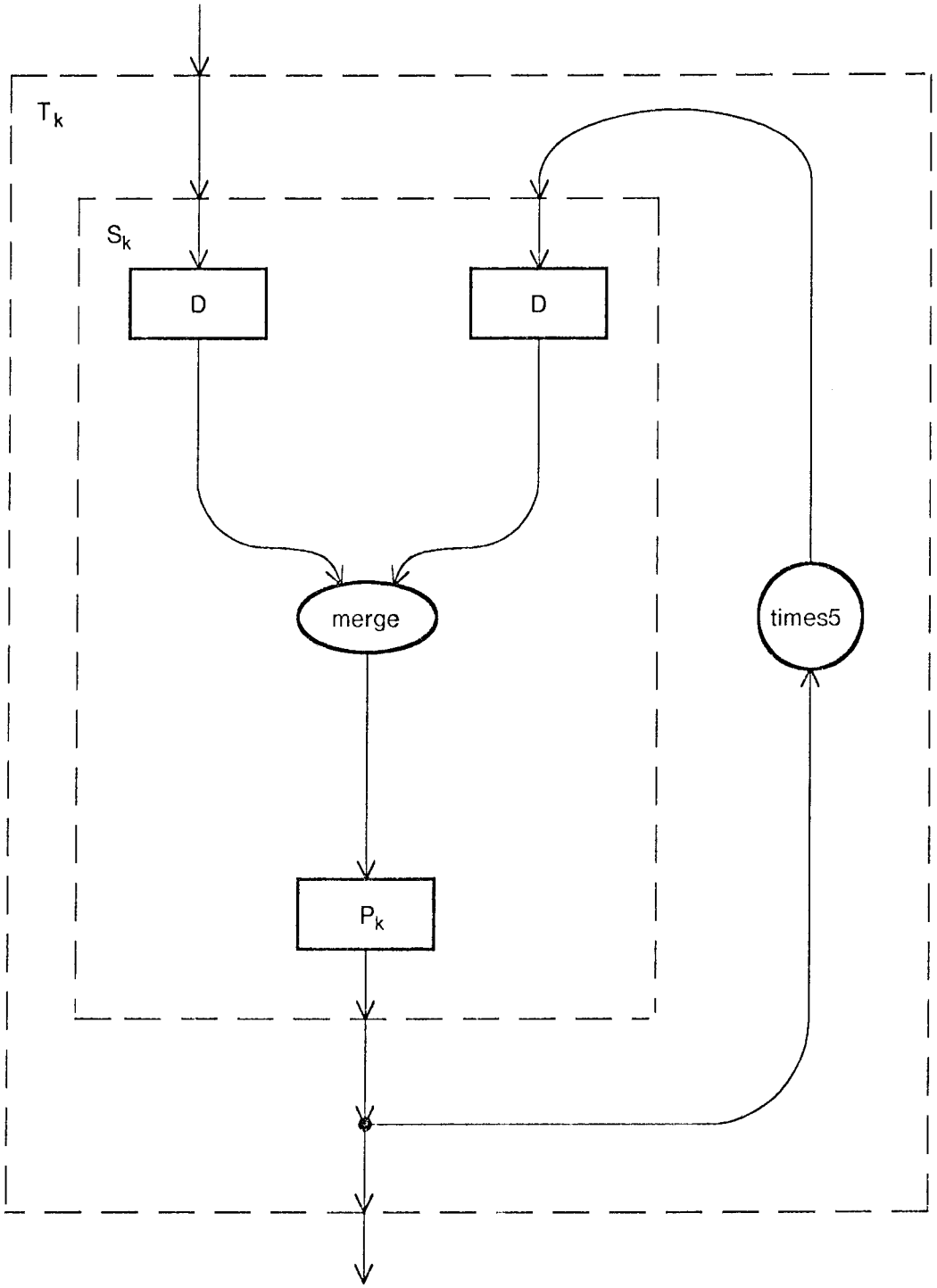


operator, to the rightmost input port of S_k. In Kahn's language, this interconnection may be specified as:

```
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)

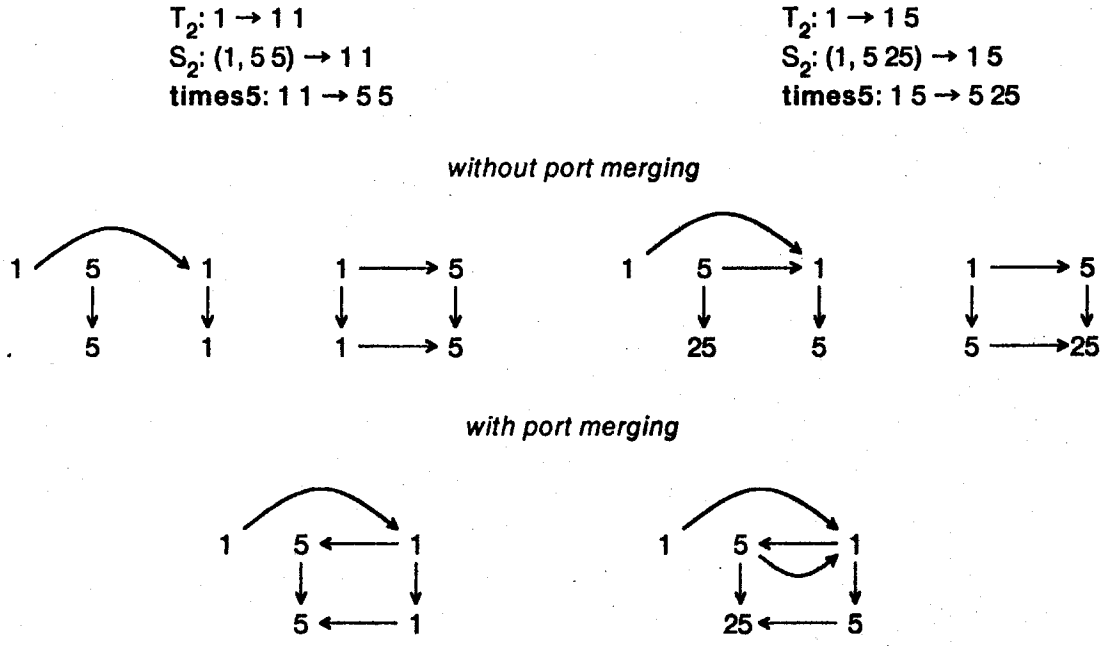


In Figure 3.10, the value–consistent pairs for T_2 are illustrated both with and without the merging of the connected ports. However, note that now only one of the merged value–consistent pairs is causality–consistent. In the other we see a cycle (between the one in S_k's output and the five in S_k's 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

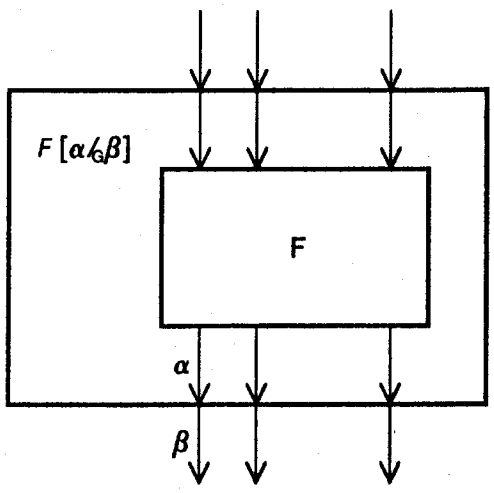
---

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



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.

Figure 4.2. Port relabeling [•$\not\leftarrow_G$•]

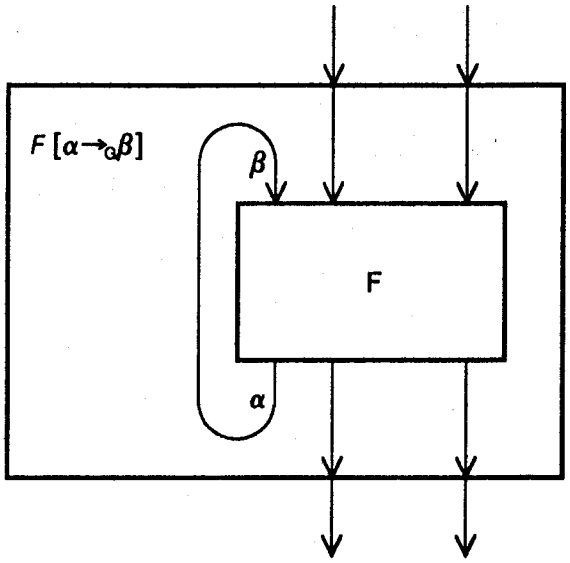


The third, and final, dataflow graph operation is port connection. Port connection [•$\rightarrow_G$•], like port relabeling, is a unary operator and is also an operator schema. The port connection operator [$\alpha \rightarrow_G \beta$] can be applied only to graphs G with an output port α and an input 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.

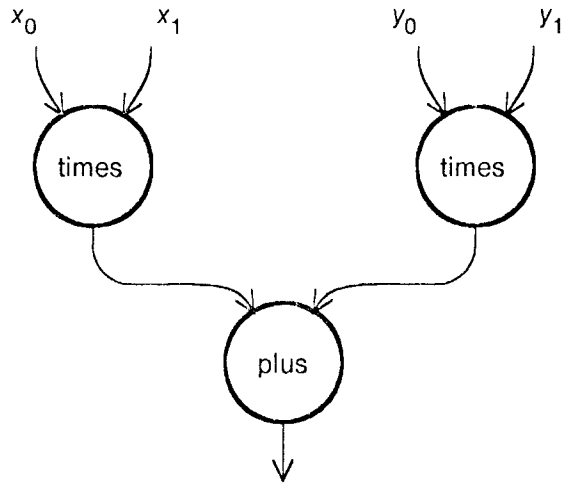
Figure 4.3. Port connection [•→$_G$•]



---

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



$$
\begin{aligned}
&(\text{times } [In_1 \not\leftarrow_G x_0] \; [In_2 \not\leftarrow_G x_1] \; [Out_1 \not\leftarrow_G Op_1Out_1] \; \|_G \\
&\quad \text{times } [In_1 \not\leftarrow_G y_0] \; [In_2 \not\leftarrow_G y_1] \; [Out_1 \not\leftarrow_G Op_2Out_1] \; \|_G \\
&\quad \text{plus } [In_1 \not\leftarrow_G Op_3In_1] \; [In_2 \not\leftarrow_G Op_3In_2] \; [Out_1 \not\leftarrow_G Result] \;) \\
&\qquad [Op_1Out_1 \rightarrow_G Op_3In_1] \; [Op_2Out_1 \rightarrow_G Op_3In_2]
\end{aligned}
$$

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 ⟨*E*, *V*, *C*⟩ 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)

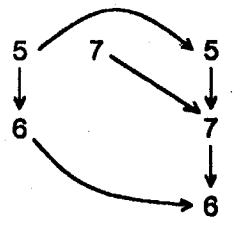


Figure 5.2. Hasse Diagram for a merge Causality Relation

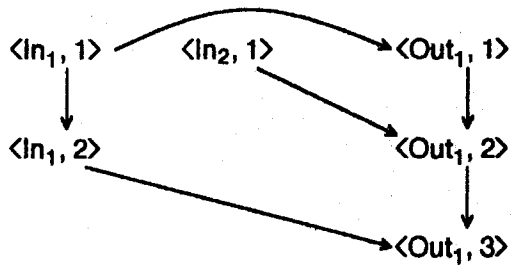


Figure 5.3. Antisymmetry: Cases 2 and 3

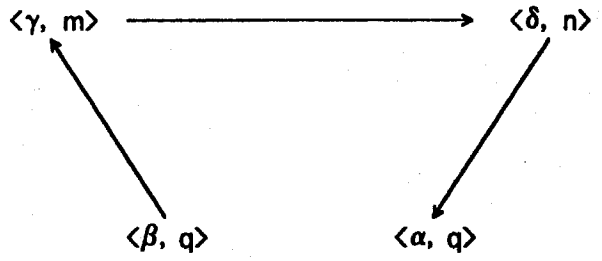
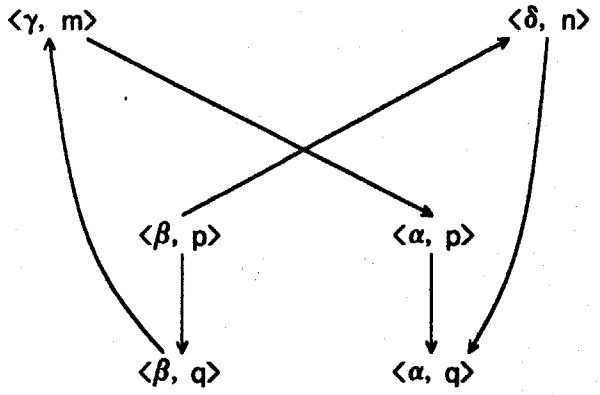


Figure 5.4. Antisymmetry: Case 4



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 $\langle\varepsilon, o\rangle$ are elements of E such that $\langle\gamma, m\rangle\ C^*\ \langle\delta, n\rangle$ and $\langle\delta, n\rangle\ C^*\ \langle\varepsilon, o\rangle$. 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 $\langle\delta, n\rangle\ C\ \langle\varepsilon, o\rangle$ or there is an element $\langle\alpha, q\rangle$ in E such that $\langle\delta, n\rangle\ C\ \langle\alpha, q\rangle$ and $\langle\beta, q\rangle\ C\ \langle\varepsilon, o\rangle$.

If $\langle\gamma, m\rangle\ C\ \langle\delta, n\rangle$ and $\langle\delta, n\rangle\ C\ \langle\varepsilon, o\rangle$ then $\langle\gamma, m\rangle\ C\ \langle\varepsilon, o\rangle$, and thus $\langle\gamma, m\rangle\ C^*\ \langle\varepsilon, o\rangle$, 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 $\langle\gamma, m\rangle\ C\ \langle\delta, n\rangle$ and there is an $\langle\alpha, q\rangle$ in E such that $\langle\delta, n\rangle\ C\ \langle\alpha, q\rangle$ and $\langle\beta, q\rangle\ C\ \langle\varepsilon, o\rangle$. This case is illustrated in Figure 5.5. As $\langle\gamma, m\rangle\ C\ \langle\delta, n\rangle$ and $\langle\delta, n\rangle\ C\ \langle\alpha, q\rangle$ then, by transitivity, $\langle\gamma, m\rangle\ C\ \langle\alpha, q\rangle$ and, from the definition of C^*, $\langle\gamma, m\rangle\ C^*\ \langle\varepsilon, o\rangle$. 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 $\langle\alpha, p\rangle$ and $\langle\alpha, q\rangle$ 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 $\langle\beta, q\rangle\ C\ \langle\varepsilon, o\rangle$. Furthermore, without loss of generality, assume that q is at least as great as p and, consequently, that $\langle\alpha, p\rangle\ C\ \langle\alpha, q\rangle$ as shown in Figure 5.6. Then from $\langle\gamma, m\rangle\ C\ \langle\alpha, p\rangle$, it follows that $\langle\gamma, m\rangle\ C\ \langle\alpha, q\rangle$ and, in turn, that $\langle\gamma, m\rangle\ C^*\ \langle\varepsilon, o\rangle$. 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

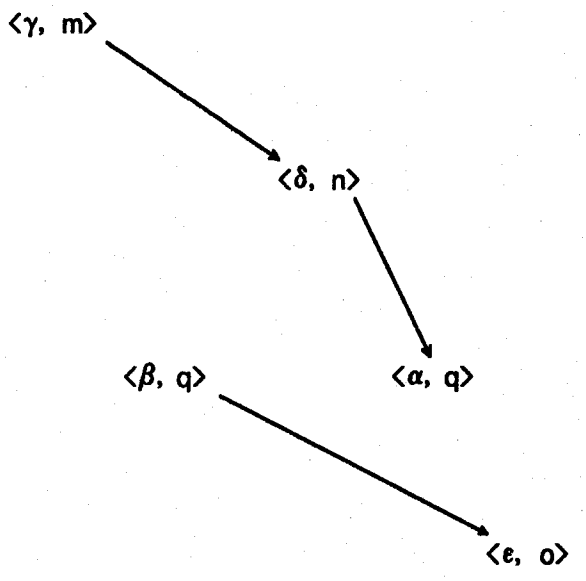
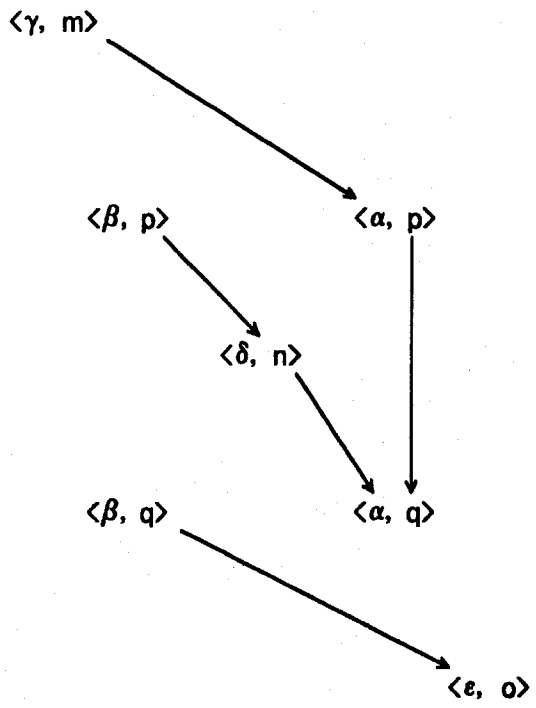


Figure 5.6. Transitivity: Case 4



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.

G $[\alpha \rightarrow_G \beta]$. Thus there are firing sequences in G $[\alpha \rightarrow_G \beta]$ for every scenario in S $[\alpha \rightarrow_S \beta]$. 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.

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.

[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.