A Partial Order Semantics for Cooperative Data-driven Computation

J. Dean Brock

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27599-3175 USA
brock@cs.unc.edu

Abstract

The scenario model is a simple semantic theory in which a data-driven computation is represented by a partial order symbolizing event causality. In the original model it was assumed that the concurrent processes were interconnected by unbounded channels and consequently there was no need to specify interprocess "flow control." In this paper, we extend scenarios with a new partial order which is used to constrain the times at which modules may send output. This allows the scenario model to be used with unbuffered data-driven computation and related concurrent paradigms, such as, asynchronous circuitry or protocol-specified computer communication.

1. Introduction

A data-driven computation, as accomplished by a dataflow machine [1, 4] or a more conventional network of processors, occurs within a collection (graph) of interconnected processes. These processes are joined to each other by channels, or arcs, connecting the output and input ports of processes. During the course of a computation, tokens, or data values, traverse these channels from one module to another. The computation is data-driven or dataflow in the sense that it is the communication of these data values that controls the computation.

Some of the processes or modules of the graph may have unconnected ports. These unconnected ports may be viewed as the input and output ports of the graph, and thus the graph itself may be abstractly viewed as just another data-driven process. The goal of the dataflow semanticist is the development of an abstract representation of data-driven processes which allows the derivation of the result of executing the "parent" graph from the representations of its components.
In 1974, Kahn [5] solved this problem for a particular form of dataflow graph, those composed of determinate processes interconnected by unbounded FIFO channels. In this case, determinacy refers to the independence of the output results of the processes and the relative timings of the input token arrivals. Thus a process with two input ports which produces as its output the first value to arrive on either port would not be determinate. It is possible to represent determinate processes as functions from sequences, one for each input port, of input tokens to sequences of output tokens. Kahn showed that the result of executing interconnected determinate processes could be derived using fixpoint semantic theory.

In 1981, the author and Ackerman [3] showed that it is impossible to represent non-determinate dataflow processes by the straightforward extension of Kahn's functions to relations because relations contain insufficient detail for use in faithfully deriving the result of process interconnection. We also defined a partial order model of computation called scenarios. A scenario consists of a set of events, in this case the tokens that exist during a data-driven computation, and a partial ordering corresponding to causality. Two tokens are related if one "causes," perhaps through a very long chain of events, the other to exist. When a single scenario is examined, two distinct forms of causality are observed. One is from input tokens to output tokens. The output token is causally preceded by all those input tokens that contributed to its production. The other is between all the tokens that arrive on a single port. All tokens of a single port are totally ordered by causality with the justification that a token cannot be produced until its predecessors on its channel exist.

A non-determinate data-driven process can be represented by a set of scenarios, each corresponding to one possible computation of that process. Given a dataflow graph and the sets of scenarios representing its constituent processes, it is possible to derive the scenario set representation of the graph itself. The mathematical techniques required to do this are very simple. The transitive closures of joined scenarios, one from each process, are "computed" and tested to see if antisymmetry is still satisfied. This ensures that causality is followed across process boundaries and that no value is allowed to cause itself. Next the internal events of joined scenarios are dropped so that only the events of the input and output ports of the graph itself appear in its scenario set representation. The represented computation may be either terminating or infinite. We will explain this construction in more detail when augmented scenarios are introduced later in this paper. The construction for scenario sets was described informally in the original paper [3] and in more formal
algebraic terms in a later report [2].

In this paper, we retain the operational basis of scenarios. We seek to extend the scenario theory by adapting it to unbuffered data-driven graphs. In the next section we describe in detail the operational model of “cooperative” data-driven computation for which we are developing this theory. In the third section we describe our theory and give examples of its application. Finally, we present other extensions we are investigating.

2. Cooperative Data-driven Computation

At the implementation-level, the primary distinction between our processes and the processes used in most concurrent execution schemes is that our processes are interconnected by very simple channels, really just wires, and not by FIFO queues of arbitrary length or by mysterious and complex networks of arbitrary capacity which may change the order of values sent between processes. Because our channels are so simple, connected processes must be very careful not to destroy previous transmitted values by overwriting.

Processes cooperate to avoid overwriting. The cooperation is voluntary in the sense that it is not enforced by some underlying mechanism as is the synchronous transmission of values in CSP or Ada. Rather, the sending and receiving processes on a channel must both adhere to a specification, or a contract, that states when it is safe, or legal, for the sending process to place a new data value on the channel. It is the responsibility of the receiving process to signal the sender when transmission is permitted. The only means of signaling is the transmission of other values, often special acknowledgment values, which the sender must “see” before it places its next output on the channel. The feedback of ready-to-receive signals need not be so direct as we will see in subsequent examples.

We will use only two easily understood examples of cooperative data-driven processes in this paper. There are more elaborate and useful processes which fit our operational requirements but these two are sufficient for introducing our theory. The first example is a very useful component for constructing asynchronous circuits. It is the C-element, sometimes called the rendezvous. The C-element has two input ports and one output port. These ports may receive or transmit only two input values, 0 and 1. The C-element produces the output 1 only when both of its inputs are also 1. The 1 output signals the rendezvous of the processes connected to the input ports. Only then may the input values be changed from 1 to 0. When both inputs are changed to 0, the C-element should produce the output value 0. Again, only then may the input values be changed from 0 to 1.

Implementation of the C-element should be easy. When both of its inputs agree, its
output must also agree. However, there’s more to the C-element than its implementation. An integral part of the C-element’s specification is the understanding that its users, the processes connected to its ports, will use it properly. Proper use means, among other things, that an input value will not change from 1 to 0 until the output value has changed to 1. This implies that when the C-element is used in a graph there will be some feedback from its output port to its input processes if the C-element is being used appropriately. Hardware designers have long used timing diagrams to express such requirements. An example timing diagram for the C-element is show below. Our semantic representation of cooperative data-driven processes will be very similar to the partial order implied by this diagram.

![Diagram of the C-element](image)

The C-element

Our other example is a useful element of dataflow machines [4], the one-place buffer. The one-place buffer has four ports: (1), data-put, (2), put-ready, (3), data-get, and (4), get-ready. The put-ready port is an output port at which a special acknowledgment token is produced whenever the buffer is ready to receive an input value. Similarly, the get-ready port is an input port from which the acknowledgment token must be received before the buffer produces an output value.

![Diagram of the one-place buffer](image)

The one-place buffer

3. Partial order representation of cooperative computation

A scenario is one possible history of the computation of a process. The scenario contains the tokens produced by that computation plus a causality relation from input
events to output events. Several input tokens may cause one output token. For example, a C-element will require two input transitions before making an output transition. One possible computation of the C-element can be represented by the following scenario:

\[ \begin{array}{c}
\text{in}_1 \\
1 \\
\text{in}_2 \\
1 \\
\text{out} \\
1 \\
\end{array} \]

A C-element scenario

Note that event causality is not externally observable. It is a property of a process that can only be derived by examining the actual implementation of the process. Being an implementation property, it is verifiable by the programmer of the process.

For "normal" data-driven models only the causality relation is necessary. However, for cooperative data-driven computation a second relation is required. We call this relation the protocol order. The protocol order is from output events to input events. It encodes restrictions placed on the process user. If an output event \( b \) is related to an input event \( a \), the user of the module must never send input \( a \) to the module until the output \( b \) has been seen (or at least heard of). We call scenarios augmented with protocol orders dialogues. The two orders, causality and protocol, form an covenant between a process and its neighbors in the program graph. Each process guarantees to fulfill its obligations, stated in terms of a set of dialogues, as long as the others do likewise.

The semantics, or meaning, of a cooperative process is the collection of all its possible communication histories, that is, its dialogues. One dialogue for the C-element is shown in the following figure.

\[ \begin{array}{c}
\text{in}_1 \\
1 \\
\text{in}_2 \\
1 \\
\text{out} \\
1 \\
\end{array} \]

A C-element dialogue

The C-element is represented by an infinite set of similar dialogues corresponding to longer and longer conversations with its users. One of these dialogues has an infinite number of elements and symbolizes the use of the C-element in a non-terminating computation.

98
Appropriate dialogues for common processes may be inferred from our informal notions of how these processes should function. For example, we know that data should not be placed in the one-place buffer until a signal is received from the put-ready port and similarly that the one-place buffer should not produce this data until a signal is received on the get-ready port. Representing the capacity of the buffer is a bit more obscure, but this may be done by ensuring that the get-ready signals are never more than one ahead of the put-ready signals. These constraints are represented in the following dialogue.

```
data-put  put-ready  data-get  get-ready
```

In order to derive the result of executing a collection of processes, it is necessary to compute from all the dialogues sets of the processes a dialogue set for the system itself. This is done by looking at combinations of scenarios from each component and discarding those which could not possibly co-exist and can easily be extended to work with dialogues. We will illustrate this process for dialogues by showing how two one-place buffers can be concatenated to form one two-place buffer.

Suppose two one-place buffers are connected as show below:

```
A two-place buffer
```

To derive dialogues for the two-place buffer, we first look at all pairs \((X, Y)\) of dialogues, where \(X\) and \(Y\) are dialogues for the one-place buffer. In order for this pair to be consistent, two facts must hold. First the events of the two dialogues must match, that is, the sequence of tokens passed on the connected ports must agree because all the outputs at one end of a channel become the inputs at the other end.

\(X\) and \(Y\) have partial orders corresponding to causality and protocol relations. If \(X\) and \(Y\) are consistent in the sense of having the input and output tokens on connected ports
match, then we can merge each matched pair of events into a single event. The combined causality and protocol relations on $X$ and $Y$ extend naturally to relations on the combined set of events. The following figure illustrates this extension in the two-place buffer.

![Diagram of causality and protocol relations]

Extension of causality and protocol relations

In this example, we only need drop the internal ports of the two-place buffer to obtain one of its dialogues. The causality relation of the composition can always be obtained merely by dropping the internal events. Unfortunately, in general, constructing the new protocol order is significantly more difficult.

Before we look at this construction we should note that the extended order is not always a partial order. For example, if two one-place buffers are joined in a ring we can obtain "dialogues" in which a token seems to cause itself, that is, the causality relation is not a partial order. We wish to eliminate such physical absurdities from our theory. This is done is by discarding dialogue pairs $(X, Y)$ whose combined causality relation is not a partial order. There are also situations in which the protocol order has loops. These situations represent unsafe computations which should also be avoided.

Rather than describe the construction of the protocol order for the general case of process graphs, let's look at how a new protocol order can be constructed when we take a single process and connect one of its input ports to one of its output ports. It is adequate to analyze this single case because we can build up arbitrary graphs with just three operators for (1), relabeling ports; (2), "joining" two graphs into a larger graph without connecting any of their ports; and (3), connecting an input port to an output port [2]. Suppose we have a process (or graph) $P$ and we wish to connect input port $\alpha$ to output port $\beta$ to form a new process $P'$. We construct the dialogues of $P'$ from those of $P$. Suppose $D$ is a dialogue for $P$. For $D$ to be a suitable dialogue for the $P'$ connection, the sequence of values on the $\alpha$ and $\beta$ ports must agree and there must be no cycles with the two sequences
are united. Having no cycles means that there is no case in which the i'th element of α precedes the i'th element of β in the causal order of D or, similarly, the i'th element of β precedes the i−1'st element of α in the protocol order. Suppose D is suitable according to these criteria, now how do we construct dialogues D' for P'? As previously mentioned the new causality order is trivially obtained. Just extend the original causality order across the united tokens.

The protocol order is more difficult to construct. Suppose there is an input port δ whose j'th element is preceded by the i'th element of β in the protocol order. We'll denote this as < (β, i), (δ, j) > ∈ pr(D). Externally, output (β, i) can no longer be seen so we cannot directly determine when it is safe to input (δ, j). However, we do know that output (β, i) results in input (α, i). Suppose (α, i) causally precedes some output (γ, k) in the causality order of D, denoted < (α, i), (γ, k) > ∈ ca(D). In that case, the output of (γ, k) is adequate evidence that it is safe to input (δ, j) in the computation of P'. Note, that as (α, i) may cause several different output tokens there are many (δ, j) that are candidate protocol predecessors of (δ, j). Some of these candidates may turn out to be unsuitable because (δ, j) will be a causal predecessor of (γ, k). We do not extend pr(D) with all the possible < (γ, k), (δ, j) > pairs. Rather, as only one such pair is needed we generate several possible dialogues for P', one for each pair. In fact, there may be many other other < (β, l), (κ, m) > in the protocol relation of D which could multiply the number of ways dialogues for P' may be formed. However, many of these dialogues will be redundant because they will have more restrictive protocol orders. (A protocol order R is more restrictive than S, if whenever < (μ, p), (ν, q) > ∈ S there is n ≥ q such that < (μ, p), (ν, n) > ∈ R. Mathematically it is convenient to add to the set of dialogues representing a process Q all the more restrictive dialogues. Then it becomes possible to exploit some fixed point techniques. If this is done the dialogues often become significantly
simpler as more and more connections are made.)

\[ \dot{\epsilon}_3 \]

\[ \langle \alpha, i \rangle \rightarrow \langle \beta, i \rangle \rightarrow \langle \gamma, k \rangle \]

Discovering a protocol relation

In the preceding paragraph we assumed that a suitable output \((\gamma, k)\) for our input \((\delta, j)\) would be found; however, there is no guarantee that such a \((\gamma, k)\) will exist and this will happen when processes are connected which do not always obey each others' causality-protocol restrictions. Of course, in such a case the candidate dialogue \(D\) of \(P\) must be discarded as being unsafe. It's even possible that there will be no safe candidates and that \(P'\) will be represented by the empty set of dialogues.

Thus far we have dealt with the problem of extending the protocol order when an output value that was used to signal safe input is now hidden on the connection. A similar problem occurs when a hidden input value needs to "see" an output before transmission. The mechanism for making this extension is similar to that explained above. We start out with \(< (\gamma, k), (\alpha, i) > \in \text{pr}(D)\) and look for \(< (\delta, j), (\beta, i) > \in \text{ca}(D)\). \((\gamma, k)\) may be considered a potential protocol predecessor of \((\delta, j)\).

4. Conclusion

We have developed a method of taking dialogues, formal descriptions of unbuffered data-driven processes, which allows us to derive the result of interconnecting these processes. The dialogues of these processes state the ways in which they can be safely used. The contribution of the work is that it allows one to determine how a interconnection of such processes can itself be safely used. We can then determine whether or not such an interconnection satisfies a priori requirements and specifications for implementing a given task.

Acknowledgments

This work was supported in part by the National Science Foundation under grant number DCR-8406850.
References


