Consistency in Dataflow Graphs

Edward Ashford Lee, Member, IEEE

Abstract—This paper explores analytical properties of programming languages with dataflow graph semantics. Such languages are currently used in certain application domains, such as digital signal processing, and a better analytical understanding of their properties may broaden their applicability. One of their most serious problems is that subtle inconsistencies between parts of the dataflow graph can be inadvertently created. These inconsistencies can lead to deadlock, or in the case of nonterminating programs, to unbounded memory requirements. Consistency is defined to mean that the same number of tokens is 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 [17]. Although inspired by the similar tests of Benveniste et al. [7], the method and the languages to which it applies are different.

Index Terms—Consistency, dataflow, functional languages, graphical programming, hybrid dynamical systems, parallel processing, signal processing, token-flow model.

I. INTRODUCTION

Effective general-purpose parallel computing paradigms remain elusive. Restricting the domain of applications, however, can yield considerable payoff in automation, hardware cost, or software efficiency. Architectures, languages, and compiler techniques specialized to regular iterative algorithms [16], neural networks, and signal processing (e.g., [19] and [21]), for example, have shown significant advances. Fortunately, each of these advances lends insight toward broader general purpose computing, and adds to a suite of techniques that may eventually be collected into a general, but heterogeneous, solution. Many of the methods that have been applied to signal processing and regular iterative algorithms are based on languages with dataflow graph semantics. For example, the techniques described in [16] and [20] use compile-time scheduling of a class of dataflow graphs to synthesize parallel implementations. Efforts to broaden the applicability of these methods, however, have been hampered by the breakdown of their analytical methods when the class of dataflow graphs is generalized.

One cause of this breakdown is that subtle inconsistencies between parts of the graph can be inadvertently created. These inconsistencies can lead to accumulation of tokens in memory. For nonterminating 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 [10]), and have resulted for example in the exclusion of certain operators in the ID language [4], [26]. This paper describes a simple systematic method that can identify these inconsistencies, and does not have the excessive restrictions of "clean" dataflow graphs [10] or "well-behaved" dataflow graphs [3]. The method would form a critical part of a compiler for languages with dataflow graph semantics. It also imposes precise theoretical constraints on the design of related languages, such as functional languages.

The analytical techniques in this paper apply to a class of languages where operators and functions (collectively called "actors") consume tokens from input streams and produce tokens on output streams. Once a token is produced, it remains in the system until it is consumed by its destination actor. Languages in this class include stream-oriented functional languages (such as Silage [14]), dataflow machine languages, and graphical signal-flow languages (such as Gabriel [20]). Languages with nondeterminate actors [15] can also be modeled. A tagged-token dataflow model [1] is not assumed, although these techniques may be applicable in that domain as well.

For illustrative purposes, we will use dataflow actors drawn from the traditional literature. Specifically, these include standard actors, such as arithmetic operators and comparators. These have the property that a single token on each input enables them to fire, and they produce a single token on each output when they fire. Actors in this class have been called homogeneous synchronous dataflow (SDF) actors [17]. 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. "Homogeneous" means that the number of tokens produced and consumed on all inputs and outputs is the same, unity. In addition, we include general SDF actors, which may produce or consume more than one token on each input or output. Two examples are shown at the top of Fig. 1. These actors are not traditional in the dataflow literature, but have important advantages that will be elaborated shortly.

In addition to SDF actors, our examples include SWITCH and SELECT, which route tokens on continuous Boolean input, as explained in Fig. 1. These are minor variations of the original Dennis actors [12], also used in [29], [28], and [25], and are the same as the DISTRIBUTOR and SELECTOR in [11]. These are not SDF actors because the compiler cannot know a priori how many tokens are produced or consumed on each input and output when the actor fires, since it depends on a Boolean input. Finally, we will illustrate the handling of nondeterminism with two types of nondeterminate merge actors.

The intent of this paper is not to propose a particular language design, nor to advocate a particular choice of actors on which to base dataflow graphs. It is instead to develop an analytical technique that applies to a class of languages in which operators are described in terms of consumption and production of tokens. The key property of these languages is that operators and functions operate on streams of tokens. There is more than traditional flexibility in managing the progression through successive tokens in a stream. In order to justify studying this class of languages, some examples of interesting programs are given in Section III below. Although we will use a graphical syntax to illustrate example programs, the method is not restricted to graphical languages. For some applications, particularly signal processing, graphical syntax has proven attractive (see [20] for one example and references to dozens of others). For others it has not. To date, however, very few experiments have been done with graphical...
languages of the type discussed in this paper. A textual language
with the same or related semantics could easily be created, but
it would demand more of the reader without adding materially
to the exposition in this paper.

II. SEMANTIC INCONSISTENCIES

Some examples of the type of inconsistencies addressed by this
paper are shown in Fig. 2. The graph in Fig. 2(a) is structurally
similar to a conditional. Without the NOT actor, this graph would
implement the functional expression:

\[ y = \text{if } (c) \text{ then } f(x) \text{ else } g(x) \]  

(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 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.

A textual description in a functional language could use the
following syntax:

\[
\begin{align*}
y_{\text{F}} &= f \text{ (if } (c_{\text{I}}) \text{ then } x) \\
y_{\text{F}} &= g \text{ (if } \neg (c_{\text{I}}) \text{ then } x) \\
y &= \text{if } (\neg (c_{\text{I}})) \text{ then } y_{\text{F}} \text{ else } y_{\text{F}} 
\end{align*}
\]

(2)

where all symbols are assumed to represent streams. The
elements of the streams are not explicitly indexed, but rather
progression through a stream is regulated by consumption and
production of tokens. In this syntax, the if-then construct is
a function that may or may not return a value (produce a
token), depending on its Boolean argument. Clearly, unrestricted
use of this or similar constructs can lead to difficulties. But
the very similarity of this syntax with traditional languages
makes misinterpretation easy, so to avoid this confusion, we rely
henceforth on a graphical representation.

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 [1] or in some cases static
buffering [18]. 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 [13]
prohibits more than one token on an arc at one time. The feedback
path in Fig. 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 (indicating 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 Fig. 2(b), 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. Textually,

\[
\begin{align*}
w &= \text{if } (c) \text{ then } x \\
y &= w + x
\end{align*}
\]

(3)

This may appear acceptable until we make the following obser-
vation. Dataflow semantics require that an arriving 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 \( x \)

\(^{1}\) Note that an efficient implementation may not need to actually perform the copy, but logically, the system must behave as if a copy occurred.
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 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 Fig. 2(c), an UPSAMPLE actor produces two tokens for each one consumed, as shown in Fig. 1. This is indicated in Fig. 2(c) by the numbers adjacent to the input and output arcs. Such an actor can be used to implement manifest iteration [19], and is consistent with the SDF model of computation [17], [18]. In Fig. 2(c), if the inputs are semi-infinite streams, then the memory requirements on the arc connecting the UPSAMPLE to the adder are unbounded.

In Fig. 2(d) 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 nonnegative 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, the second time a token x arrives, the behavior deviates from that of a guarded count. 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 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.

In all four cases in Fig. 2, the dataflow graphs do not express the intended behavior. Like type checking in strongly-typed languages, checking for consistency may reveal semantic errors.

Languages used to program dataflow machines, such as Val [24] and ID [25], [4] do not have the same sort of semantics of the programs given above, although an early version of ID had some of the features. These languages impose syntactic constraints that prevent the type of inconsistencies found in Fig. 2. Instead of actors that return multiple tokens, like UPSAMPLE, or actors that conditionally consume tokens, like SELECT, higher level constructs similar to familiar if-then-else and for-loops are provided in the languages. However, this strategy for avoiding inconsistencies may be overly restrictive. With a systematic method for identifying such inconsistencies, a new class of languages can be seriously explored.

Constructs in Val and Id such as if-then-else and do-while can be viewed as ‘graph constructors,’ since for execution on dataflow machines they are translated into dataflow graphs by a compiler. Consistency is maintained by construction. 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

\[ y = \text{if-then-else} \ (c, f(x), g(x)) \ (x). \quad (4) \]

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 is said to be curried [5]. Curried functions can be part of a dataflow graph language. The construct (4) 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 (4) 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 below.)

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 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

Fig. 2. Some problematic configurations of dataflow actors.

Fig. 3. Finite token capacity on arcs can be modeled with feedback paths and delays.
Fig. 4. A nested manifest iteration expressed using the SDF model.

have been explored before; a key reference that includes many pointers to the earlier literature is [11].

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 [29]. 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. Nonetheless, the method described in this paper applies equally well to programs written using such a language.

The method in this paper is inspired by the algebraic techniques of Benveniste et al. [7], [8], [9], [22], which can accomplish some of the same objectives for a different class of languages. Fortunately, for dataflow semantics our method is usually simpler than an adaptation to dataflow of that in [7]. This is discussed in more detail below in Section VI.

III. PROGRAM EXAMPLES

From the examples in Fig. 2, the reader might conclude that SWITCH, SELECT, and UPSAMPLE actors are the culprits, and should therefore be excluded from any practical language. Equivalently, in a textual syntax, the culprit would be any function or operator that consumes or produces (returns) any nonunitary number of tokens on a stream. We will see that nondeterministic actors cause similar difficulties. However, these actors or operators can lead to some elegant programs with important practical advantages.

Consider the program in Fig. 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 times for each firing of the A actor.

Fig. 5. A guarded count. This function takes a nonnegative integer input and counts down from that integer to zero.

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 [12], 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 to the scheduler without need for subscript analysis. Furthermore, when the iteration is manifest, the numbers of tokens produced and consumed are fixed, as implied in Fig. 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 an important practical advantage; they can be scheduled at compile time without any loss of concurrency. Hence, not only do we have a representation of a parallel for, but we have one that requires no run-time overhead to implement.

The key property of UPSAMPLE and DOWNSAMPLE actors is the ability to consume (or produce) multiple tokens on the same logical input (or output). They lead to a truly data-driven representation of iteration. The motivation for introducing SWITCH and SELECT, or similar actors that conditionally consume (or produce) tokens, can be given by example. Consider the example shown in Fig. 5. It is a corrected guarded count function [unlike Fig. 2(d)] 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 I. The actor labeled "-1" simply decrements the value of its input token by one. The actor labeled "≠ 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 "counting" 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.

TABLE I

<table>
<thead>
<tr>
<th>n</th>
<th>count</th>
<th>counting</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1 0 3 2 1 0 2 1 0 0 1 0</td>
<td>T F T T T T F T F T</td>
</tr>
</tbody>
</table>

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 I. 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 in a textual language, 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.
Note that the UPSAMPLE actor could be implemented using the guarded count, thus simplifying the language. Nonetheless, it is useful to maintain a distinction between these functions. Specifically, the UPSAMPLE produces a fixed number of output tokens, so the compiler can schedule downstream actors statically. To do the same with the guarded count would mandate that the compiler analyze the semantics of the subgraph supplying the input \( n \) to the guarded count. In general, this may be difficult.

The guarded count is used in Fig. 6(a) to build a function, called “last of \( N \),” that takes an integer input \( N \) and consumes \( N \) input tokens from the \( x \) stream, and outputs only the last of the \( N \) tokens consumed, discarding the rest. This corresponds therefore to a function where the number of arguments taken depends on the value of the first argument. Again, this may be difficult to express cleanly in standard functional languages.

The “last of \( N \)” macro dataflow actor is used in Fig. 6(b) to construct an iterative program to compute Fibonacci numbers (a recursive version will be discussed later). A Fibonacci number is the sum of the two previous Fibonacci numbers. The two delays (initialized to one) remember the two previous Fibonacci numbers. The “last of \( N \)” actor selects the desired Fibonacci number from the stream of numbers supplied at its \( x \) input. The style of iteration here is similar to that in Fig. 4, but the iteration is not manifest.

Unfortunately, the function in Fig. 6(b) 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 Fibonacci number has been produced. One possibility is to introduce the notion of context as in the tagged-token scheme [1]. Another possibility is shown in Fig. 6(c). In that graph, the boxes labeled “1” put out a token with unity value whenever they fire. The MUX (for multiplexer) actors are related to SELECT, but consume a token at both inputs, discarding one, and copying the other 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 reinitialized to false after each computation of a Fibonacci number because the counting? Boolean has value false when the guarded count has finished counting.\(^2\)

\(^{2}\)There is a significant disadvantage to the program shown in Fig. 6(c). Without a detailed analysis of the program semantics, it appears that there

It would hard to claim that this Fibonacci implementation is conventional. The control is truly data-driven, via the production and consumption of multiple tokens and not by manipulation of a loop counter. A more dramatic example, one that is more difficult to build using established functional languages, is the ordered merge shown in Fig. 7. First, a macro actor called sort is explained in Fig. 7(a). 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, and is a homogeneous SDF macro actor itself. Hence, it can be statically scheduled. The same MUX actor as in Fig. 6(c) is used.

The ordered merge program, shown in Fig. 7(b), uses the sort macro actor to merge two monotonically increasing numerical streams into one monotonically increasing stream. The xor actor is an Exclusive on. 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. 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

\[
\text{max}(+\infty, +\infty) = -\infty \quad \text{and} \quad \text{min}(+\infty, +\infty) = +\infty. \tag{5}
\]

then the merge function can take streams of finite streams with \( \infty \) marking the beginning and end of each stream.\(^3\)

\(^3\)are data dependencies between successive Fibonacci computations, while actually this is not true. One solution to this problem is a “resetting delay,” introduced in [19]. 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.

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 [19].
Fig. 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.

Fig. 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.

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. Fundamentally, what is unique here is the use of actors that can conditionally or unconditionally produce or consume one or more tokens on each logical input or output. Control is truly data-driven. Again, a language with these properties could have a textual syntax. The major difficulty is the possibility of introducing sometimes subtle inconsistencies such as those in Fig. 2.

IV. VERIFYING CONSISTENCY

In [17] it is shown that consistency can be easily checked for SDF graphs. The test is generalized here to arbitrary dataflow graphs, thus identifying all the other inconsistencies in Fig. 2.

A. The Token-Flow Model

Consider the actors in Fig. 8. These are SDF actors plus control actors fashioned after those of Dennis [12]. 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 SELECT actor in Fig. 8(b), by contrast, is not an SDF actor. The numbers of tokens produced on the two outputs are not constant (they depend on the Boolean input). In Fig. 8(b), the outputs are labeled with the long-term average number of tokens produced as a function of the proportion $p_x$ 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 FALSE GATE and FALSE GATE are merely shorthand for the SWITCH when one of the two outputs is discarded. If a textual syntax is being used, these actors may be more convenient than the SWITCH because they return only one stream.

The proportions used in Fig. 8 can be interpreted as probabilities in the Bayesian 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 \( p_{true} \) of being true. As we will see shortly, we need not be concerned with the value of \( p_{true} \), since all manipulations can be done symbolically. If two Boolean streams are produced by testing the same data, they are likely to be correlated. It may not be practical for a compiler to detect this correlation, so it is forced to make worst case assumptions. Sometimes, when Boolean streams are combined to produce new Boolean streams, a compiler may be able to detect correlations. This possibility will be addressed in Section V below. For now, we assume that the compiler knows nothing of correlation between Boolean streams.

### B. 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 stream \( c \) in the graph.

First, a requirement for consistency or strong consistency is that the graph have the potential to be nonterminating. In other words, it must be free of deadlocks. Deadlocks would occur, for example, when the graph has insufficient delays on a cycle. In [17], a systematic method for checking for deadlocks is given for SDF graphs. A related method has been applied by Benveniste et al. to a language that includes conditional invocation and non-determinism [8]. This method could perhaps be extended to dataflow graphs, but since detecting deadlocks can be decoupled from detecting inconsistencies, 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 Fig. 2(b) is consistent if \( c \) is known to be always true. To show this, we can perform a simple analysis using Fig. 8. This analysis alludes to a more systematic method that will be developed below.

Since both actors in Fig. 2(b) 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 \( p \) tokens on its \( T \) output, implying that the add actor should fire \( p \) times as often as the SWITCH. These two statements can be reconciled if and only if \( p = 1 \). So we see that the graph in Fig. 2(b) is consistent, subject to \( p = 1 \), but not strongly consistent.

The remaining examples in Fig. 2 can be dispatched similarly. The example in Fig. 2(a) has more actors than the one in Fig. 2(b), so we should be more systematic about its analysis. Define \( q_{not} \) to be the proportion of total firings that are firings of the SWITCH actor. Similarly, \( q_{not} \) is the proportion of SELECT firings, \( q_f \) of \( f() \) firings, and \( q_g \) of \( g() \) firings. Since these are all the actors under consideration,

\[
q_{not} + q_f + q_g = 1
\]  

The path of \( c_1 \) and \( c_2 \) tokens requires that

\[
q_{sw} = q_{sc} = q_{sc}.
\]  

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

\[
p_c q_{sw} = q_f
\]

\[
(1 - p_c) q_{sw} = q_g
\]

\[
p_c q_{sc} = q_f
\]

\[
(1 - p_c) q_{sc} = q_g.
\]  

Combining (7) and (8) we conclude that consistency requires that

\[
p_{c1} = p_{c2} = 0.5.
\]  

Hence, this graph is consistent subject to (9), and therefore not strongly consistent. Applying the same method to Fig. 2(c), consistency requires that

\[
2q_c = q_f
\]

\[
q_i = q_g.
\]  

Since we also require that

\[
q_i + q_g = 1
\]  

this graph is not consistent.

This method can be made even more systematic, and indeed can be automated. Consider an actor \( A \) connected to actor \( B \). Let \( \gamma_A \) 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 \( \gamma_B \) 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

\[
\gamma_A q_A = \gamma_B q_B.
\]  

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

\[
\Gamma q = 0
\]  

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

\[
1^T q = 1.
\]  

where \( 1 \) is a column vector full of 1's. The matrix \( \Gamma \) 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. Consistency requires that there be a solution to (13) and (14) for some set of Boolean proportions. Strong consistency requires that there be a solution to (13) and (14) for any set of Boolean proportions.

We first illustrate the systematic method by proving that the attempted guarded count of Fig. 2(d) is not consistent. The graph has been reduced to its essentials in Fig. 9(a), where the three actors have been numbered, as have the arcs connecting them (using triangular flags). 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 Fig. 2(d) has been omitted in Fig. 9(a). 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
propotion $p_c$ of true's at the output of the delay is the same as
the proportion at the input. Hence, if the initial token has value
"false," this simply imposes the constraint that $p_c < 1$, while if
it has value "true," the corresponding constraint is $p_c > 0$. We
will address this and related constraints in Section V below, so
for now the delay will be omitted in the analysis.

The topology matrix for Fig. 9(a) is

$$
\Gamma = \begin{bmatrix}
1 & -1 & 0 \\
-3 & 1 & 0 \\
1 & 0 & -1 \\
\end{bmatrix}
$$

(15)

From this we can see that (13) and (14) have a solution if and
only if $p_c = 1$.

To make the conclusion even more obvious, notice that (13)
implies that the rank of $\Gamma$ 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 (15) 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 [17]), so
we conclude that consistency requires that

$$
\text{rank}(\Gamma) = s - 1.
$$

(16)

Strong consistency requires that this equation hold for all values
$p_c \in [0, 1]$ for each Boolean probability $p_c$ in the system.

The systematic method can also be illustrated by proving that
the guarded count in Fig. 5 is strongly consistent. The outline
of the guarded count is shown in Fig. 9(b) with each actor and
each pertinent arc numbered. There are six arcs and four actors.

With this numbering,

$$
\Gamma = \begin{bmatrix}
-3 & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & -1 \\
0 & -1 & p_c \\
\end{bmatrix}
$$

(17)

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

$$
Q = \frac{1}{3 + p_c}
\begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & -1 \\
0 & -1 & p_c \\
\end{bmatrix}
$$

(18)

satisfies (13) and (14). 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 Fig. 10. This approach is consistent with using
hierarchy to control complexity. The labels in Fig. 10(a) indicate
that for every $\frac{1}{3 + p_c}$ tokens consumed, there will be $1/3 + p_c$
"count" and "counting?" tokens produced. These expressions
are obtained by multiplying the entry $q_c$ in (18) by the average
number of input (output) tokens consumed (produced) by each actor
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 $\Gamma$ can be scaled without affecting its rank. Multiplying
the expressions in Fig. 10(a) by $\frac{1}{3 + p_c}$, we get the more intuitive
expressions in Fig. 10(b).

The same technique can be applied to the examples in Fig. 6
and Fig. 7, getting the consolidated results shown in Fig. 11.

The interpretation of the solution $q_c$ to (13) and (14) as a vector
of proportions of firings of each actor is useful for developing
intuition. Consider Fig. 12(a). 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
Fig. 12(a) is consistent if and only if $p_c = 0.5$. By contrast, the
program in Fig. 12(b) is strongly consistent.

Consistency can also be checked when recursion is used.
Recursion in a dataflow graph can be represented using self-referential hierarchy. In Fig. 13, for example, the path of the $c$
Boolean indicates that $q_{AB} = q_E$, so $x = y$, where $x$ and $y$
are the number of tokens consumed and produced by the Fibonacci
function. Since any $x = y$ will yield the same consistency result,
we can set $x = 1$. Showing that Fig. 13 is consistent then
becomes trivial.

Finally, nondeterminate actors can also be accommodated in
the analysis. It has been argued that nondeterminate 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 [15].
A nondeterminate merge has been proposed by Arvind and Brock [2],
and resource managers have been built using it. The
nondeterminate merge is an "unordered" merge where token
streams can be merged in unpredictable ways, depending on the
time of arrival of tokens. By contrast, the ordered merge of Fig. 7
and Fig. 11 merges two streams deterministically based on their
values. The token-flow model can be used to verify consistency of
programs using a nondeterminate merge, precisely because the
token-flow model does not require knowledge of the expected number of tokens consumed. Fundamentally, nondeterminism means that there is not enough information in the program for this to be known. The token-flow model for the unordered merge is the same as that for the ordered merge shown in Fig. 11.

The unordered merge of the previous paragraph will eventually pass all tokens that appear at its inputs to the output. Hence, the sum of the expected number of tokens consumed on the two inputs is unity. Another type of nondeterminate merge might discard one input if both input tokens appear "simultaneously," where the meaning of "simultaneously" depends on the implementation and/or the context. Such an actor would be a dataflow version of the "default" operator in the SIGNAL language [8]. The token-flow model for such an actor is only slightly more complicated, as shown in Fig. 14. When the actor fires, it has probability \( p_i \) of consuming a token from the upper input and probability \( p_r \) of consuming a token from the lower path. Normally, \( p_i + p_r \geq 1 \).

V. MUTUALLY DEPENDENT BOOLEANS

The analysis so far ignores any possible relationship between values in separate Boolean streams. Each Boolean stream \( e \) gets a symbol \( p_e \) associated with it, and the analysis reveals constraints on \( p_e \) that may have to be satisfied for consistency. When a program is strongly consistent, there are no constraints, and no further analysis is required. Similarly, when a program is inconsistent, additional analysis will not help. However, when a program is conditionally consistent, further examination may be required. In particular, Boolean streams may be interrelated, and these interrelationships may either contradict or enforce the conditions for consistency.

Some examples of Boolean operators are shown in Fig. 16. These actors take Boolean inputs and produce Boolean outputs. In these cases, it is sometimes possible for a compiler (or a programmer) to make use of the joint statistics. The probabilities of the outputs being "true" can be expressed in terms of joint probabilities of the inputs being true, as shown in Fig. 16. 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. However, care must be taken that this joint probability applies to tokens in the \( c_1 \) and \( c_2 \) streams that are consumed by the same firing of the actor. When the compiler (or programmer) can analyze these joint statistics, these expressions can be used to determine consistency when Boolean streams are not independent.

None of the examples discussed so far require using the Boolean relationships in Fig. 16 to determine consistency. Even the ordered merge of Fig. 7, which contains an Exclusive or, is consistent for any Boolean stream coming out of the xor operator, so the logical properties of the xor are not important. Any logical operator in its place would also yield a strongly consistent graph. However, not all programs exhibit this property. For some programs, solving (13) and (14) may lead to a conclusion that a program is only conditionally consistent. Applying the relationships in Fig. 16 to this result may show that the program is strongly consistent, or not consistent at all. Hence, if an analysis ignoring the logical relationships between Booleans reveals conditional consistency, this result should be considered inconclusive. Further analysis can sometimes help.

Consider the example shown in Fig. 16. Only Fig. 16(c) is strongly consistent. Solving (13) and (14) for Fig. 16(a), we find that it is consistent if and only if \( p_{c_2} = 1 \). It is easy to verify by inspection that this situation leads to consistent graphs, but also to semantics that are probably not intended. The program in Fig. 16(b) is an attempt to correct the problem. However, it also fails the strong consistency test because the \( \And \) operator synchronizes the \( c_1 \) and \( c_2 \) token streams, constraining the rates of tokens to be the same. Again, we find consistency if and only if \( p_{c_2} = 1 \). There is no need to use the Boolean relationships of Fig. 16. For Fig. 16(c), however, solving (13) and (14), we get consistency if and only if

\[
p_{c_3} = p_{c_2} p_{c_4}. \tag{19}
\]

From the relationships in Fig. 16,

\[
p_{c_2} = \Pr[c_1 = T, c_2 = T] \quad \text{and} \quad p_{c_4} = \Pr[c_1 = T | c_2 = T].
\]

Hence, condition (19) is equivalent to

\[
\Pr[c_1 = T, c_2 = T] = p_{c_2} \Pr[c_1 = T | c_2 = T]. \tag{20}
\]
which is always true by the multiplication rule in probability! Consequently, this graph is strongly consistent.

In summary, the system of equations given by (13) and (14) can be augmented with a set of relationships such as those in (20), which combined with the rules of probability may reveal strong consistency or inconsistency when (13) and (14) alone only reveal conditional consistency.

The above analysis must be performed carefully, ensuring that when probabilities are combined at a logical operator, they are combined for tokens that are simultaneously consumed. If, for example, a delay were introduced in the path of one of the Boolean streams in Fig. 16(c), the analysis might require knowledge about joint statistics of successive tokens on the streams. This knowledge may not be available to a compiler, or even to the programmer. Similarly, Boolean streams that are generated by testing non-Boolean streams may have joint statistics that are difficult or impossible for a compiler to discern. In principle, a compiler that does elaborate semantic analysis of the program and has complete information about the input streams could use this type of analysis to verify consistency (barring nondeterminate actors), but such a compiler is probably not practical. A practical compiler will only be able to completely analyze a subset of all programs. For this reason, a simple version of a compiler would solve (13) and (14), and if there are conditions for consistency, then it would alert the programmer to them, using for example the following output:

**Warning:** Consistency holds only if Boolean signals $c_1$ and $c_2$ satisfy the following relationship: $p_{y_1} = 1 - p_{y_2}$.

We then rely on the programmer's understanding of the program semantics to draw the final conclusion about consistency. A more elaborate compiler could use relationships like those in Fig. 16 to draw firm conclusions when possible, and again alert the programmer when firm conclusions are not attainable. A comparatively simple symbolic processing kernel could accomplish this task, and would avoid some unnecessary warnings.

It is worth noting that Benveniste *et al.* propose the use of finite-state machines to represent the dynamics of Boolean signals in a program containing delays [7]. But ultimately the technique still relies on human interpretation of the conditions for consistency on at least some programs. For example, the technique does not help to establish interrelationships between Boolean streams that are generated from non-Booleans. Further study is warranted to identify the best method for automating the analysis of the class of programs that can be analyzed.

**VI. COMPARISON TO HDS THEORY**

The method given in this paper was inspired by the algebraic techniques of Benveniste *et al.* [7]–[9], [22]. These methods have been applied to a language called SIGNAL which is not a dataflow graph language, but is related. The SIGNAL language is used to describe systems in a class called Hybrid Dynamical Systems (HDS). Analytical methods in HDS theory identify inconsistencies similar to those found in dataflow graph languages.

HDS theory 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 nonzeros 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:

$$
x + x = -x$$

$$x^2 = x$$

(22)

Non-Boolean values are simply assigned an indeterminate $\pm 1$. 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 size is represented for non-Booleans.

Given this framework, functions similar to the dataflow actors we have been using can be described as shown in Fig. 17. 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 $x = 2$. If $c = 1$ (the control signal is true) then $x = y$, using (22). This is precisely the behavior we expect from the SELECT function. However, the description is 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 Fig. 17 requires that $y^2 = 1$ if $c = 1$ and the third requires that $x^2 = 1$ 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
Fig. 15. 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.

Fig. 16. The programs in (a) and (b) are consistent only for \( p_{xy} = 1 \). The program in (c) is strongly consistent.

(say \( y^2 = 1 \)), then the other input must be present \( z^2 = 1 \), and the output will be present \( z^2 = 1 \). Nonhomogeneous SDF actors cannot be described statically; a finite-state-machine model is introduced to model the dynamics.

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. A consequence of this is that slots in different signals have a temporal relationship with one another. 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. However, the FIFO queue introduces dynamics, considerably complicating the analysis. Using HDS to represent dataflow semantics, therefore, is not very
Fig. 17. The input/output relationships of certain functions in a hybrid dynamical system (HDS) are shown here. All relationships are expressed in a Galois field with three elements, -1, 0, and +1.

Fig. 18. An example of a program that is inconsistent under either the HDS model or the dataflow model.

The FIFO queue implicit on every arc also introduces dynamics. Although analysis is possible in this framework, it is not simple, and is beyond the scope of this paper. The reader is referred to [7]. The comparative simplicity of the token-flow analysis of dataflow programs with dynamics should be viewed as one of its major advantages. The simplicity arises from performing a steady-state analysis. The steady-state analysis is made possible by the implicit queuing of tokens on arcs, absent in the language model in [7].

As mentioned before, neither the token-flow model nor HDS theory can always verify consistency. For some programs, both models yield inconclusive results when the relationships between Boolean streams cannot be known to the compiler. This occurs, for example, when two Boolean streams are derived by testing the same non-Boolean stream.

VII. CONCLUSIONS

Consistency and strong consistency in dataflow graphs has been defined, and a systematic test for them has been developed. The test uses a simple steady-state analysis of the flow of tokens, and yields one of three possible results. A program is either inconsistent, conditionally consistent, or strongly consistent. When a program is conditionally consistent, a compiler could simply alert the programmer of the conditions for consistency, and the programmer could use his or her understanding of the semantics of the program to draw the final conclusion. A more sophisticated compiler could symbolically manipulate Boolean relationships to attempt to draw a conclusion. For some programs, however, a compiler cannot draw a definitive conclusion without extensive semantic analysis of the program and information about the program inputs.

The token-flow model given here has been compared to similar tests applied to hybrid dynamical systems by Benveniste et al. [7]–[9], [22]. It is argued that dataflow semantics make steady-state analysis possible, which leads to a simpler method in most cases.

ACKNOWLEDGMENT

Numerous discussions helped solidify the arguments given in this paper. The author would like to especially thank A. Benveniste, who went well beyond the call of duty in carefully reading and commenting on the manuscript. In addition, other significant contributions of ideas and criticism were made by Arvind, D. Culler, S. Ha, A. Ranade, R. Righter, and V. Srinivas. In addition, P. LeGuernic of INRIA was kind enough and patient enough to explain to me the basics of HDS theory and the SIGNAL language, which inspired this work.
REFERENCES


Edward Ashford Lee (S’80-M’86) received the B.S. degree from Yale University in 1979, the masters (S.M.) degree from MIT in 1981, and the Ph.D. degree from the University of California, Berkeley, in 1986. He is an Associate Professor in the Department of Electrical Engineering and Computer Science, University of California, Berkeley. He joined the faculty in 1986. His research activities include parallel computation, architecture and software techniques for programmable DSP’s, design environments for real-time software development, and digital communication. He is coauthor of Digital Communication with D. G. Messerschmitt (Norwood, MA: Kluwer Academic, 1988) and Digital Signal Processing Experiments with Alan Kamen (Englewood Cliffs, NJ: Prentice-Hall, 1989) as well as numerous technical papers. From 1979 to 1982 he was a member of the Technical Staff at Bell Telephone Laboratories, Holmdel, NJ, in the Advanced Communications Laboratory, where he did extensive work with early programmable DSP’s and exploratory work in voice-band data modem techniques and simultaneous voice and data transmission.

Dr. Lee was a recipient of the 1987 NSF Presidential Young Investigator Award, a paper award from the Signal Processing Society of the IEEE, an IBM Faculty Development Award, and the 1986 Saksirisorn Prize at U.C. Berkeley for the best thesis in electrical engineering.