CONSISTENCY IN DATAFLOW GRAPHS

by

Edward Ashford Lee

Memorandum No. UCB/ERL M89/125

9 November 1989

COVER PAGE

CONSISTENCY IN DATAFLOW GRAPHS

by

Edward Ashford Lee

Memorandum No. UCB/ERL M89/125

9 November 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

CONSISTENCY IN DATAFLOW GRAPHS

by

Edward Ashford Lee

Memorandum No. UCB/ERL M89/125

9 November 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

CONSISTENCY IN DATAFLOW GRAPHS

Edward Ashford Lee

U. C. Berkeley
Berkeley, CA 94720
(415) 642-0455
eal@janus.Berkeley.EDU

November 9, 1989

ABSTRACT

This paper explores the style of programming in which the programmer directly manipulates dataflow graphs, and arcs in the graph carry streams of tokens. Such a style of programming is attractive for certain application domains, such as digital signal processing, and has been studied extensively. One of its most serious problems is that subtle semantic inconsistencies between parts of the dataflow graph can be inadvertently created. These inconsistencies can lead to deadlock, or in the case of non-terminating programs, to unbounded memory requirements. Consistency is defined to mean that the same number of tokens are consumed as produced on any arc, in the long run. A *token-flow* model is developed for testing for consistency. The method is a generalization of consistency checks for *synchronous dataflow* (SDF) graphs [Lee87a]. Although inspired by the similar tests of Benveniste, et. al. [Ben88], the method and the languages to which it applies are different.

1. SEMANTIC INCONSISTENCIES

For certain software applications, such as digital signal processing, programming by directly manipulating dataflow graphs is attractive. However, when arcs in a dataflow graph are permitted to carry streams of tokens, subtle semantic inconsistencies between parts of the graph can be inadvertently created. These inconsistencies can lead to accumulation of tokens in memory. For non-terminating programs, which are common in signal processing, such inconsistencies imply either unbounded memory requirements or deadlock. These problems have been noted before by other researchers (see for example [Dav78]), and have resulted for example in the exclusion of operators in the ID language [Arv87][Nik88] that could lead to these problems. This paper describes a simple systematic method that can identify these inconsistencies, and does not have the restrictions of "clean" dataflow graphs [Dav78] or "well-behaved" dataflow graphs [Arv86]. The method would form a critical part of a compiler for languages that involve direct manipulation of dataflow graphs.

Some examples of such inconsistencies are shown in figure 1. The actors used here are ordinary functions plus **SWITCH** and **SELECT**, which route tokens conditional on a Boolean

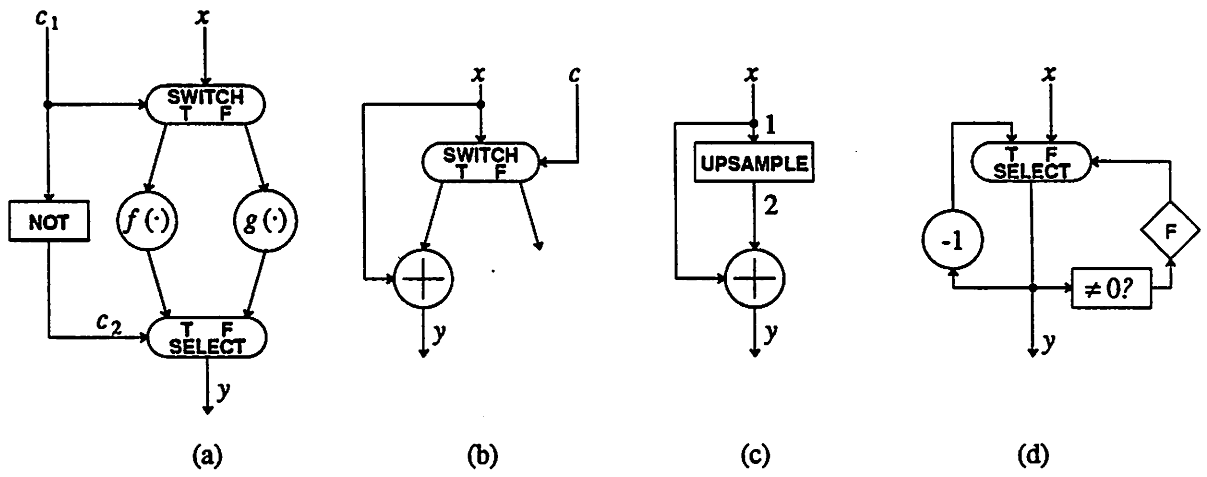


Figure 1. Some problematic configurations of dataflow actors.

input, as explained in figure 2. These are minor variations of the original Dennis actors [Den75], and are the same as the **DISTRIBUTOR** and **SELECTOR** in [Dav82]. The graph in figure 1a is structurally similar to a conditional. Without the **NOT** actor, this graph would implement the functional expression:

$$y = \text{if}(c) \text{ then } f(x) \text{ else } g(x). \tag{1}$$

The **NOT** actor, a Boolean negation, creates a problem. To understand precisely the difficulty, observe that the firing rule for the **SELECT** is that it must have a Boolean token on its control input and a token on the data input corresponding to the value of the Boolean. The first time a control token arrives, the **SELECT** cannot fire, because it does not have a token on

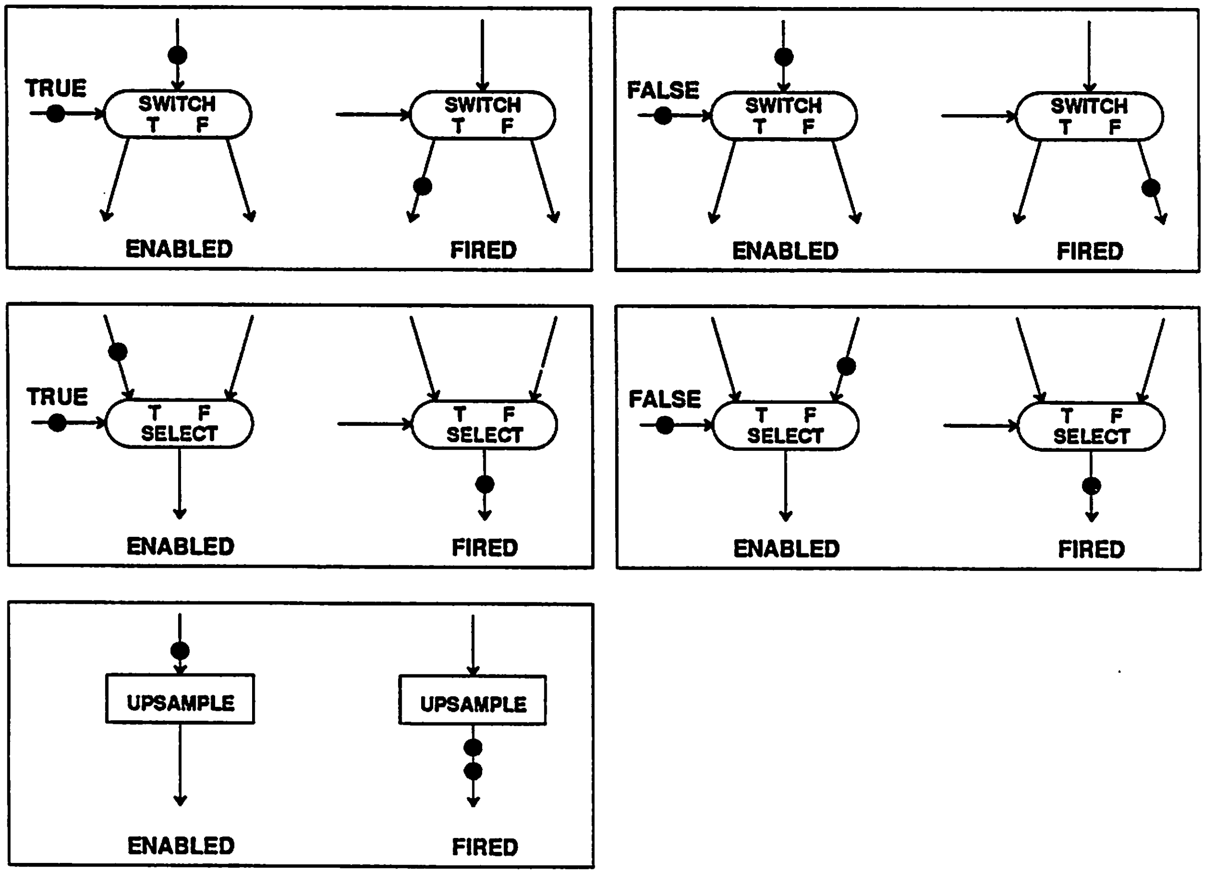


Figure 2. The behavior of SWITCH, SELECT, and UPSAMPLE actors for different input conditions is illustrated here. The UPSAMPLE actor can be parameterized to produce any number of tokens given an input token.

the proper data input. Some iterations later it may be able to fire. However, notice that unless the Boolean input stream consists of the same number of "true" and "false" tokens, then memory requirements for token storage will grow. If the program does not terminate, memory requirements could grow indefinitely.

Implicit in our language model is the FIFO behavior of arcs connecting actors, with no particular size limit. This can be implemented using tagged tokens [Arv82] or in some cases static buffering [Lee87b]. However, the language model can also subsume implementations with finite-size buffers by using feedback paths with delays. For example, the MIT *static dataflow* model [Den80] prohibits more than one token on an arc at one time. The feedback path in figure 3 models this. A *delay*, indicated with a diamond, can be viewed simply as an initial token on the arc. In this case, a token on the feedback path represents an empty location in the buffer on the feedforward path. The total number of delays in the loop (one) is equal to the size of the buffer. The numbers adjacent to the inputs and outputs of actors indicate that actor B requires one token on the feedback path (i.e. an empty location in the buffer) to fire. When actor C fires, it consumes one token from the forward path, freeing a buffer location, and indicating the free buffer location by putting one token on the feedback path.

In figure 1b, an input token x is added to itself if the Boolean input c is *true*, but if it is false, then the addition actor cannot fire, and no output token is produced. This may appear acceptable until we make the following observation. Dataflow semantics require that an arriv-

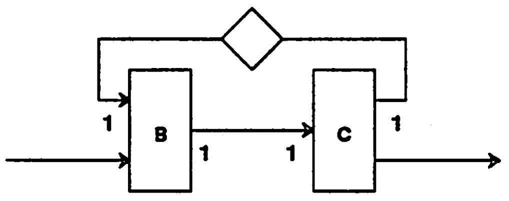


Figure 3. Finite token capacity on arcs can be modeled with feedback paths and delays.

ing token x be copied to provide an input for the switch actor and a separate input for the adder[1]. When the c token has value false, the x token remains on the arc into the adder, unconsumed. If this system is fired again, and the arriving c token is *true*, then the arriving x token will be added to the *previous* x token, rather than to itself. If the inputs are semi-infinite streams, then memory requirements are unbounded because every *false* adds one more token to the queue at the adder input.

In figure 1c, an **UPSAMPLE** actor produces two tokens for each one consumed, as shown in figure 2. This is indicated in figure 1c by the numbers adjacent to the input and output arcs. Such an actor can be used to implement iteration [Lee89a], and is consistent with the *synchronous dataflow* (SDF) model of computation [Lee87]. SDF means that for each input and output of each actor, the number of tokens produced and consumed when the actor fires is fixed and known at compile time. In figure 1c, if the inputs are semi-infinite streams, then the memory requirements on the arc connecting the **UPSAMPLE** to the adder are unbounded.

In figure 1d is an attempted implementation of a *guarded count*. Given an input $x = 4$, for example, a guarded counted produces a sequence of output tokens $y = 4, 3, 2, 1, 0$. Such a system could be used to implement data-dependent iteration, as we will see shortly. The initial token indicated by the delay has value **F**, for a Boolean false. On the arrival of a non-negative integer x, the **SELECT** actor can fire, putting the integer x on its output. If $x = 0$, then the output of the test function is false, and the subsystem waits for the arrival of another x. If $x > 0$, then it is decremented, selected, and tested again. However, there is a semantic inconsistency here that shows up the second time a token x arrives. Suppose the second x has value $x = 2$. It will be selected and tested, but the next firing of the select actor will consume the -1 token left over from the previous iteration. The output sequence will be

[1] Note that an efficient implementation may not need to actually perform the copy, but *logically*, the system must behave as if a copy occurred.

$y = 2,-1,1,-2,0$. On the next firing, there will be a -1 and -3 left over on the **T** input to the select. With repeated firings, the output will get bizarre indeed, and memory requirements will again become unbounded.

This paper gives a systematic method for finding these types of inconsistencies. The method is inspired by the algebraic techniques of Benveniste, et. al. [Ben88], which can accomplish some of the same objectives for a different class of languages. Fortunately, our method is simpler than that in [Ben88]. A comparison of the two methods and language classes will be made later in this paper.

2. EXAMPLES OF INTERESTING PROGRAMS

From the examples in figure 1, the alert reader might conclude that **SWITCH**, **SELECT**, and **UPSAMPLE** actors are the culprits, and should therefore be excluded from any practical language. Indeed, if these could be used only to realize constructs equivalent to those found in standard functional languages, such as if-then-else, then there would be little motivation for using them. Such higher level constructs are more familiar because of their similarity with imperative languages. More importantly, they cannot lead to the kinds of semantic inconsistencies described above. However, these actors can lead to some elegant programs with important advantages.

Consider the program in figure 4. The numbers adjacent to the inputs and outputs of the actors again indicate how many tokens are produced and consumed each time the actor fires. Consequently, the **C** actor fires 10 times for each firing of the **B** actor, which in turn fires 10

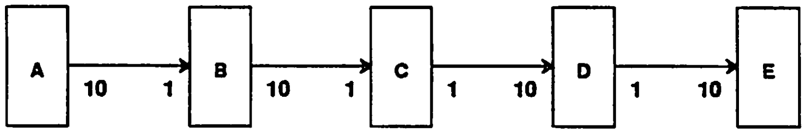


Figure 4. A nested manifest iteration expressed using the SDF model.

times for each firing of the **A** actor. Since there is nothing in this model to prevent simultaneous firing of the actors being iterated, this schema solves the first open problem listed by Dennis in [Den75], providing the semantics of a "parallel-for" in a dataflow language. Dependencies across cycles of the iteration can be described using delays and feedback paths, as can history sensitivity, state, and recurrences. These dependencies may restrict concurrency, but these restrictions are immediately evident without need for subscript analysis. Furthermore, when the iteration is manifest, the numbers of tokens produced and consumed are fixed, as implied in figure 4. Iteration is manifest when the number of cycles to be computed is known at compile time. In this case, the graph is SDF. SDF graphs have two important advantages; first, they can be scheduled at compile time without any loss of concurrency, and second, a consistency check is already known [Lee87a].

The motivation for introducing **SWITCH** and **SELECT** into a language can be given by example. Consider the example shown in figure 5. It is a corrected guarded count function that takes an input token *n*, assumed to be a positive integer, and counts down from that integer to zero. Each new input token restarts the count, and is only consumed after the previous count has been completed, as illustrated in table 1. The actor labeled "-1" simply

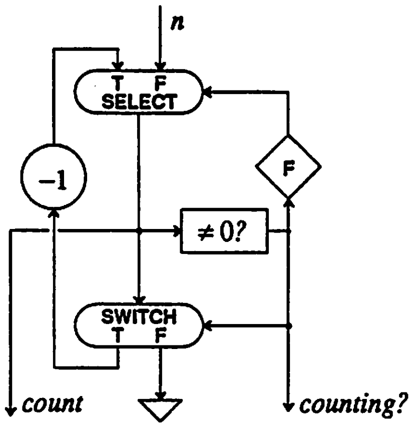


Figure 5. A guarded count. This function takes a non-negative integer input and counts down from that integer to zero.

n	1		3				2			0	1	
count	1	0	3	2	1	0	2	1	0	0	1	0
counting?	T	F	T	T	T	F	T	T	F	F	T	F

Table 1. An example of an input stream *n* and the output streams *count* and *counting?* are shown here. The tokens are aligned to show when output tokens are produced relative to the input tokens consumed.

decrements the value of its input token by one. The actor labeled "$\neq 0?$" tests the value of the input token and outputs a Boolean. The diamond shape represents a delay, which is simply an initial token on the arc, as explained earlier. The label "**F**" indicates that the value of the initial token is a Boolean false. With these actors understood, it should be easy to see by inspection exactly how this function works.

There is nothing difficult about constructing a counting function in any functional language. However, our function returns a token stream "*count*" and a boolean stream "*counting?*", as shown in table 1. Viewed as a macro dataflow actor, the guarded count consumes one input token when it fires, and produces a number of output tokens that depends on the data carried by the input token. In other words, it takes an input stream of any length and outputs a stream of finite streams. Viewed as a function, the number of values returned depends on the value of its argument. This functionality would be difficult in most functional languages, and yet it proves quite useful.

The guarded count is used in figure 6a to build a function, called "last of N", that takes an integer input N, consumes N input tokens from the x stream, and outputs only the last of the N tokens consumed, discarding the rest. This is therefore a function where the number of arguments taken depends on the value of the first argument.

The "last of N" macro dataflow actor is used in figure 6b to construct an iterative program to compute Fibonnacci numbers (a recursive version will be discussed later). A Fibonnacci number is the sum of the two previous Fibonnacci numbers. The two delays (initialized

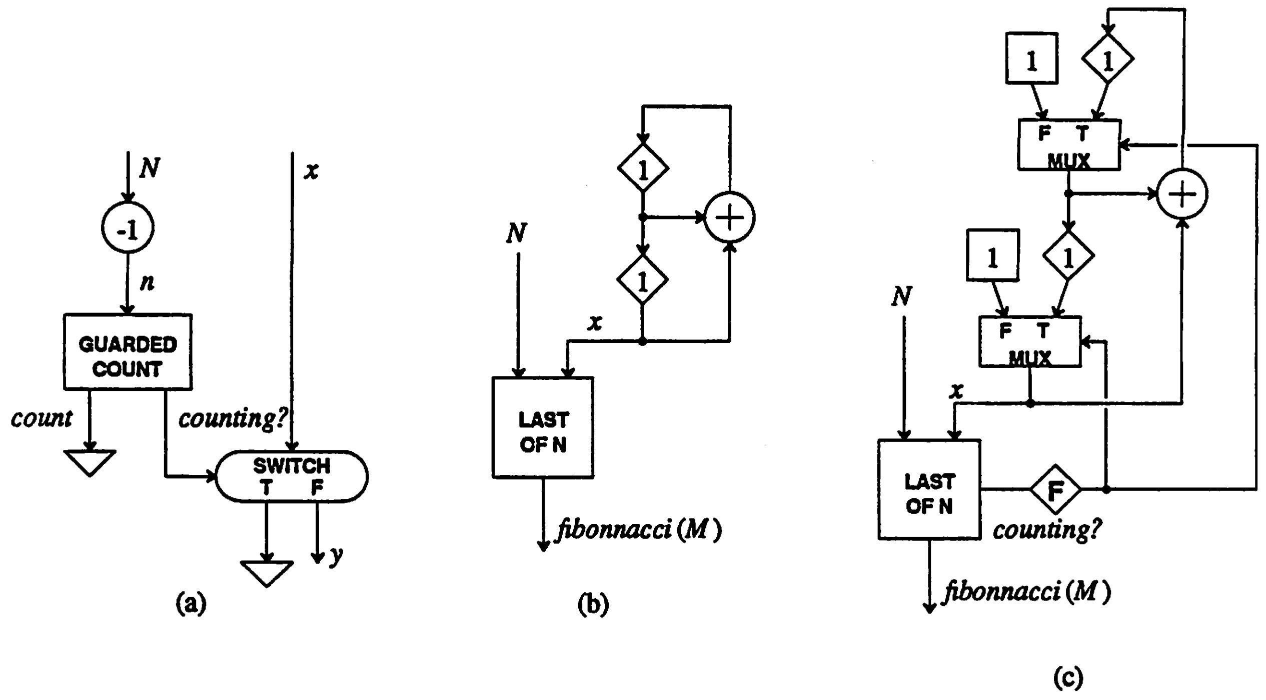


Figure 6. a. The "last of N" system consumes N tokens from its x input path and outputs only the last token. b. An iterative Fibonnacci number function that can only be run once without re-initializing the delays. c. A Fibonnacci number function that can be run indefinitely.

to one) remember the two previous Fibonnacci numbers. The "last of N" actor selects the desired Fibonnacci number from the stream of numbers supplied at its x input. The style of iteration here is similar to that in figure 4, but the iteration is not manifest.

Unfortunately, the function in figure 6b can only be run once because once the initial tokens in the delays are consumed, there is no mechanism for resetting them to one after the Fibonnacci number has been produced. One possible modification that will work for a stream of inputs is shown in figure 6c. In that graph, the boxes labeled "1" put out a token with unity value whenever they fire. The **MUX** (for multiplexer) actors consume a token at each input, discard one, and copy one to the output. With the two delays in the feedback path initialized to unity, the Boolean delay can be initialized to true or false. It is shown initialized to false. It will be re-initialized to false after each computation of a Fibonnacci number because the *counting?* Boolean has value false when the guarded count has finished counting[2].

It would hard to claim that this Fibonnacci implementation is conventional. The control is truly data-driven, via the production and consumption of multiple tokens. A more dramatic example, one that is more difficult to build using established functional languages, is the *ordered merge* shown in figure 7. First, a macro actor called *sort* is explained in figure 7a. This actor accepts two numerical inputs and outputs the maximum on one path and the minimum on the other. It also outputs a Boolean indicating whether or not the ordering of the two inputs was swapped on the outputs. Note that this macro actor is built entirely of *homogeneous* SDF actors [Lee87a], and is a homogeneous SDF macro actor itself. Homogeneous SDF means that exactly one token is consumed and produced on each input and output when

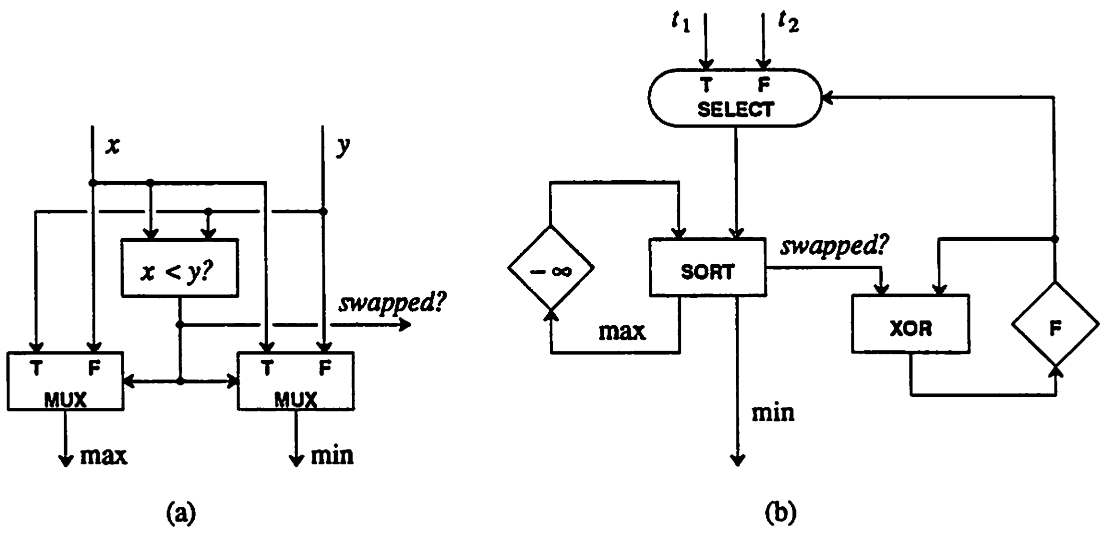


Figure 7. a. This sort macro dataflow actor takes two input tokens when it fires, and outputs the larger one on the max output and the smaller one on the min output. It is built entirely of homogeneous SDF actors. b. The sort actor is used to build an "ordered merge" system, which takes two monotonically increasing token streams and combines them into one monotonically increasing stream.

[2] There is a significant disadvantage to the program shown in figure 6c. Without a detailed analysis of the program semantics, it appears that there are data dependencies between successive Fibonnacci computations, while actually this is not true. One solution to this problem is a "resetting delay", introduced in [Lee89a]. These are delays that revert to their initial value each time a subgraph fires, where the name of the subgraph is a property of the delay.

the actor fires. The same **MUX** actor as in figure 6c is used.

The ordered merge program, shown in figure 7b, uses the *sort* macro actor to merge two monotonically increasing numerical streams into one monotonically increasing stream. The **XOR** actor is an exclusive or. The delay on the right is used to remember which input stream supplied the last token consumed. Whenever the Boolean *swapped?* is true, the next token consumed comes from the opposite stream. The delay on the left remembers the token that was not output the last time. In other words, the **SELECT** takes inputs from t_2 until it gets a token with value exceeding the stored token. Then it stores the t_2 token and takes inputs from the t_1 stream until it again gets a token exceeding the stored token. It continues alternating. The resulting output is the merged stream. Notice that the program is built of very few operations.

Notice that the first output of the ordered merge is $-\infty$. This output can be easily discarded if it is problematic to the downstream system. However, it may be useful. Note that if we design the *sort* actor so that

$$\max(+\infty, +\infty) = -\infty \quad \text{and} \quad \min(+\infty, +\infty) = +\infty, \tag{2}$$

then the *merge* function can take streams of finite streams with $\pm\infty$ marking the beginning and end of each stream[3].

Although all of these examples can be implemented using more conventional techniques, there are some advantages to the representations given here in conciseness and implementation issues. Of course there are aesthetic issues that cannot be resolved by logical discourse, and the reader may rightfully reject this programming style on aesthetic grounds. Even so, it would be difficult to argue that this programming style is not interesting, primarily because of its purely data-driven control. The major difficulty with using it is the possibility

[3] This solution was suggested by David Culler. As before, another way to do this may be with resetting delays, although there are still open problems with such a method [Lee89a].

of introducing sometimes subtle semantic inconsistencies such as those in figure 1.

3. VERIFYING CONSISTENCY

In [Lee87a] it is shown that consistency can be easily checked SDF graphs. The test is generalized here to arbitrary dataflow graphs, thus identifying all the other inconsistencies in figure 1.

3.1. The Token-Flow Model

Consider the actors in figure 8. These are SDF actors plus control actors fashioned after those of Dennis [Den75]. The first one consumes N and M tokens on each input when it fires and produces L tokens on the output. If these numbers are positive integer constants, the actor is an SDF actor. The **SWITCH** actor in figure 8b, by contrast, is not an SDF actor. The

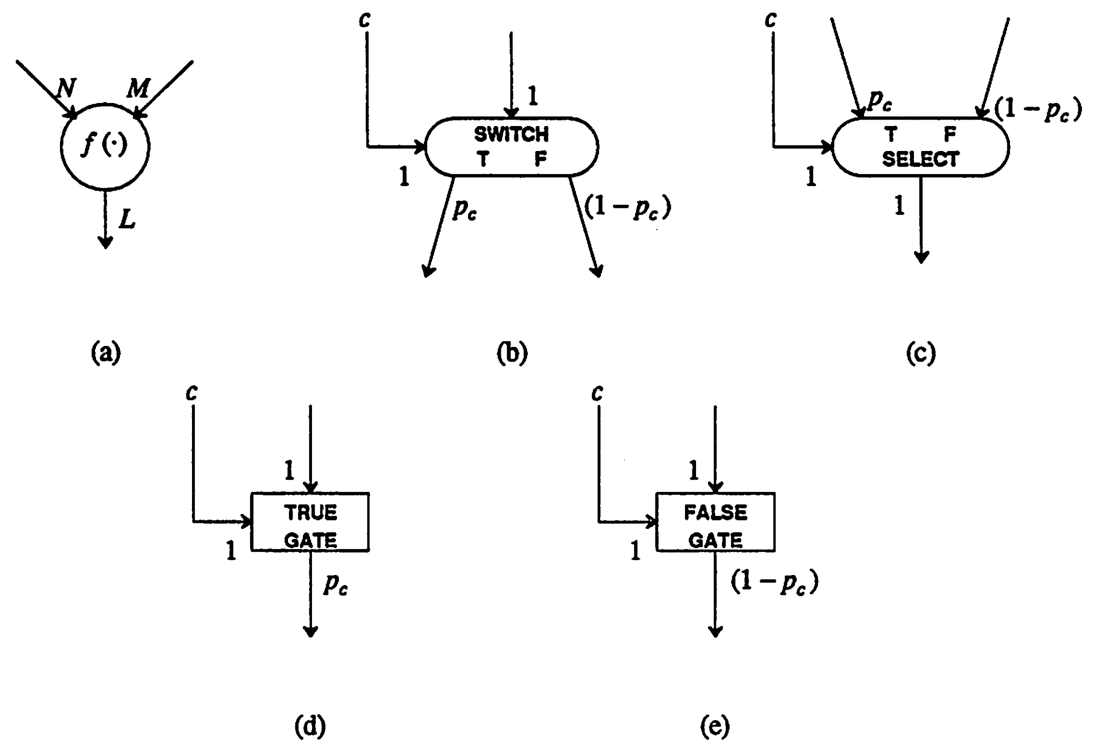


Figure 8. Some dataflow actors are shown together with the expected number of tokens produced or consumed as a function of the probabilities of "true" for each Boolean.

numbers of tokens produced on the two outputs are not constant (they depend on the Boolean input). In figure 1b, the outputs are labeled with the long-term *average* number of tokens produced as a function of the *proportion* p_c of Boolean input tokens c that are *true*. The **SELECT** actor is the complement, in that it consumes a token from one of two input streams depending on a Boolean input. The average number of tokens consumed and produced is similarly shown. The **TRUE GATE** outputs the token from its top input if the Boolean input is "true". The **TRUE GATE** and **FALSE GATE** are merely shorthand for the **SWITCH** when one of the two outputs is discarded.

The proportions used in figure 8 can be interpreted as probabilities in the Bayesean sense, in that they model *uncertainty* about the value of the Boolean token. In other words, a program may be completely deterministic, but we can nonetheless use probabilities to model what the compiler cannot easily discern about the Boolean stream. The appropriate stochastic interpretation is as follows: if a Boolean stream c has proportion p_c of *true* tokens, then a randomly selected token from the stream has *probability* p_c of being true. This seemingly pedantic statement is necessary to avoid implying unjustifiable assumptions about independence of tokens in a stream.

A Boolean stream may be an input to the system, but more likely it is generated by testing non-Boolean data values. The actor performing the test is most likely an SDF actor, but without analyzing its semantics and those of the program that generate its input, the compiler cannot know the probability of "true" for the output of the test. Hence, each such output is given a unique name and the compiler assumes that the value of the output has some unknown probability p_{name} of being true. As we will see shortly, we need not be concerned with the *value* of p_{name}, since all manipulations can be done symbolically.

Some examples of Boolean operators are shown in figure 9. These actors take Boolean inputs and produce Boolean outputs. The probabilities of the outputs being "true" can be

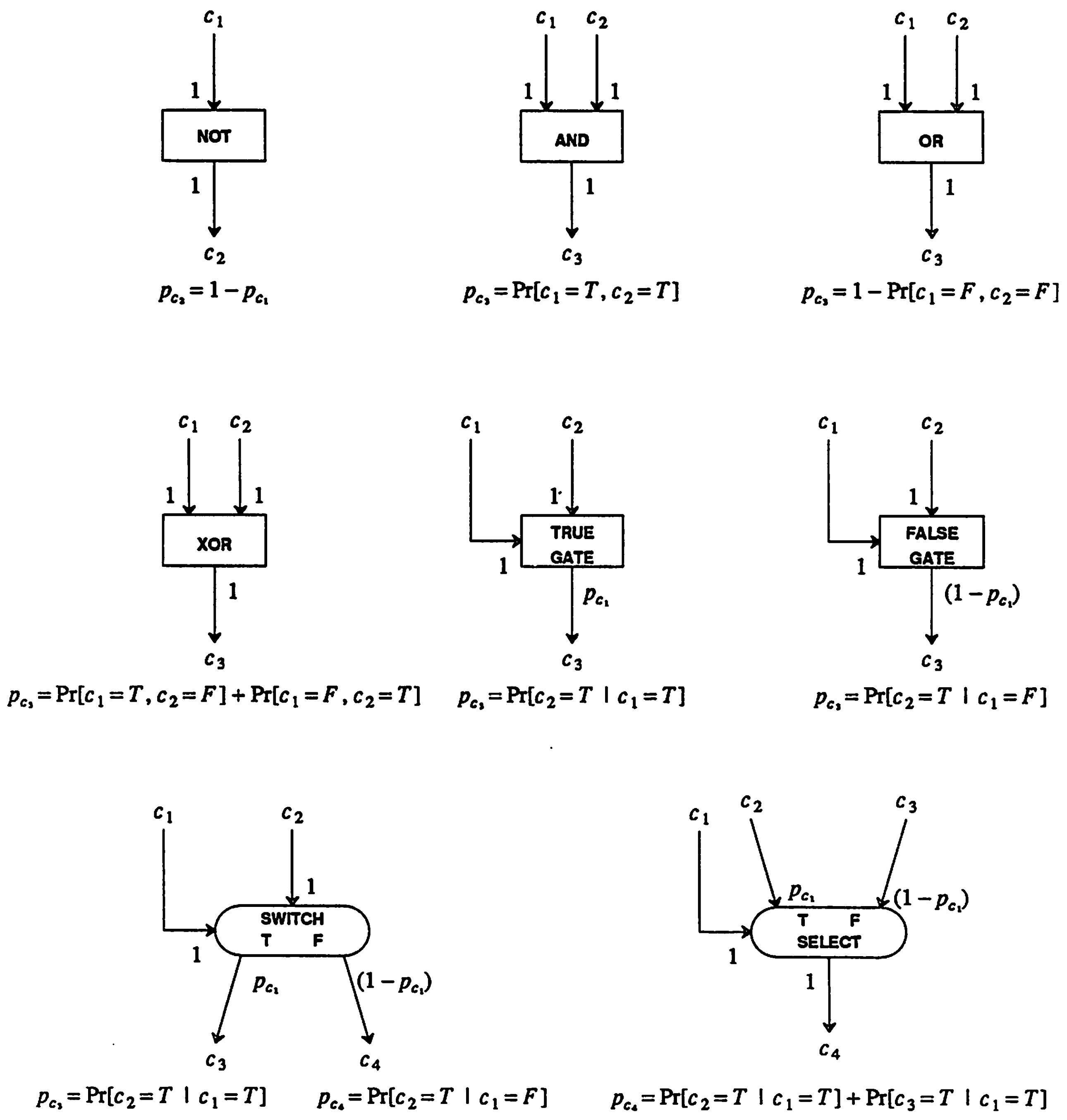

NOT
AND
OR
XOR
TRUE GATE
FALSE GATE
SWITCH T F
T F SELECT
c_1
c_2
c_3
c_4
1
p_{c_1}
$(1-p_{c_1})$
$p_{c_2} = 1 - p_{c_1}$
$p_{c_3} = \Pr[c_1 = T, c_2 = T]$
$p_{c_3} = 1 - \Pr[c_1 = F, c_2 = F]$
$p_{c_3} = \Pr[c_1 = T, c_2 = F] + \Pr[c_1 = F, c_2 = T]$
$p_{c_3} = \Pr[c_2 = T \mid c_1 = T]$
$p_{c_3} = \Pr[c_2 = T \mid c_1 = F]$
$p_{c_3} = \Pr[c_2 = T \mid c_1 = T]$
$p_{c_4} = \Pr[c_2 = T \mid c_1 = F]$
$p_{c_4} = \Pr[c_2 = T \mid c_1 = T] + \Pr[c_3 = T \mid c_1 = T]$

Figure 9. Boolean actors produce new Booleans as a function of input Booleans. The true-probabilities of the outputs are related to the joint and conditional probabilities of the inputs, as shown here.

expressed in terms of joint probabilities of the inputs being true, as shown in figure 9. In the figure, the notation $\Pr[c_1 = T, c_2 = F]$ is the joint probability that c_1 is true and c_2 is false.

These expressions can be used sometimes to determine consistency when Boolean streams are not independent.

3.2. Consistency and Strong Consistency

We will call a dataflow graph *consistent* if on each arc, in the long run, the same number of tokens are consumed as produced. We will call it *strongly consistent* if it is consistent for all possible p_c for each Boolean stream c in the graph.

First, a requirement for consistency or strong consistency is that the graph have the potential to be non-terminating. In other words, it must be free of deadlocks. Deadlocks would occur, for example, when the graph has insufficient delays on a cycle. In [Lee87a], a systematic method for checking for deadlocks is given for SDF graphs. To the author's knowledge, a systematic checking algorithm for more general dataflow graphs is still an open problem. For the purposes of this paper, we will simply assume all graphs are free of deadlocks.

Some perfectly correct, albeit bizarre programs, are consistent, but not strongly consistent. For example, the graph in figure 1b is consistent if c is known to be always true. To show this, we can perform a simple analysis using figure 8. This analysis alludes to a more systematic method that will be developed below.

Since both actors in figure 1b consume exactly one token from the x input stream, both actors should fire the same number of times, in the long run. However, the **SWITCH** only produces an expected p_c tokens on its **T** output, implying that the add actor should fire p_c times as often as the **SWITCH**. These two statements can be reconciled if and only if $p_c = 1$. So we see that the graph in figure 1b is consistent, subject to $p_c = 1$, but not strongly consistent.

The remaining examples in figure 1 can be dispatched similarly. The example in figure 1a has more actors than the one in figure 1b, so we should be more systematic about its analysis. Define q_{SW} to be the proportion of total firings that are firings of the **SWITCH** actor. Similarly, q_{SE} is the proportion of **SELECT** firings, q_N of **NOT** firings, q_f of $f(\cdot)$ firings, and q_g of $g(\cdot)$ firings. Since these are all the actors under consideration,

$$q_{SW} + q_{SE} + q_N + q_f + q_g = 1 . \tag{3}$$

The path of c_1 and c_2 tokens requires that

$$q_{SW} = q_N = q_{SE} . \tag{4}$$

Assuming $f(\cdot)$ and $g(\cdot)$ each consume and produce one token when they fire, then from figure 8 we have

$$\begin{aligned} p_{c_1} q_{SW} &= q_f \\ (1 - p_{c_1}) q_{SW} &= q_g \\ p_{c_2} q_{SE} &= q_f \\ (1 - p_{c_2}) q_{SE} &= q_g . \end{aligned} \tag{5}$$

Combining (4) and (5) we conclude that consistency requires that

$$p_{c_1} = p_{c_2} = 0.5 . \tag{6}$$

Hence this graph is consistent subject to (6), and therefore not strongly consistent.

Applying the same method to figure 1c, consistency requires that

$$\begin{aligned} 2q_U &= q_+ \\ q_U &= q_+ . \end{aligned} \tag{7}$$

Since we also require that

$$q_U + q_+ = 1 , \tag{8}$$

this graph is not consistent.

This method can be made still more systematic, and indeed can be automated. Consider an actor A connected to actor B. Let γ_A denote the average number of tokens produced on the arc for each firing of A. This can be a function of the Boolean proportions in the system.

Similarly, let γ_B be the average number of tokens consumed for each firing of B. If A fires proportionally q_A times, and B fires proportionally q_B times, then consistency requires that

$$\gamma_A q_A = \gamma_B q_B \,. \tag{9}$$

Collecting one such equation for each arc in the graph, we get a system of equations that can written compactly using matrix notation,

$$\Gamma \mathbf{q} = \mathbf{0} \,. \tag{10}$$

The vector **q** specifies the proportion of times each actor fires, and should therefore be normalized,

$$\mathbf{1}^T \mathbf{q} = 1 \,, \tag{11}$$

where $\mathbf{1}^T$ is a column vector full of 1's. The matrix Γ has one row for each arc in the graph and one column for each actor. Each row has two entries, the average number of tokens produced, and the negative of the average number of tokens consumed. Composed with this system may be a set of relationships between the Boolean probabilities in the system, when such relationships are known. Examples of such relationships are shown in figure 9. Consistency requires that there be a solution to (10) and (11) for some set of Boolean proportions consistent with the constraints. Strong consistency requires that there be a solution to (10) and (11) for any set of Boolean proportions consistent with the constraints.

We first illustrate the systematic method by proving that the attempted guarded count of figure 1d is not consistent. The graph has been reduced to its essentials in figure 10a, where the three actors have been numbered, as have the arcs connecting them. To simplify things slightly, the select is shown duplicating its output, to avoid adding a "fork" actor to the analysis.

Notice that the delay in figure 1d has been omitted in figure 10a. Recall that a delay is an initial token on an arc, with value "false" in this case. The delay is not an actor. In steady state, the proportion p_c of true's at the output of the delay is the same as the proportion at the

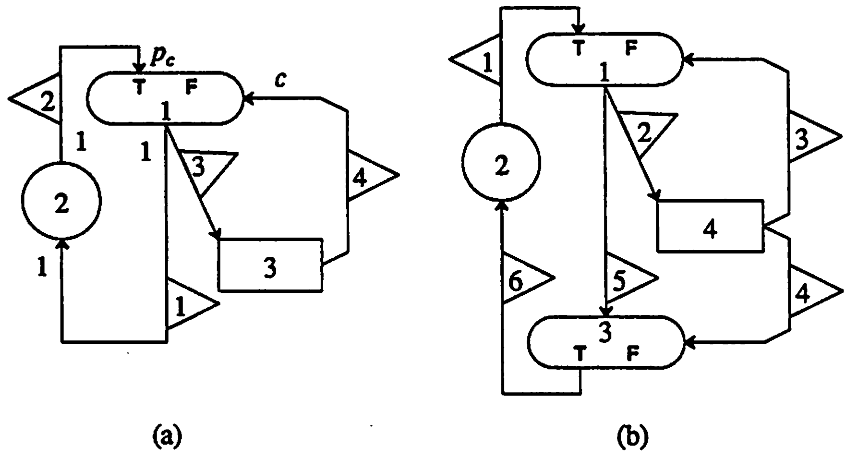


Figure 10. a. The essentials of the example in figure 1d with actors and arcs numbered for systematic analysis. b. A similar representations of the guarded count in figure 5.

input, so the delay can be omitted in the analysis.

The topology matrix for figure 10a is

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ -p_c & 1 & 0 \\ 1 & 0 & -1 \\ -1 & 0 & 1 \end{bmatrix} . \qquad (12)$$

From this we can see that (10) and (11) have a solution if and only if $p_c = 1$.

To make the conclusion even more obvious, notice that (10) implies that the rank of Γ is smaller than s, the number of actors in the graph (equal to the number of columns of the matrix). It is easy to see that (12) has rank 3, equal to s, unless $p_c = 1$. In that case, the rank is 2, or $s - 1$. It can be shown that for any connected graph, the rank must be at least $s - 1$ (see [Lee87a]), so we conclude that consistency requires that

$$rank[\Gamma] = s - 1 . \qquad (13)$$

The systematic method can also be illustrated by proving that the guarded count in figure 5 is strongly consistent. The outline of the guarded count is shown in figure 10b with each actor and each pertinent arc numbered. There are six arcs and four actors. With this number-ing,

$$\Gamma = \begin{bmatrix} -p_c & 1 & 0 & 0 \\ 1 & 0 & 0 & -1 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & p_c & 0 \end{bmatrix} . \tag{14}$$

In this matrix, p_c is the proportion of tokens in the "*counting?*" Boolean signal that are *true*. This matrix has rank 3, or $s - 1$. By inspection,

$$q = \frac{1}{3+p_c} \begin{bmatrix} 1 \\ p_c \\ 1 \\ 1 \end{bmatrix} \tag{15}$$

satisfies (10) and (11). This is true for any p_c, so the graph is *internally* strongly consistent.

The qualifier "internally" is used because we are ignoring the rest of the system, to which the guarded count is connected. To consider the behavior of the guarded count in a larger system we can consolidate its properties into macro "guarded count" actor, as shown in figure 11. This approach is consistent with using hierarchy to control complexity. The labels in figure 11a indicate that for every $\frac{1-p_c}{3+p_c}$ tokens consumed, there will be $1/(3+p_c)$ "*count*"

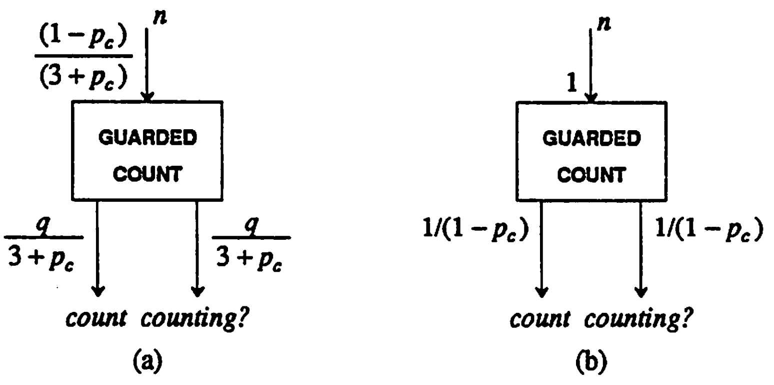


Figure 11. The summarized token-flow model for the guarded count, shown with two equally valid sets of "average number tokens consumed and produced".

and "*counting?*" tokens produced. These expressions are obtained by multiplying the entry q_x in (15) by the average number of input (output) tokens consumed (produced) by each actor x connected to the outside. Notice, however, that the analysis of a system using the guarded count (or any other actor) is not affected by multiplying the average numbers of tokens produced and consumed by any constant. In other words, the token-flow model for any actor is not unique. Equivalently, any row of Γ can be scaled without affecting its rank. Multiplying the expressions in figure 11a by $\frac{3+p_c}{1-p_c}$, we get the more intuitive expressions in figure 11b.

The same technique can be applied to the examples in figure 6 and figure 7, getting the consolidated results shown in figure 12.

4. DISCUSSION AND EXAMPLES

The interpretation of the solution q to (10) and (11) as a vector of proportions of firings of each actor is useful for developing intuition. Consider figure 13a. The top arc implies $q_A = q_B$, meaning that actors A and B should fire the same number of times. Furthermore, it is immediately evident that the program in figure 13a is consistent if and only if $p_s = 0.5$. By contrast, the program in figure 13b is strongly consistent.

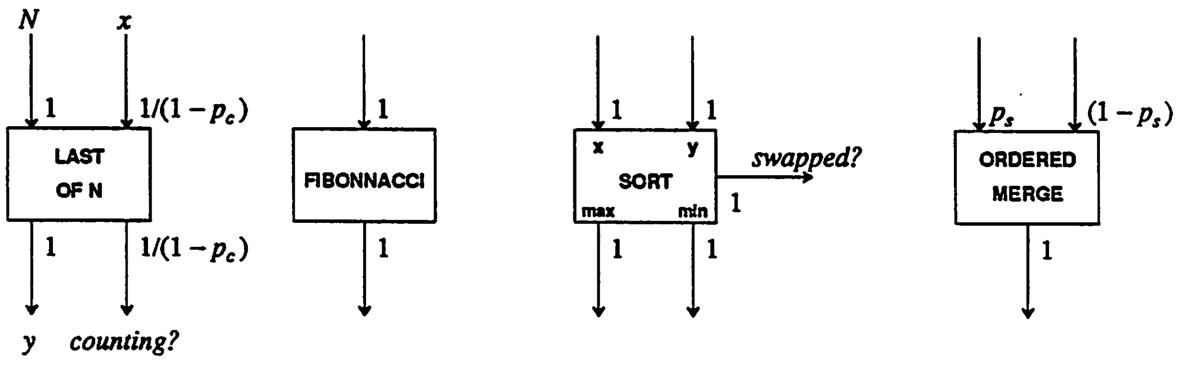


Figure 12. Consolidations of the systems shown in figure 6 and figure 7.

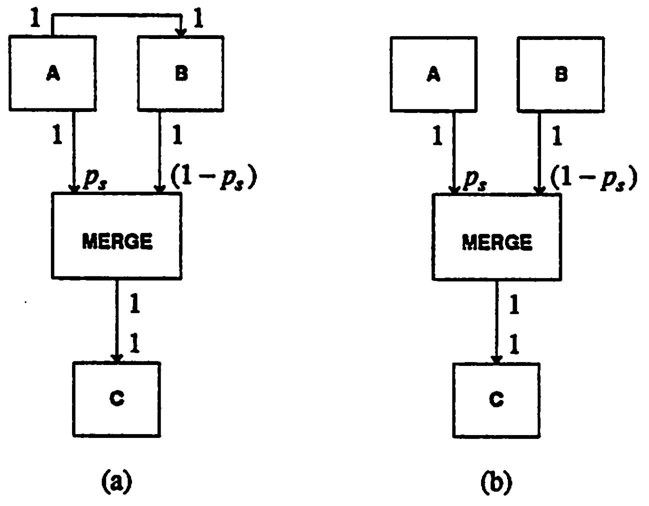


Figure 13. a. A graph that is consistent for $p_s = 0.5$. b. A graph that is strongly consistent.

A particularly interesting example is shown in figure 14. Only figure 14c is strongly consistent. Applied to figure 14a, our method reveals that it is consistent if and only if

$$p_{c_1} = p_{c_2} p_{c_1} , \tag{16}$$

which occurs only if $p_{c_2} = 1$ or $p_{c_1} = 0$. It is easy to verify by inspection that these two situa-

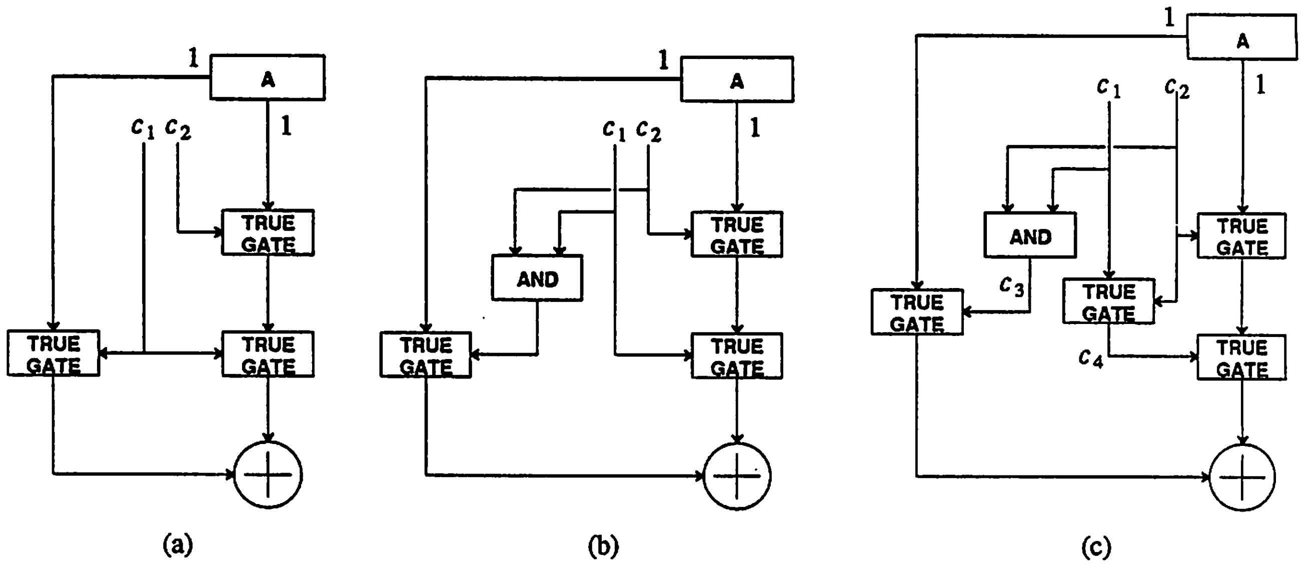


Figure 14. The programs in (a) and (b) are consistent only for $p_{c_2} = 1$. The program in (c) is strongly consistent.

tions lead to consistent graphs, but also to semantics that are probably not intended. The program in figure 14b is an attempt to correct the problem. However, it also fails the strong consistency test because the **AND** actor synchronizes the c_1 and c_2 token streams, constraining the rates of tokens to be the same. Applying our systematic method to the program in figure 14c we get consistency if and only if

$$p_{c_3} = p_{c_2} p_{c_4} . \tag{17}$$

From the relationships in figure 9,

$$p_{c_3} = \Pr[c_1 = T, c_2 = T] \quad \text{and} \quad p_{c_4} = \Pr[c_1 = T \mid c_2 = T] . \tag{18}$$

Hence condition (17) is equivalent to

$$\Pr[c_1 = T, c_2 = T] = p_{c_2} \Pr[c_1 = T \mid c_2 = T] , \tag{19}$$

which is always true by the multiplication rule in probability! Consequently, this graph is strongly consistent.

Consistency can also be checked when recursion is used. Recursion in dataflow graph languages is represented using self-referential hierarchy. In figure 15, for example, the path of the c Boolean indicates that $q_{SW} = q_{SE}$, so $x = y$, where x and y are the number of tokens consumed and produced by the **FIBONNACCI** function. Since any $x = y$ will yield the same consistency result, we can set $x = y = 1$. Showing that figure 15 is consistent then becomes trivial.

5. DATAFLOW GRAPH LANGUAGES

Languages used to program dataflow machines, such as Val [Mcg82] and ID [Nik88][Arv87]. do not currently have the same sort of semantics of the programs given above, although an early version of ID had some of the features. To distinguish our style of programming, we refer to a dataflow *graph* language as one with the semantics of a dataflow graph. Distinguishing features of a dataflow graph language are the use of streams as the essential data structure, the. ability of actors to consume or produce multiple tokens, and a

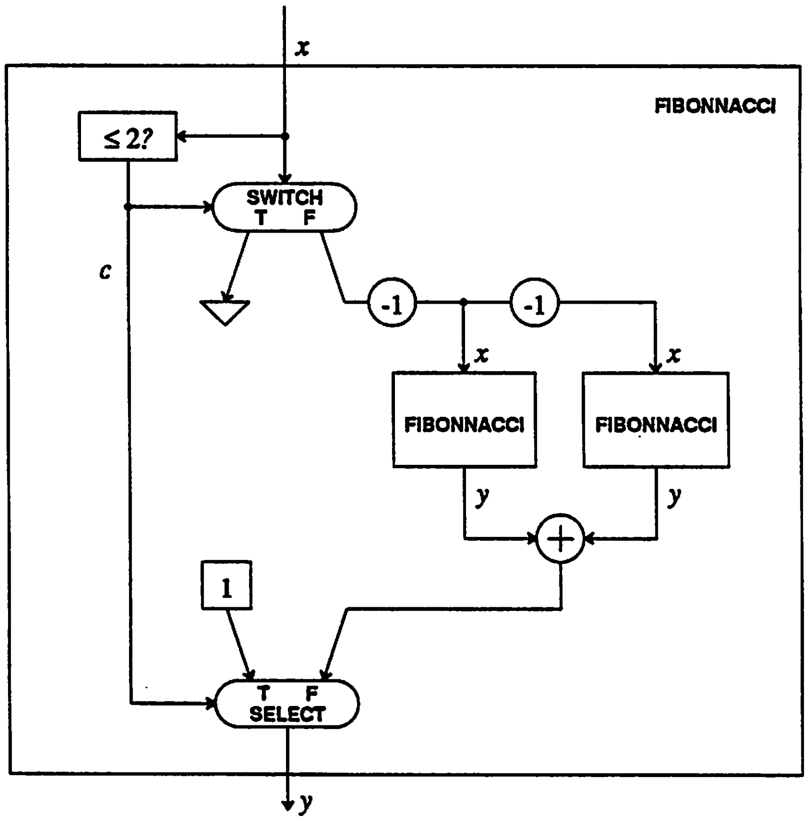


Figure 15. A recursive dataflow graph for computing Fibonnacci numbers. It can be analyzed using the token-flow model.

strict locality of semantics. The latter means that every operator in the language is a dataflow actor, and not a higher level construct (such as if-then-else) with semantics that can extend over a large part of a dataflow graph. Subsets of the class of dataflow graph languages have been explored before; a key reference that includes many pointers to the earlier literature is [Dav82].

It is not fundamentally important whether the syntax of the language is graphical or textual; all of the above examples could be given in a textual syntax without altering the semantics. However, although graphical programming is a controversial topic, the techniques described in this paper may help make it truly attractive for some applications. I ask the reader to keep an open mind, even if he or she comes with a predisposition against it. In this paper we represent all program constructs graphically, hoping that the reader will decide that this enhances the clarity considerably, compared with an equivalent textual description.[4] Hierarchy is used liberally to control complexity.

Conventional wisdom has it that the principal motivation for working with dataflow graphs is to exploit parallelism. While this is an important motivation, it provides no incentive for languages that directly manipulate dataflow graphs. Indeed, standard practice is to design functional, applicative, or single-assignment languages which are translated into dataflow graphs by a compiler. The programmer does not interact directly with the dataflow graph.

In general-purpose computation, there are good reasons for avoiding dataflow graph languages. One reason is that semantic inconsistencies such as those in figure 1 can be prevented by the syntactic constraints of the language. For example, the program in figure 1a cannot be expressed conveniently using the functional if-then-else of (1). Although **SWITCH** and **SELECT** are used implicitly in (1), the control inputs of the two actors are constrained by the syntax of the language to be identical. Such syntactic constraints also ensure that iteration is accomplished without inconsistencies.

Constructs such as if-then-else and do-while can be viewed as "graph constructors", since they are translated into dataflow graphs by a compiler. In some modern work with functional languages, an if-then-else is alternatively a function with arity three, the first argument of which is a Boolean, and the second two of which are functions. The value returned is a function, which can then applied to the data. The operation (1) could therefore be written:

[4] Arvind, of MIT, has suggested an interesting experiment. Design a programming language and environment that simultaneously maintains a graphical and textual description of a program, and permit the programmer to modify either one. Such an experiment would reveal a great deal about programmer preferences, probably indicating that for some types of programs, the programmer will prefer to work with the graphics, and for others with the text.

$$y = \text{if–then–else}(c\,, f\,(\cdot), g\,(\cdot))\,(x)\,. \tag{20}$$

The if-then-else function is first applied to a subset of its arguments, and the function returned is applied to x. The general form of such functions are said to be *curried* [Arv88]. Curried functions can be part of a dataflow graph language. The construct (20) is more directly a dataflow graph description than (1), but dataflow tokens must now be able to carry functions. One way to view this innovation, therefore, is as a mechanism for getting closer to a dataflow graph description without the hazards of semantic inconsistencies. The if-then-else function in (20) is a homogeneous SDF actor, meaning that it requires exactly one token on each input to fire, and it produces exactly one token on each output (the output of the if-then-else is a function). Homogeneous SDF actors are the best behaved of all, since they cannot lead to semantic inconsistencies of the type discussed in this paper. (This can be easily proven using the mathematics developed above.)

A second reason to avoid programming with dataflow graphs is that they are unfamiliar to programmers. Functional languages can be made reasonably familiar using if-then-else, do-while, and similar constructs found in imperative languages. Although the semantics of these constructs are slightly different in functional languages, their basic operation is similar, so a programmer can more easily learn to use them.

Keeping these valid objections in mind, there are nonetheless compelling reasons for dataflow graph languages. First, in certain application domains, particularly signal processing, the most natural data structure is the stream. Operators on streams are best viewed as dataflow actors. For this reason, throughout the 30 year history of digital signal processing, countless "block-diagram languages" have been developed by the signal processing community. For an example and a partial list of other examples, see [Lee89b]. Block-diagram languages are essentially large-grain dataflow languages [Bab84], often with graphical syntax, and they form much of the motivation for this paper. The author intends to apply the technique described here to the programming environment described in [Lee89b].

In the past, streams have been introduced to functional languages; for example, Weng defined operators *first*, *rest*, *cons*, and *empty* (a predicate) for building functions that operate on streams [Wen75]. The function *first* yields the first value in a stream, while *rest* yields the rest. The *cons* operator takes as arguments an elementary value and a stream and returns a stream with the elementary value prepended. This is identical to our delay. These functions would be familiar to Lisp programmers, but foreign to the signal processing community, for example. They do not reflect the notion of "flow" of values which is so natural when thinking of streams.

A second motivation is provided by the examples given above, particularly the ordered merge. An equivalent function in an established functional language would be far more complicated. Of course, it could be added as a primitive, like the if-then-else, but other examples would arise.

It has been observed that non-determinate actors should be added to dataflow languages in order for programs to interact with multiple external events, for example transaction processing, interprocess communication, or interaction with external hardware [Kos78]. A non-determinate merge has been proposed by Arvind and Brock [Arv84], and resource managers have been built using it. The non-determinate merge is an "unordered" merge where token streams can be merged in unpredictable ways, depending on the time of arrival of tokens. The token-flow model can be used to verify consistency of programs using such a merge, precisely because the token-flow model does not require knowledge of the expected number of tokens consumed. The token-flow model for the unordered merge is the same as that for the ordered merge shown in figure 12. But for some applications, the ordered merge can be used as effectively as an unordered merge. Often, transaction processing should not be strictly speaking non-determinate, but rather should use information not normally available in a dataflow program: the absolute time of occurrence of an event. An airline reservation system, for example, should probably grant a seat to the first request (in real time) for it, and not to some random request. Such fairness can be ensured by adding a time stamp to each request and using the (determinate) ordered merge to sort the requests. The resulting program is determinate. The main drawback with this approach is that the ordered merge may introduce time delay. For applications where this time delay is objectionable, further enhancements are required, and a non-determinate approach may be preferred. In *either case*, a dataflow graph language is preferable to a functional language, and the token-flow model can be used to verify consistency.

The third justification for dataflow graph languages is an aesthetic one. Some programs can be be expressed very elegantly in such languages, and the exercise of rethinking algorithms to express them in such languages can be both enjoyable and enlightening. One example of such an elegant program is the ordered merge of the previous section.

A fourth justification currently applies only to SDF graphs, and consequently is only valid for applications, such as signal processing, where programs are reasonably static. SDF graph languages can be more effectively analyzed for concurrency *at compile time* than established functional languages. An example already discussed is shown in figure 4. Most established functional languages use a loop counter to implement all iteration, manifest or not, and a compiler that wishes to detect concurrency must analyze the semantics of the operations on the loop counter; this is not an easy task. To detect inter-cycle dependencies, in some languages it must do subscript analysis, which is also not easy.

To summarize, The graph constructor or curried function points of view have been used to match dataflow semantics with practical languages. In either case, one objective is to impose *syntactic constraints* that prevent the semantic inconsistencies that could arise from arbitrarily connecting dataflow actors. However, through the use of the token-flow model, a compiler can impose looser constraints by checking for consistency, leaving the programmer free to build perfectly correct programs such as the ordered merge. Another objective is to maintain familiarity in the languages. However, for some applications such as signal processing, dataflow graph languages are more natural. Moreover, dataflow graph languages can express certain programs, such as the ordered merge in figure 7, very elegantly. Finally, although it is easier to exploit concurrency in functional languages than in imperative languages, it is easier still in some dataflow graph languages. In particular, the available concurrency can be more easily recognized at compile time.

6. COMPARISON WITH HDS THEORY

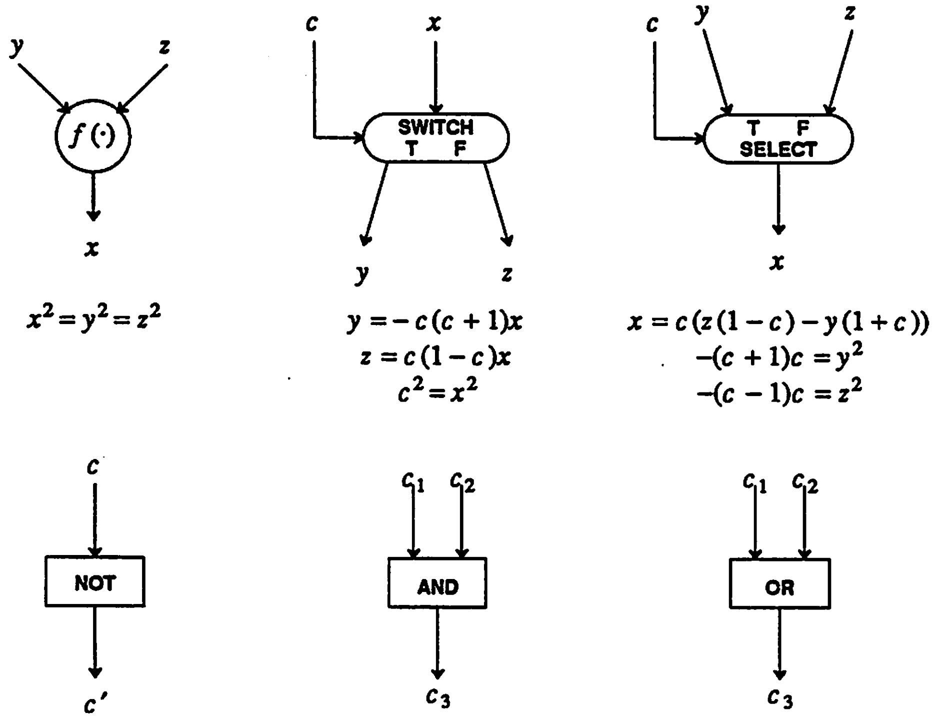
The method given in this paper was inspired by the algebraic techniques of Benveniste, et. al. [Ben88]. These methods have been applied to a language called SIGNAL which is not a dataflow graph language (the differences will be described shortly). The SIGNAL language is used to describe *Hybrid Dynamical Systems* (HDS), and the algebraic method is called model Ω. Model Ω inspired the token-flow model because it systematically identifies inconsistencies similar to those found in dataflow graph languages.

Model Ω works as follows. A signal consists of a sequence of *slots*. Each slot is assigned an indicator -1, 0, or +1, where -1 refers to a Boolean false, 0 refers to the absence of a value (*bottom*), and +1 refers to a Boolean true. Any number of zeros can appear between two non-zeros without changing the meaning of a signal. Given this model, the input/output relationships of operators can be described algebraically, where the algebra is performed in a finite field (or Galois field) with three elements, -1, 0, and +1. This simply means that modulo arithmetic is used. In this algebra, for example, the following statements are true:

$$\begin{aligned} x + x &= -x \\ x^3 &= x \\ x^4 &= x^2 \end{aligned} \quad . \qquad (21)$$

Non-Boolean values are simply assigned an indeterminate ±1. Hence, a slot from the non-Boolean signal x either has a value (in which case $x^2 = 1$), or does not ($x^2 = 0$). In other words, only the signal *clock* is represented for non-Booleans.

Given this framework, functions similar to the dataflow actors we have been using can be described as shown in figure 16. The relationships between the input and output signals within the algebra are shown. To understand these, consider the **SELECT** function, which has three relationships describing its behavior. Considering the first relationship, notice that if $c = 0$ (the control signal is absent) then $x = 0$ (the output is absent). If $c = -1$ (the control signal is false) then $x = z$. If $c = 1$ (the control signal is true) then $x = y$, using (21). This is precisely the behavior we expect from the **SELECT** function. However, the description is



$$x^2 = y^2 = z^2$$

$$y = -c(c+1)x \qquad z = c(1-c)x \qquad c^2 = x^2$$

$$x = c(z(1-c) - y(1+c)) \qquad -(c+1)c = y^2 \qquad -(c-1)c = z^2$$

$$c' = -c$$

$$c_3 = c_1c_2(c_1+1)(c_2+1)$$

$$c_3 = -c_1c_2(c_1-1)(c_2-1)$$

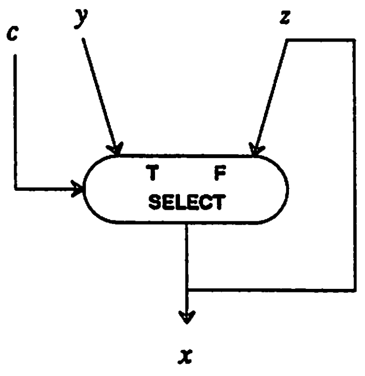
Figure 16. The model Ω input/output relationships of certain functions in a hybrid dynamical system are shown here. All relationships are expressed in a Galois field with three elements, -1, 0, and +1.

still not complete. Suppose $c = 1$ and $y = 0$. The dataflow version of the **SELECT** does not fire if y is absent and c is true. To get similar behavior in the HDS function, we impose two more constraints on the input/output behavior of the **SELECT**. The second relationship shown in figure 16 requires that $y^2 = 1$ if $c = 1$ and the third requires that $z^2 = 1$ if $c = -1$.

Another example, the $f(\cdot)$, a homogeneous SDF actor, has the relationship $x^2 = y^2 = z^2$, indicating that if one input is present (say $y^2 = 1$), then the other input must be present $z^2 = 1$, and the output will be present $x^2 = 1$. The reader may have already noticed the key difference between this model and the dataflow model. Specifically, it is not permitted for one input of $f(\cdot)$ to be present unless the other input is also present. In other words, the arcs connecting functions do not have implicit FIFO queues, as they do in the dataflow model. Expressed yet a third way, the production of a data value on an arc is simultaneous with its consumption at the destination. In principle, a dataflow model can be built using HDS by defining a function with the behavior of a FIFO queue, and inserting this function between every pair of dataflow actors. Hence, HDS can be viewed as a lower-level description.

Using HDS to represent dataflow semantics, however, is not very attractive because the complexity is considerable. Instead, we will use HDS in its simplest form, and we will use model Ω to identify inconsistencies that are similar in flavor to those we have previously identified in dataflow graphs. Unlike the dataflow inconsistencies, the HDS inconsistencies do not lead to unbounded token buildup, since there is no queueing of tokens on arcs. Instead, they lead to constraints on Boolean signals, or to absent outputs, or to contradictions in algebraic relationships.

Consider the example in figure 17. Under the dataflow model, this graph is inconsistent unless $p_c = 0$. Applying model Ω to this graph, we get the relationships shown in the figure. These four relationships can be solved to get a constraint on c, namely



$$x = z$$
$$x = c(z(1-c) - y(1+c))$$
$$-(c+1)c = y^2$$
$$-(c-1)c = z^2$$

Figure 17. An example of a program that is inconsistent under either the HDS model or the dataflow model.

$$c = -c^2. \qquad (22)$$

This constraint is satisfied if either $c = 0$ (nothing happens), or $c = -1$ (the control input is false). Hence the conclusion is the same as for the corresponding dataflow graph.

For the graph in figure 1a, interpreted as an HDS program, model Ω reveals that $y = 0$, or the output is always absent. This is not quite the same conclusion as for the corresponding dataflow program, which is consistent if $p_c = 0.5$. The difference is entirely due to the FIFO behavior of arcs in the dataflow model. For the graph in figure 1b, model Ω reveals that when $x^2 = 1$ (the x input is present) then $c = 1$ (the control input is true). This is the same conclusion as that of the token-flow model, which requires that $p_c = 1$.

The examples in figure 1c and d are much more complicated to analyze using model Ω. The reason is that the relationships shown in figure 16 are instantaneous input/output relationships. They do not model dynamics. In model Ω, delays represent state variables, and a finite-state machine models the program dynamics. The HDS equivalent of the **UPSAMPLE** function in figure 1c introduces a state variable into the system, as does the delay in the feedback path of figure 1d. Although analysis is possible in this framework, it is not simple, and is beyond the scope of this paper. The reader is referred to [Ben88]. The comparative simplicity of the token-flow analysis of dataflow programs with dynamics should be viewed as one of its major advantages.

7. COMPARISON WITH SIGNAL

In [Ben88], the language primitives used are not the same as those in figure 16. Instead, a language called SIGNAL is defined with more basic primitives that can be used to construct the functions shown in figure 16. In [Ben88], the language is described using only a textual syntax, but to try to make a connection with dataflow graph languages, we will use an equivalent graphical syntax here. The two basic decision-making operators are shown in figure 18 along with their model Ω input/output relationships. The **WHEN** operator is exactly like the "true" side of the **SWITCH** operator in figure 16. The **DEFAULT** operator is much more subtle, and fundamentally more versatile (and dangerous) than anything in figure 16. As the reader can see from the model Ω input/output relationship, if the input x is present, then $y = x$. If the input is absent, then $y = d$. Like the unordered merge discussed above, this operator is not determinate. Put another way, its semantics depend on the context. Consider the program in figure 19a. Two unconnected subsystems supply input signals to a **DEFAULT** operator. Without further information about the relative timing of the two input signals, any permutation of values can result at the output of the **DEFAULT**. The program in figure 19b has a related feature (or problem, depending on your perspective). Any number of values can circulate around the loop before a value from the d input is processed. It is typical, therefore,

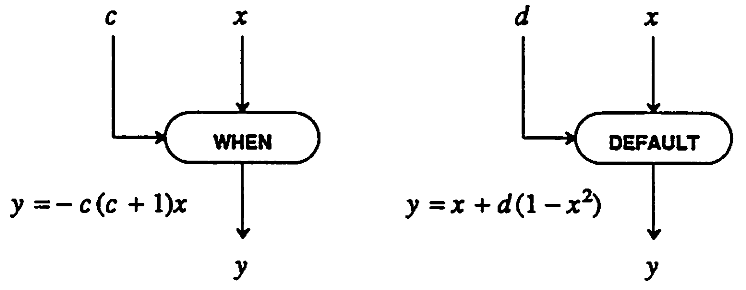


Figure 18. The two basic decision-making operators of the SIGNAL language [Ben88] are shown along with their model Ω input/output relationships.

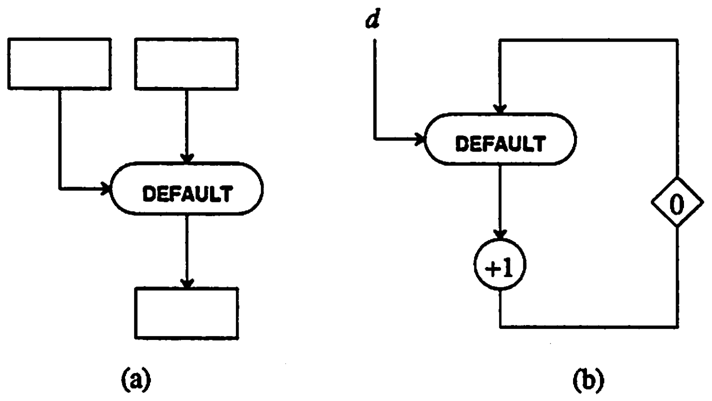


Figure 19. Two non-determinate programs built with the **DEFAULT** operator of the SIGNAL language.

of SIGNAL programs to specify additional synchronization constraints not modeled by the flow of data.

The SIGNAL version of a guarded count, borrowed from [Ben88], is shown in figure 20. The textual version of this program is:

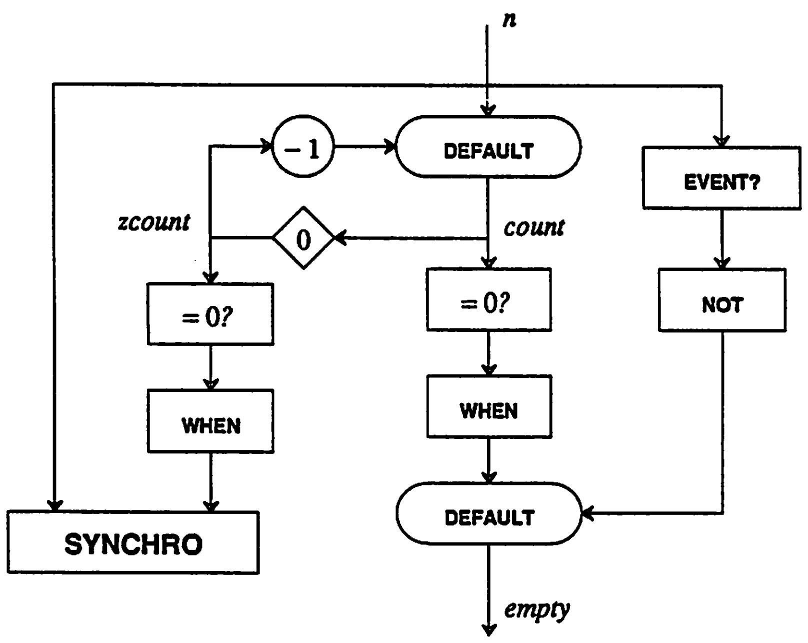


Figure 20. A version of the guarded count in the SIGNAL language.

```
GUARDED_COUNT { ? integer n ! bool empty }
        =
        count = n DEFAULT (zcount - 1)
        zcount = count$ INIT 0
        empty = WHEN(count = 0) DEFAULT (NOT EVENT(n))
        SYNCHRO (WHEN(zcount = 0)),n
end
```

To understand this program, first note that its input is named *n* and output is named *empty*. This is not the same output as the guarded count in figure 5, but it would be easy to modify this program so that the outputs are the same. The dollar sign is used to represent a delay, with the initial value given explicitly. The **EVENT** operator outputs a Boolean true whenever the input is present. The monadic **WHEN** operator outputs a Boolean true whenever the input is present *and true*. The **SYNCHRO** operator is more subtle. It is required because of the non-local semantics of the **DEFAULT** operator. It states that the input *n* can only be present when previous count is finished, or **WHEN**(*zcount* = 0). This statement overcomes the non-determinacy of the **DEFAULT** operator by synchronizing the signals on opposite sides of it.

Two conclusions can be drawn from these examples. First, the SIGNAL language is more fundamental, and therefore harder to work with than our dataflow graph language. The requirement for explicit synchronization accounts for most of the difficulty. Second, since it includes a non-determinate operator (**DEFAULT**), it is more flexible. This will probably prove important if the language is used to describe physical systems. However, such an operator can be used to construct programs that are incorrect in very subtle ways. Functions like the ordered merge are far less dangerous. Nonetheless, further investigation of this issue is warranted.

8. CONCLUSIONS

Consistency and strong consistency in dataflow graphs has been defined, and a systematic consistency test has been developed. It has been compared to similar tests applied to hybrid dynamical systems by Benveniste, et. al. [Ben88]. Dataflow graph languages have been discussed, with numerous programming examples given, and the language model has been compared to the HDS model and the SIGNAL language of [Ben88].

9. ACKNOWLEDGEMENTS

Numerous discussions helped solidify the arguments given in this paper. The author would like to acknowledge significant contributions of ideas and criticism by Arvind (MIT), David Culler, Soonhoi Ha, Abhiram Ranade (Berkeley), Rhonda Righter (Univ. of Santa Clara), and Vason Srini (Berkeley). In addition, Paul LeGuernic of INRIA was kind enough and patient enough to explain to me the basics of model Ω and the SIGNAL language, very inspiring work.

10. REFERENCES

[Arv82]
Arvind and K. P. Gostelow, "The U-Interpreter", *Computer*, 15(2), February 1982.

[Arv84]
Arvind and J. D. Brock, "Resource Managers in Functional Programming", *J. of Parallel and Distributed Computing*, 1(5-21), 1984.

[Arv86]
Arvind and D. E. Culler, "Dataflow Architectures", Computation Structures Group Memo TM-294, 1986, Lab. for Computer Science, MIT, 545 Technology Sq., Cambridge, MA 02139.

[Arv87]
Arvind, R. S. Nikhil, and K. K. Pingali, "ID Nouveau Reference Manual", Computation Structures Group Memo, April 24, 1987, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

[Arv88]
Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture", Computation Structures Group Memo 271, Laboratory for Computer Science, 545 Technology Square, Combridge, MA 02139, June 20, 1988.

[Bab84]
R. G. Babb, "Parallel Processing with Large Grain Data Flow Techniques", *Computer*, July, 1984, 17(7).

[Ben88]
A. Benveniste, B. Le Goff, and P. Le Guernic, "Hybrid Dynamical Systems Theory and the Language 'SIGNAL'", Research Report No. 838, April 1988, Institut National de Recherche en Informatique et en Automatique (INRIA), Domain de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France.

[Dav78]
A. L. Davis, "Data Driven Nets: A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems", Technical Report UUCS-78-108, Dept. of Computer Science, Univ. of Utah, Salt Lake City, Utah, 1978.

[Dav82]
A. L. Davis and R. M. Keller, "Data Flow Program Graphs", *Computer*, 15(2), February, 1982.

[Den75]
J.B. Dennis, "First Version Data Flow Procedure Language", Technical Memo MAC TM61, May, 1975, MIT Laboratory for Computer Science.

[Den80]
J. B. Dennis, "Data Flow Supercomputers", *Computer*, 13(11), Nov., 1980.

[Kos78]
P. R. Kosinski, "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs", *Conf. Record of the 5th Ann. ACM Symp. on Principles of Programming Languages*, Tuscon, AZ, 1978.

[Lee87a]
E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Trans. on Computers*, January 1987, C-36(2).

[Lee87b]
E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow" *IEEE Proceedings*, September, 1987.

[Lee89a]
E. A. Lee, "Static Scheduling of Dataflow Programs: How Far Can It Go?", to appear in *Dataflow: A Status Report*, ed. L. Bic and J.-L. Gaudiot, Prentice-Hall, 1990.

[Lee89b]
E. A. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP", IEEE Trans. on Acoustics, Speech, and Signal Processing, Nov., 1989.

[Mcg82]
J. R. McGraw, "The VAL Language: Description and Analysis", *ACM Trans. on Programming Languages and Systems*, 4(1), pp. 44-82, January 1982.

[Nik88]
R. S. Nikhil, "ID Reference Manual", Computation Structures Group Memo 284, August 29, 1988, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

[Wen75]
K.-S. Weng, "Stream-Oriented Computation in Recursive Data Flow Schemas", Laboratory for Computer Science (TM-68), MIT, Cambridge, MA 02139, Oct. 1975.