Bounded Scheduling of Process Networks

by

Thomas M. Parks

B.S.E (Princeton University) 1987

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering — Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Edward A. Lee, Chair
Professor David G. Messerschmitt
Professor David L. Wessel

1995

Thomas M. Parks. *Bounded Scheduling of Process Networks.* Technical Report UCB/ERL-95-105. PhD Dissertation. EECS Department, University of California. Berkeley, California 94720. December 1995. http://ptolemy.eecs.berkeley.edu/papers/parksThesis.

Bounded Scheduling of Process Networks

Abstract

Bounded Scheduling of Process Networks

by

Thomas M. Parks

Doctor of Philosophy in Engineering — Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Edward A. Lee, Chair

We present a scheduling policy for complete, bounded execution of Kahn process network programs. A program is a set of processes that communicate through a network of first-in first-out queues. In a complete execution, the program terminates if and only if all processes block attempting to consume data from empty communication channels. We are primarily interested in programs that operate on infinite streams of data and never terminate. In a bounded execution, the number of data elements buffered in each of the communication channels remains bounded.

The Kahn process network model of computation is powerful enough that the questions of termination and bounded buffering are undecidable. No finite-time algorithm can decide these questions for all Kahn process network programs. Fortunately, because we are interested in programs that never terminate, our scheduler has infinite time and can guarantee that programs execute forever with bounded buffering whenever possible. Our scheduling policy has been implemented using Ptolemy, an object-oriented simulation and prototyping environment.

For my wife Beth
and our children
Rachel, and Benjamin.

Contents

List of Figures

Acknowledgments

This work is part of the Ptolemy project which is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), Semiconductor Research Corporation (project 95-DC-324), National Science Foundation (MIP-9201605), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, Mentor Graphics, Mitsubishi, Motorola, NEC, Pacific Bell, Philips, and Rockwell.

Chapter 1

Process Networks

In the process network [24, 25] model of computation, concurrent processes communicate through unidirectional first-in, first-out (FIFO) channels. This is a natural model for describing signal processing systems where infinite streams of data samples are incrementally transformed by a collection of processes executing in sequence or in parallel. Embedded signal processing systems are typically designed to operate indefinitely with limited resources. Thus our goal is to execute process network programs forever with bounded buffering on the communication channels whenever possible.

A process network can be thought of as a set of Turing machines connected by one-way tapes, where each machine has its own working tape [24]. Many of the undecidable issues of the process network model of computation are related to the halting problem for Turing machines. It is well known that it is impossible to decide (in finite time) whether or not an arbitrary Turing machine program will halt. We wish to execute process network programs forever, but because the halting problem is undecidable we cannot determine (in finite time) whether this is even possible. We also wish to execute process network programs with bounded buffering on the communication channels. Because this question can be transformed into the halting question, as we will show later, it is also undecidable.

Thus there are two properties we might use to classify a process network program: termination and boundedness. But first we must determine if these are actually properties of programs. Are termination and boundedness determined by the definition of the process network, or could they depend on the execution order? We want to execute programs indefinitely, but could we make a bad scheduling decision that turns a nonterminating program into a terminating one? For an important class of programs, which we call Kahn process networks, termination *is* a property of the program and does not depend on the execution order. However, the number of data elements that must be buffered on the communication channels during execution does depend on the execution order and is not completely determined by the program's definition. We show later how to transform an arbitrary Kahn process network so that it is *strictly bounded*: the number of data elements buffered on the communication channels remains bounded for all possible execution orders.

Because the questions of termination and boundedness are undecidable it is not possible to classify

every program in finite time. An algorithm can be devised that will find a "yes" or "no" answer for many systems, but there will always be some systems for which no answer can be obtained in finite time. It has been shown for restricted forms of process networks that both of these questions are decidable and can be answered for arbitrary systems that conform to the restricted model of computation. Thus, for these restricted models it is possible to classify any program before beginning execution. We will discuss some of these restricted computation models, but in this thesis we address the more difficult problem of scheduling general Kahn process networks.

In scientific computing, programs are designed to terminate. In fact, great effort is expended to optimize programs and ensure that they terminate as soon as possible. It would be unacceptable to have a program that produced its results only after infinite time. It would be even worse to have a scheduling algorithm that could require infinite time before even beginning execution of the program. However, many signal processing applications are designed to never terminate. They have an infinite data set as input and must produce an infinite data set as output. In this context it is not strictly necessary to classify a program before beginning execution. We can let the scheduler work as the program executes. Because the program is designed to run forever without terminating, the scheduler has an infinite amount of time to arrive at an answer. We will develop a general policy that describes a class of schedulers that satisfy our goals of non-terminating, bounded execution for arbitrary Kahn process networks when this is possible.

1.1 Kahn Process Networks

Kahn [24, 25] describes a model of computation where processes are connected by communication channels to form a network. Processes produce data elements or *tokens* and send them along a communication channel where they are consumed by the waiting destination process. Communication channels are the *only* method processes may use to exchange information. Kahn requires that execution of a process be suspended when it attempts to get data from an empty input channel. A process may not, for example, examine an input to test for the presence or absence of data. At any given point, a process is either *enabled* or it is *blocked* waiting for data on *only one* of its input channels: it cannot wait for data from one channel *or* another. Systems that obey Kahn's model are *determinate*: the history of tokens produced on the communication channels do not depend on the execution order [24].

Figures 1.1, 1.2 and 1.3 show the definitions of three processes. We use pseudo code with a syntax similar to C. All of these processes operate on integers or streams of integers. In our examples, the `put` and `get` operations are used to produce and consume single tokens, although they could be generalized to consume and produce multiple tokens simultaneously.

Individual processes can be defined in a *host language*, such as C or C++ with the semantics of the network serving as a *coordination language*. Care must be taken when writing code in the host language. It may be tempting to use shared variables to circumvent the communication channels, but this may violate Kahn's model and result in a nondeterminate system. With a little discipline it is not difficult to write deter-

```
int stream W = process f(int stream U, int stream V)
{
    do
    {
        put(get(U),W);
        put(get(V),W);
    } forever;
}
```

Figure 1.1: A process that interleaves two streams into one.

```
(int stream V, int stream W) = process g(int stream U)
{
    do
    {
        put(get(U),V);
        put(get(U),W);
    } forever;
}
```

Figure 1.2: A process that distributes even and odd elements of one stream to two streams.

```
int stream V = process h(int stream U, int x)
{
    put(x,V);
    do
    {
        put(get(U),V);
    } forever;
}
```

Figure 1.3: A process that inserts an element at the head of a stream.

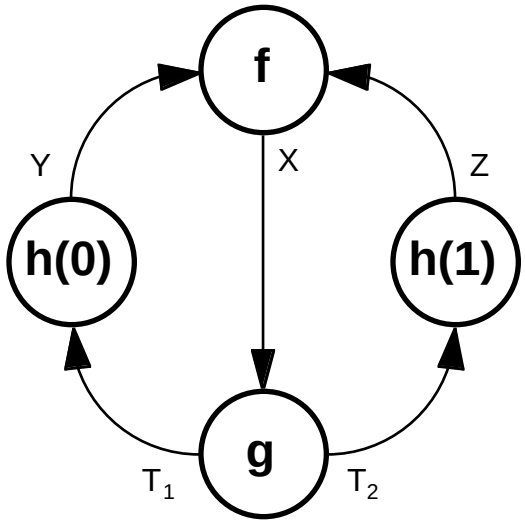

Figure 1.4: Graphical representation of a process network constructed with the processes defined in figures 1.1, 1.2 and 1.3.

minate programs. This use of host and coordination languages is the approach taken in our implementation of process networks, described later in chapter 5.

Figure 1.4 shows a graphical definition of a network built with instances of the processes defined in figures 1.1, 1.2 and 1.3. The two instances of process *h* produce the initial tokens that allow the program graph to execute. In this example, every process is part of a directed cycle, so output tokens must propagate around the graph to provide the next set of input tokens for a process. This process network is strictly bounded because the directed cycles ensure that there are never more than two tokens buffered on any communication channel, and it is non-terminating because the program can execute forever without stopping. We will later prove that this program produces an infinite sequence of 0's and 1's: $X = [0, 1, 0, 1, \ldots]$. Because of the determinate nature of the computation model, this result is independent of the execution order: Kahn process networks can be executed sequentially or in parallel with the same outcome.

This model of computation supports both recurrence relations and recursion. In a recurrence relation, a stream is defined in terms of itself. Recurrence relations appear as directed cycles in the program graph, as in the example of figure 1.4. It is also possible to have recursion, where a process is defined in terms of itself. A simple example of this is shown in figure 1.5. A more complicated example in figure 1.6 shows a recursive definition of process f from figure 1.1. The cons, first and rest operations are defined later in section 1.2.2. As the process executes, it replaces itself by a subgraph. This is a recursive definition because the process appears in the definition of this subgraph. Non-recursive reconfigurations, where a process does not appear in the definition of the subgraph that replaces it, are also allowed.

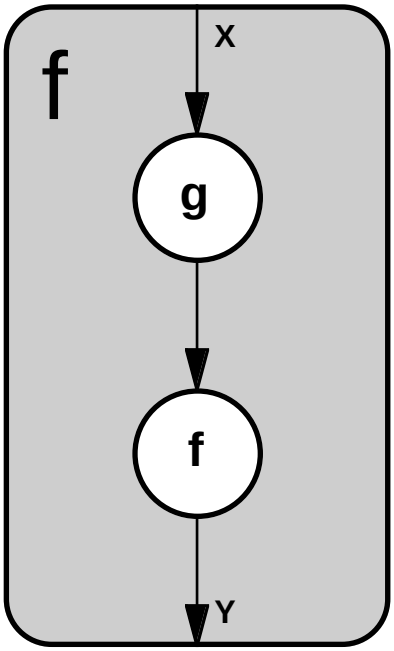

Figure 1.5: Recursive definition of a process.

1.2 Mathematical Representation

We now review Kahn's formal, mathematical representation of process networks [24]. Communication channels are represented by streams and processes are functions that map streams into streams. This allows us to describe a process network by a set of equations. The least fixed point of these equations is unique and corresponds to the histories of the streams in the network. Thus these histories are determined by the definitions of the processes and the network describing the communication between them. The number of tokens produced, and their values, are determined by the definition of the system and not by the scheduling of operations. This is a key result that supports further results later in this thesis.

1.2.1 Streams

A stream is a finite or infinite sequence of data elements: $X = [x_1, x_2, x_3, \ldots]$. The empty stream is represented by the symbol $\perp$. Consider a *prefix ordering* of sequences, where the sequence X *precedes* the sequence Y (written $X \sqsubseteq Y$) if X is a prefix of (or is equal to) Y. For example, the sequence $X = [0]$ is a prefix of the sequence $Y = [0,1]$, which is in turn a prefix of $Z = [0,1,2,\ldots]$. The empty sequence $\perp$ is a prefix of all sequences: $\forall X,\ \perp \sqsubseteq X$. Any increasing chain $\vec{X} = (X_1, X_2, \ldots)$ with $X_1 \sqsubseteq X_2 \sqsubseteq \cdots$ has a least upper bound $\sqcup\vec{X}$ which can be interpreted as a limit:

$$\lim_{i\to\infty} X_i = \sqcup\vec{X} \tag{1.1}$$

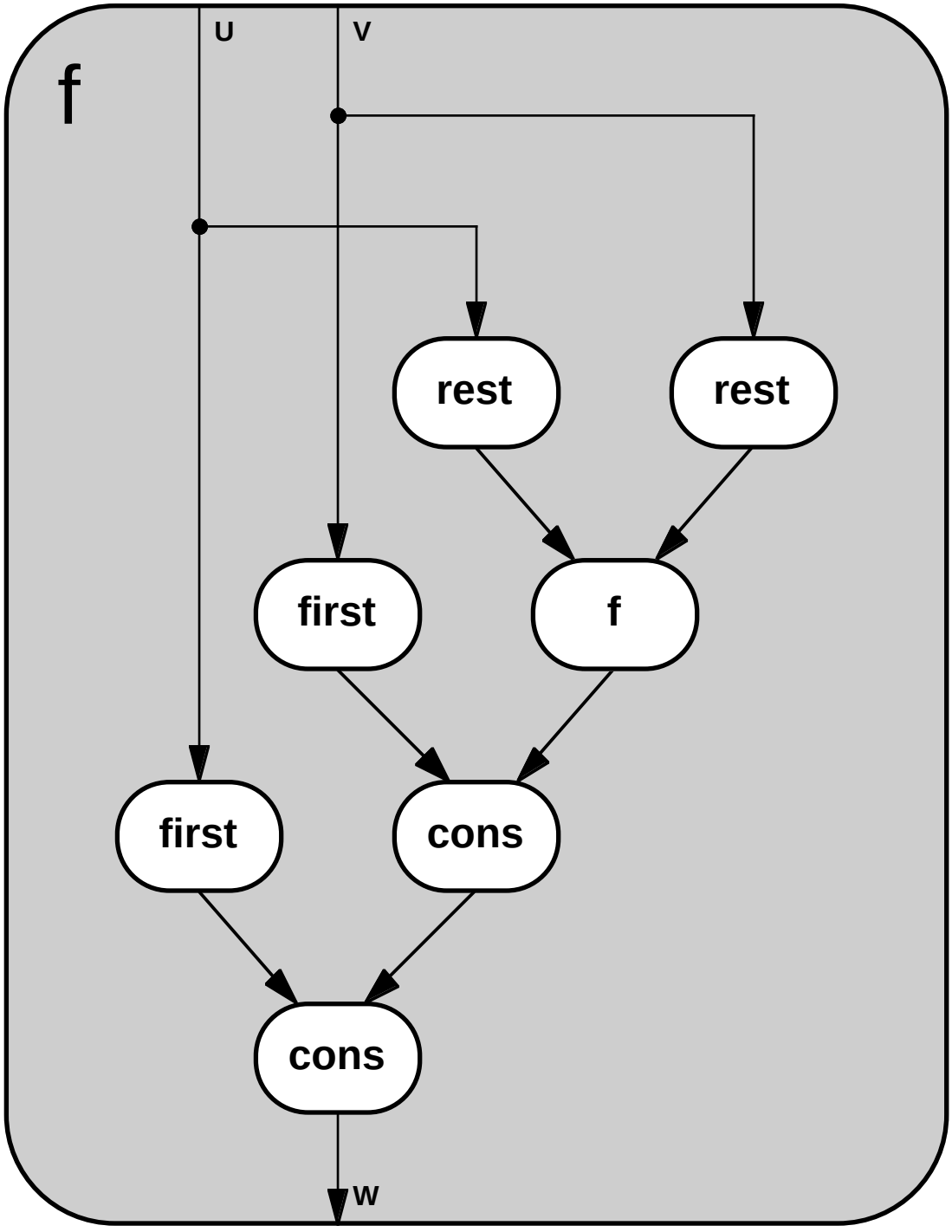

Figure 1.6: Recursive definition of the process f from figure 1.1.

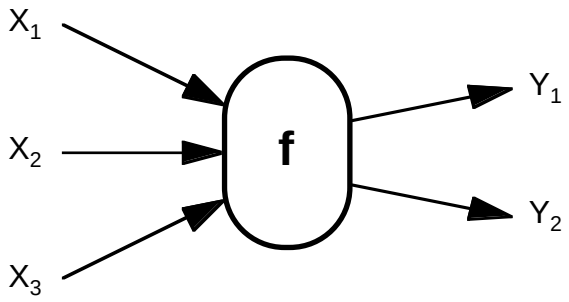

Figure 1.7: Graphical representation of a functional process with multiple inputs and outputs.

The set of all streams is a *complete partial order* with $\sqsubseteq$ defining the ordering. The order is *complete* because every increasing chain of streams has a least upper bound that is itself a stream. Tuples of streams, such as (X_1, X_2), also form a complete partial order. In this case $(X_1, X_2) \sqsubseteq (Y_1, Y_2)$ if and only if $X_1 \sqsubseteq Y_1$ and $X_2 \sqsubseteq Y_2$.

1.2.2 Processes

A process is a functional mapping from input streams to output streams. For each process, we can write an equation that describes this functional mapping. For example, the following equation describes the process shown in figure 1.7:

$$(Y_1, Y_2) = \mathrm{f}(X_1, X_2, X_3) \tag{1.2}$$

A functional mapping is *continuous* if and only if for any increasing chain $X_1 \sqsubseteq X_2 \sqsubseteq \cdots$

$$\mathrm{f}(\lim_{i\to\infty} X_i) = \lim_{i\to\infty} \mathrm{f}(X_i) \tag{1.3}$$

The limit on the right hand sided of equation 1.3 exists if the function f maps an increasing chain into another increasing chain. Such functions are *monotonic*:

$$X \sqsubseteq Y \implies \mathrm{f}(X) \sqsubseteq \mathrm{f}(Y) \tag{1.4}$$

The following functions, are examples of continuous mappings:

first(U) Returns the first element of the stream U. By definition, first$(\perp) = \perp$.

rest(U) Returns the stream U with the first element removed. By definition, rest$(\perp) = \perp$.

cons(x, U) Inserts a new element x at the beginning of the stream U. By definition, cons$(\perp, U) = \perp$, and cons$(x, \perp) = [x]$.

The processes described in figures 1.1, 1.2 and 1.3 can be defined by composing these three basic functions and thus are also continuous mappings:

$$\mathrm{f}(U,V) = \mathrm{cons}(\mathrm{first}(U), \mathrm{cons}(\mathrm{first}(V), \mathrm{f}(\mathrm{rest}(U), \mathrm{rest}(V)))) \tag{1.5}$$

$$\mathrm{g}(U) = (\mathrm{g}_1(U), \mathrm{g}_2(U)) \tag{1.6}$$

$$\mathrm{g}_1(U) = \mathrm{cons}(\mathrm{first}(U), g_1(\mathrm{rest}(\mathrm{rest}(U)))) \tag{1.7}$$

$$\mathrm{g}_2(U) = \mathrm{cons}(\mathrm{first}(\mathrm{rest}(U)), g_2(\mathrm{rest}(\mathrm{rest}(U)))) \tag{1.8}$$

$$\mathrm{h}(U,x) = \mathrm{cons}(x,U) \tag{1.9}$$

The definition of process f in figure 1.6 is a graphical representation of the definition in equation 1.5

1.2.3 Fixed Point Equations

Using the mathematical representation of communication channels and processes, we can describe a process network as a set of equations. The example from figure 1.4 can be described by the following equations:

$$(T_1, T_2) = \mathrm{g}(X) \tag{1.10}$$

$$X = \mathrm{f}(Y, Z) \tag{1.11}$$

$$Y = \mathrm{h}(T_1, 0) \tag{1.12}$$

$$Z = \mathrm{h}(T_2, 1) \tag{1.13}$$

This collection can be reduced to a single equation:

$$(T_1, T_2) = \mathrm{g}(\mathrm{f}(\mathrm{h}(T_1, 0), \mathrm{h}(T_2, 1))) \tag{1.14}$$

If the functions are continuous mappings over a complete partial order, then there is a unique least fixed point for this set of equations, and that solution corresponds to the histories of tokens produced on the communication channels [24]. For example, the least fixed point for the equation $X = \mathrm{f}(X)$ is the least upper bound of the increasing chain $[\bot \sqsubseteq \mathrm{f}(\bot) \sqsubseteq \mathrm{f}(\mathrm{f}(\bot)) \sqsubseteq \ldots]$. This gives us an iterative procedure to solve for the least fixed point. In the initial step, set all streams to be empty. In this example, there is just one stream, so $X^0 = \bot$. Then for each stream compute $X^{j+1} = \mathrm{f}(X^j)$ and repeat. For a terminating program, this procedure will stop at some j where $X^{j+1} = X^j$. For a non-terminating program, some or all of the streams will be infinite in length, so this procedure will not terminate. Instead, induction can be used to find the solution.

Returning to our previous example, the solution of the fixed point equation proceeds as follows:

$$(T_1, T_2)^0 = (\bot, \bot) \tag{1.15}$$

$$(T_1, T_2)^1 = \mathrm{g}(\mathrm{f}(\mathrm{h}(\bot, 0), \mathrm{h}(\bot, 1))) = ([0], [1]) \tag{1.16}$$

$$(T_1, T_2)^2 = \mathrm{g}(\mathrm{f}(\mathrm{h}([0], 0), \mathrm{h}([1], 1))) = ([0,0], [1,1]) \tag{1.17}$$

$$(T_1, T_2)^3 = \mathrm{g}(\mathrm{f}(\mathrm{h}([0,0], 0), \mathrm{h}([1,1], 1))) = ([0,0,0], [1,1,1]) \tag{1.18}$$

$$(T_1, T_2)^{j+1} = \mathrm{g}(\mathrm{f}(\mathrm{h}(T_1^j, 0), \mathrm{h}(T_2^j, 1))) = ([0,0,0\ldots], [1,1,1\ldots]) \tag{1.19}$$

Using induction, we can prove that $T_1 = [0,0,0\ldots]$ and $T_2 = [1,1,1\ldots]$. From this we conclude that $Y = \mathrm{h}(T_1,0) = [0,0,0\ldots]$ and $Z = \mathrm{h}(T_2,1) = [1,1,1\ldots]$. This gives us $X = \mathrm{f}(Y,Z) = [0,1,0,1\ldots]$ as claimed earlier.

Fixed point equations can also be used to describe recursive process networks. Continuous functions on streams are themselves a complete partial order where $\mathrm{f} \sqsubseteq \mathrm{g}$ if and only if $\forall X,\ \mathrm{f}(X) \sqsubseteq \mathrm{g}(X)$ [24]. If we use the recursive definition of f given in figure 1.6 and equation 1.5, then the equations for our example become:

$$(T_1,T_2) = \mathrm{g}(\mathrm{f}(Y,Z)) \tag{1.20}$$

$$\mathrm{f}(Y,Z) = \mathrm{cons}(\mathrm{first}(Y),\mathrm{cons}(\mathrm{first}(Z),\mathrm{f}(\mathrm{rest}(Y),\mathrm{rest}(Z)))) \tag{1.21}$$

$$Y = h(T_1,0) \tag{1.22}$$

$$Z = h(T_2,0) \tag{1.23}$$

Now the function f appears on the left-hand side along with the other unknowns.

1.3 Determinism

We say that a process network program is *determinate* if the results of the computation (the tokens produced on the communication channels) do not depend on the execution order. Every execution order that obeys the semantics of the process network will produce the same result. Kahn uses the fact that the equations for a process network have a unique least fixed point to prove that programs are determinate[24]. Karp and Miller[26] prove that computation graphs, a restricted model of computation similar to process networks, are also determinate.

1.3.1 Execution Order

We define the *execution order* of a process network to be the order of the get and put operations. If $X = [x_1,x_2,x_3\ldots]$ is a stream, then $\mathrm{put}(x_1)$ represents the writing or production of the element x_1 and $\mathrm{get}(x_1)$ represents the reading or consumption of the element x_1. A data element must be written before it can be read, so $\mathrm{put}(x_i) \leq \mathrm{get}(x_i)$ for every element of every stream in a process network. The first-in, first-out (FIFO) nature of the communication channels also imposes the restriction that $\mathrm{get}(x_i) \leq \mathrm{get}(x_j)$ if and only if $\mathrm{put}(x_i) \leq \mathrm{put}(x_j)$. We allow the possibility of some operations occurring simultaneously. For example, if a process produces two elements x_i and x_{i+1} simultaneously, then $\mathrm{put}(x_i) = \mathrm{put}(x_{i+1})$. Processes introduce additional restrictions. For example, if $W = \mathrm{f}(U,V)$, where f is the process defined in figure 1.1, then we have the following restrictions. The process reads alternately from the two input streams U and V,

$$\mathrm{get}(u_i) \leq \mathrm{get}(v_i) \leq \mathrm{get}(u_{i+1}) \leq \mathrm{get}(v_{i+1}) \leq \cdots \tag{1.24}$$

The process writes to stream W after reading from each input stream U and V,

$$\mathrm{get}(u_i) \leq \mathrm{put}(w_{2i}) \leq \mathrm{get}(v_i) \leq \mathrm{put}(w_{2i+1}) \leq \cdots \tag{1.25}$$

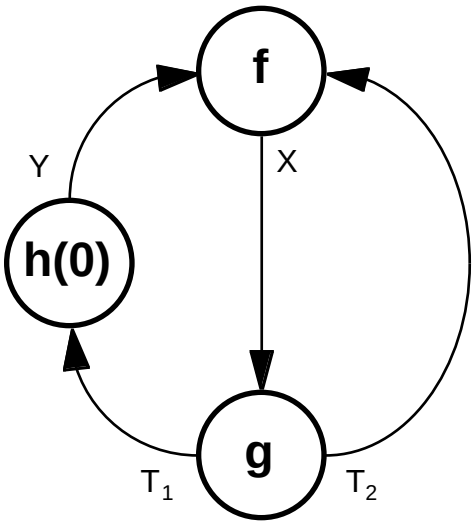

Figure 1.8: A terminating process network.

A *sequential* execution is a *total ordering* of get and put operations. We can compare any pair of operations and know if $\text{get}(x_i) \leq \text{put}(y_j)$ or $\text{put}(y_j) \leq \text{get}(x_i)$, for example. A *parallel* execution is a *partial ordering* of get and put operations. We may be able to compare some operations, such as $\text{put}(x_1) \leq \text{put}(x_2)$ and $\text{get}(x_1) \leq \text{get}(x_2)$. But we will not be able to compare other operations, such as $\text{get}(x_1)$ and $\text{put}(x_2)$.

Any execution order must satisfy the restrictions imposed by the communication channels and the processes. Unlike the restrictions imposed by the semantics of the computation model, such as write-before-read and FIFO channels, we may not always know the restrictions imposed by a particular process, depending on the information available to us and the complexity of its definition. In some restricted process network models, the processes are fully characterized. Generally, there will be many possible execution orders that satisfy these restrictions.

1.3.2 Termination

We proved earlier that all of the streams for the process network shown in figure 1.4 are infinite in length. Thus there are an infinite number of put operations. Consider the modified version of this program shown in figure 1.8. The equation describing this process network is:

$$(T_1, T_2) = \text{g}(\text{f}(\text{h}(T_1, 0), T_2)) \tag{1.26}$$

The least fixed point for this equation is computed as follows:

$$(T_1, T_2)^0 = (\bot, \bot) \tag{1.27}$$

$$(T_1, T_2)^1 = \text{g}(\text{f}(\text{h}(\bot, 0), \bot)) \ = ([0], \bot) \tag{1.28}$$

$$(T_1, T_2)^2 = \text{g}(\text{f}(\text{h}([0], 0), \bot)) = ([0], \bot) \tag{1.29}$$

We can see that $(T_1, T_2)^2 = (T_1, T_2)^1 = ([0], \perp)$, so these streams (and all the others in the network) have finite length. Thus there are only a finite number of put operations.

We see that termination is closely related to determinism. Kahn process networks are determinate because a system of continuous functions over a complete partial order has a unique least fixed point. This solution determines the value, and consequently the length, of every stream in the program. If *all* of the streams are finite in length, then the program must terminate. Otherwise, the program never terminates and produces at least one infinite stream of data. Termination is determined completely by the definition of the process network and is not affected by the particular choice of execution order [44].

The least fixed point solution determines the length of every stream, but it does not determine the order in which the stream elements are produced. Consequently there are many possible execution orders that can lead to the least fixed point. A *complete execution* of a Kahn process network corresponds to the least fixed point — none of the streams can be extended. A *partial execution* does not correspond to a fixed point — one or more of the streams can still be extended.

We define a *terminating* Kahn process network program to be one where all complete executions have a finite number of operations. We define a *non-terminating* program to be one where all complete executions have an infinite number of operations.

1.3.3 Boundedness

Even though the lengths of all streams are determined by the definition of a process network, the number of unconsumed tokens that can accumulate on communication channels depends on the choice of execution order.

A communication channel is defined to be *strictly bounded by b* if the number of unconsumed tokens buffered on the channel cannot exceed *b* for *any* complete execution of the process network. A *strictly bounded* communication channel is one for which there exists a finite constant *b* such that the channel is strictly bounded by *b*. A communication channel is defined to be *bounded by b* if the number of unconsumed tokens cannot exceed *b* for *at least one* complete execution of the process network. A *bounded* communication channel is one for which there exists a finite constant *b* such that the channel is bounded by *b*.

A process network is defined to be *strictly bounded by b* if every channel in the network is strictly bounded by *b*. A *strictly bounded* process network is one for which there exists a finite constant *b* such that it is strictly bounded by *b*. A process network is defined to be *bounded by b* if every channel in the network is bounded by *b*. A *bounded* process network is one for which there exists a finite constant *b* such that it is bounded by *b*. This is actually a weaker condition than insisting that the total number of unconsumed tokens in the network be less than *b*. This allows us to include recursive networks, where there can be an unbounded number of channels, in our definition of bounded systems. A Kahn process network is defined to be *unbounded* if it is not bounded. That is, at least one channel is not bounded for all complete executions of the process network.

Our example from figure 1.4 is a *strictly bounded* program. Each process consumes tokens at the

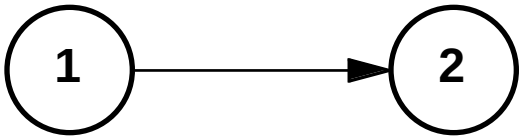

Figure 1.9: A bounded process network.

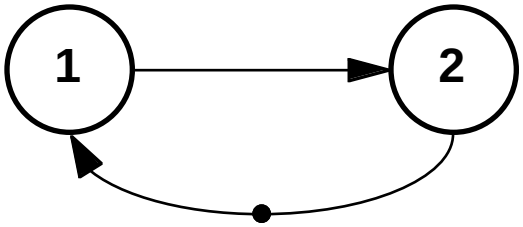

Figure 1.10: A strictly bounded process network.

same rate that it produces them, and the directed cycles in the graph prevent the production of new tokens until the previously produced tokens are consumed. For all execution orders, there can never be more than 1 unconsumed token buffered on channels Y, Z, T_1 and T_2, and no more than 2 unconsumed tokens on channel X. Thus we say that channels Y, Z, T_1 and T_2 are strictly bounded by $b = 1$, and channel X is strictly bounded by $b = 2$. The program is therefore strictly bounded by $b = 2$. However, the simple system shown in figure 1.9 is only bounded. While it is possible to execute this system with bounded buffering, some execution orders lead to unbounded token accumulation. If we activate process 1 but never activate process 2, which is one of many possible execution orders, then an infinite number of tokens accumulate on the communication channel.

An arbitrary Kahn process network can be transformed so that it is strictly bounded. This is done by adding a feedback channel for each data channel and modifying the processes so that they must read from a feedback channel before writing to a data channel. We place b initial tokens on the feedback channels so that all pairs of channels have b initial tokens. The number of tokens in a data channel cannot increase without a corresponding decrease in the number of tokens in the feedback channel. Because the number of tokens for a feedback channel (or any channel) can never fall below zero, the data channels are strictly bounded. However, these restrictions could introduce deadlock, transforming a non-terminating program into a terminating one. Figure 1.10 shows how our simple bounded system of figure 1.9 is transformed into a strictly bounded system. Details of this graph transformation are presented in chapter 4.

1.4 Open and Closed Systems

In an open system there are a set of input channels that provide data to the process network and a set of output channels that receive the results. Consequently, each process has at least one input and one output.

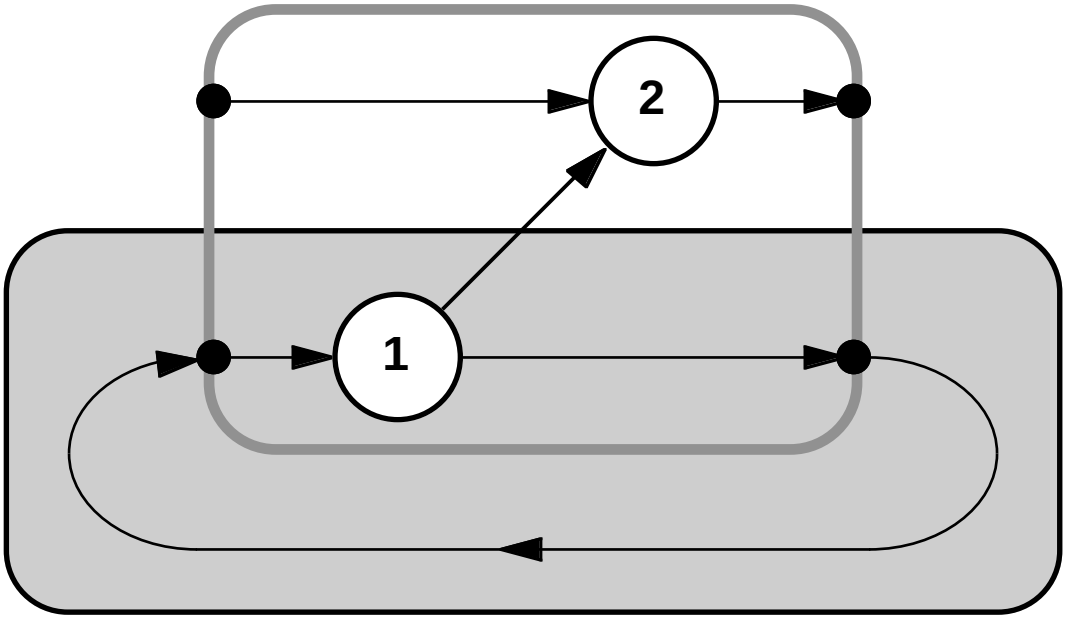

Figure 1.11: A data source (1) constructed with external feedback connections.

In this case, the process network is not a complete system in and of itself, but is a sub-system that is part of a larger, external system.

In a closed system, there are no external input or output channels. Processes that have no input channels are always enabled and act as sources of data. Processes that have no outputs act as data sinks. Input and output operations for this system are managed by individual processes. For example, a process may read data samples from a sensor and produce them on a stream. Or a process may consume data from a stream and write it to a file. In this case, the process network describes a complete system.

In Kahn's process network model, attempting to get data from an input channel is the only operation that may cause the execution of a process to suspend. In closed systems, processes that have no inputs are always enabled; they never attempt to get data from an empty channel. There may never be an opportunity to activate other processes, which can be problematic. In an open system each process has at least one input, so it will be suspended whenever it attempts to get data from an empty channel. At this point, other processes that are enabled can be activated.

Even if every process must have an input, it is possible to build constructs that are data sources, subsystems that require no input from the rest of the system. If feedback connections are allowed in the external system, it is possible to construct a source by connecting an external output to an external input, as shown in figure 1.11. Process 1 is self-enabling, it provides its own input and requires no input from the rest of the system. A strongly connected subgraph, where there is a directed path from any process to any other process (including itself), can also act as a data source, as shown in figure 1.12. Figures 1.13 and 1.14 show similar constructs for data sinks, subsystems that consume their own output and provide no data to the rest of the

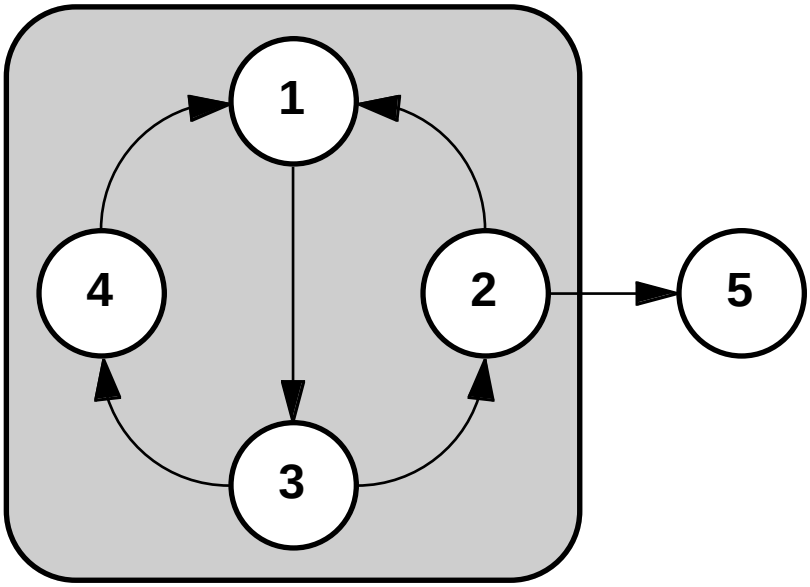

Figure 1.12: A connected subgraph (1,2,3,4) that is a data source.

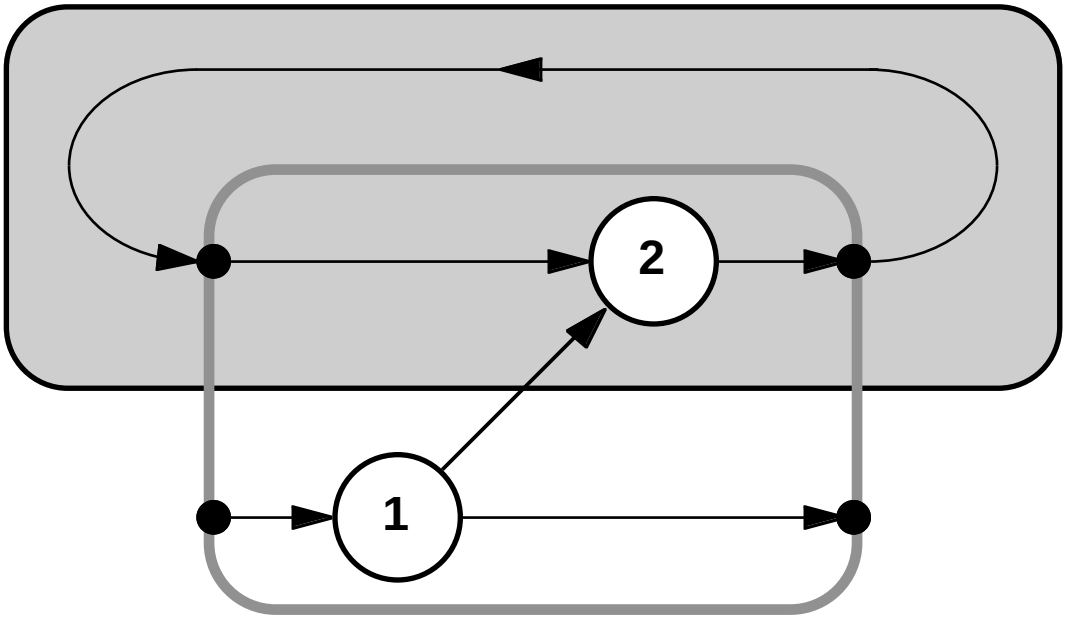

Figure 1.13: A data sink (2) constructed with external feedback connections.

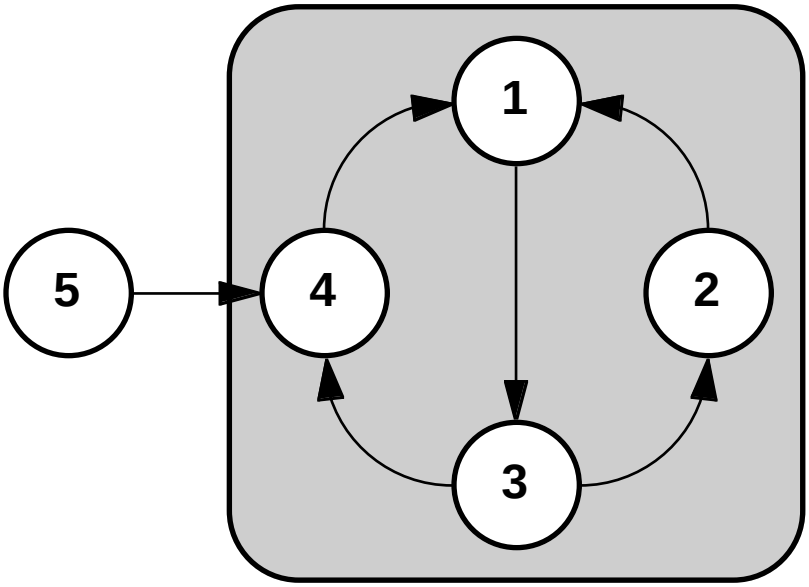

Figure 1.14: A connected subgraph (1,2,3,4) that is a data sink.

system.

Kahn and MacQueen consider both open and closed systems: a program is described as a graph with a set of input and output channels [24], or processes are used for input and output [25]. Pingali and Arvind [37, 39, 38] consider open systems. In demand driven execution, there is the problem of synchronizing with the external system so that inputs are provided only when demanded. Pingali puts gates on the inputs to solve this problem[38]. This makes an open system resemble a closed system where processes (gates in this case) act as data sources and there are effectively no external inputs. We will see an example of this in chapter 3.

Data driven execution of sources and demand driven execution of sinks can be problematic, as we will demonstrate in chapter 3. Sources and sinks, either in the form of individual processes or subnetworks of processes, are present in both open and closed systems. Thus, without loss of generality, we choose to restrict our discussion to closed systems.

Chapter 2

Dataflow Process Networks

Dataflow is a model of computation that is closely related to process networks. Dataflow programs can be described by graphs, just as process networks can be. The arcs of the graph represent FIFO queues for communication, just as in process networks. The nodes of the graph represent actors. Instead of using the blocking read semantics of Kahn process networks, dataflow actors have firing rules. These firing rules specify what tokens must be available at the inputs for the actor to fire. When an actor fires, it consumes some finite number of input tokens and produces some finite number of output tokens. A process can be formed from repeated firings of a dataflow actor so that infinite streams of data may be operated on. We call these dataflow processes[33].

Breaking a process down into smaller units of execution, such as dataflow actor firings, makes efficient implementations of process networks possible. Restricting the type of dataflow actors to those that have predictable token consumption and production patterns makes it possible to perform static, off-line scheduling and to bound the memory required to implement the communication channel buffers. Thus, for some restricted forms of process networks, it is possible to satisfy our requirements for non-terminating execution with bounded buffer sizes.

2.1 Mathematical Representation

Streams represent the sequences of tokens that flow along the arcs in a dataflow graph, just as in process networks. Dataflow actors are represented by functions that map input tokens to output tokens. This is in contrast to the representation of processes as functions that map streams to streams.

Dataflow actors have firing rules that determine when enough tokens are available to enable the actor. When the firing rules are satisfied and sufficient input tokens are available, the actor fires. It consumes a finite number of input tokens and produces a finite number of output tokens. For example, when applied to an infinite input stream a firing function f may consume just one token and produce one output token:

$$\mathrm{f}([x_1, x_2, x_3 \ldots]) = \mathrm{f}(x_1) \tag{2.1}$$

To produce an infinite output stream, the actor must be fired repeatedly. A processes formed from repeated firings of a dataflow actor is called a dataflow process [33]. The higher-order function map converts an actor firing function f into a process F. A higher-order function takes a function as an argument and returns another function. For simple dataflow actors that consume and produce a single token, the higher-order function map behaves as follows [27]:

$$\mathrm{map}(\mathrm{f})\,[x_1, x_2, x_3 \ldots] = [\mathrm{f}(x_1), \mathrm{f}(x_2), \mathrm{f}(x_3) \ldots] \tag{2.2}$$

When the function returned by $\mathrm{map}(\mathrm{f})$ is applied to the stream $[x_1, x_2, x_3 \ldots]$, the result is a stream in which the firing function f is applied point-wise to the elements of that stream, $[\mathrm{f}(x_1), \mathrm{f}(x_2), \mathrm{f}(x_3) \ldots]$. The map function can also be described recursively:

$$\mathrm{map}(\mathrm{f})\,(X) = \mathit{cons}(\mathrm{f}(\mathrm{first}(X)), \mathrm{map}(\mathrm{f})\,(\mathrm{rest}(X))) \tag{2.3}$$

The use of map can be generalized for firing functions f that consume and produce multiple tokens on multiple streams [33]. We will see this in more detail in the next section.

2.1.1 Firing Rules

Firing rules specify the pattern of tokens that must be available at each input to enable an actor. Actors with no input streams are always enabled. Once enabled, an actor can fire. Actors with one or more input streams can have several firing rules.

$$R = \{\vec{R}_1, \vec{R}_2, \ldots, \vec{R}_N\} \tag{2.4}$$

An actor can fire if and only if one or more of the firing rules are satisfied. For an actor with p input streams, each firing rule $\vec{R}_i$ is a p-tuple of *patterns*, one pattern for each input.

$$\vec{R}_i = (R_{i,1}, R_{i,2}, \ldots, R_{i,p}) \tag{2.5}$$

Each pattern $R_{i,j}$ is a finite sequence. For a firing rule $\vec{R}_i$ to be satisfied, each pattern $R_{i,j}$ must be a prefix of the sequence of tokens available on the corresponding input, $R_{i,j} \sqsubseteq X$.

For an actor firing that consumes no tokens from an input, the pattern for that input is $R_{i,j} = \perp$. Because $\perp \sqsubseteq X$ for any sequence X, any sequence of available tokens is acceptable, $\forall X,\ R_{i,j} \sqsubseteq X$. Note that an empty pattern $R_{i,j} = \perp$ does not mean that the input must be empty. For an actor firing that consumes a finite number of tokens from an input, the pattern is of the form $R_{i,j} = [*, *, \cdots, *]$. The symbol $*$ is a token wildcard. The sequence $[*]$ is a prefix of any sequence with one or more tokens. The sequence $[*, *]$ is a prefix of any sequence with two or more tokens. Only $\perp$ is a prefix of $[*]$. For an actor firing that requires input tokens to have particular values, the pattern $R_{i,j}$ includes this data value.

For example, the switch actor in figure 2.1(a) has a single firing rule $\vec{R} = ([*], [*])$. It consumes a single token from each of its inputs. The first token is copied to either the "TRUE" or "FALSE" output

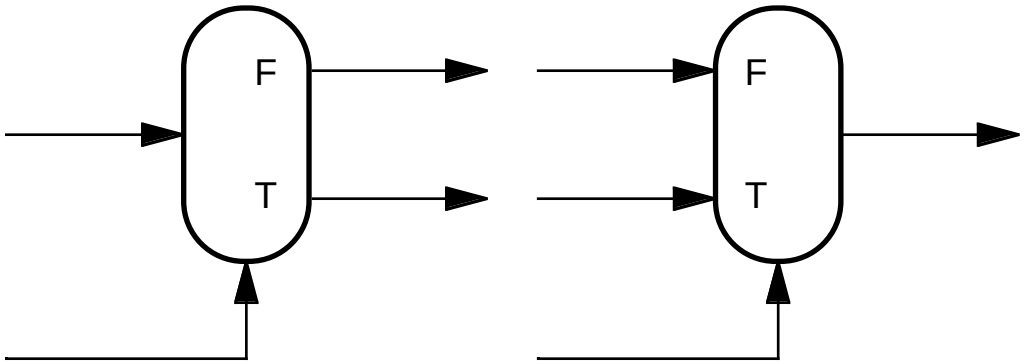

Figure 2.1: The dataflow actors switch and select.

depending on the value of the control token. The select actor in figure 2.1(b) has two firing rules:

$$\vec{R}_1 = ([*], \bot, [F]) \tag{2.6}$$

$$\vec{R}_2 = (\bot, [*], [T]) \tag{2.7}$$

When the control token has the value "FALSE," rule $\vec{R}_1$ applies and one token is copied from the "FALSE" input to the output. When the control token has the value "TRUE," rule $\vec{R}_2$ applies and one token is copied from the "TRUE" input to the output. Notice that the firing rules do not capture any information about the number or values of tokens produced when an actor fires.

If $\vec{X} = (X_1, X_2, \ldots, X_p)$ are the sequences of tokens available on an actor's inputs, then the firing rule $\vec{R}_i$ is satisfied if

$$\forall R_{i,j} \in \vec{R}_i, \quad R_{i,j} \sqsubseteq X_j \tag{2.8}$$

Firing rules that can be implemented as a sequence of blocking read operations is defined to be *sequential firing rules* [33]. The firing rules of the select actor are sequential: a blocking read of the control input is followed by a blocking read of the appropriate data input.

An example of an actor with firing rules that are not sequential is the non-determinate merge in figure 2.2. Its firing rules are

$$\vec{R}_1 = ([*], \bot) \tag{2.9}$$

$$\vec{R}_2 = (\bot, [*]) \tag{2.10}$$

As soon as a token is available on either input, it is copied to the output. If tokens are available on both inputs, then both firing rules are satisfied. There is ambiguity about which rule should be used. For actors with sequential firing rules, there is no such ambiguity. For any set of available tokens, no more than one firing rule

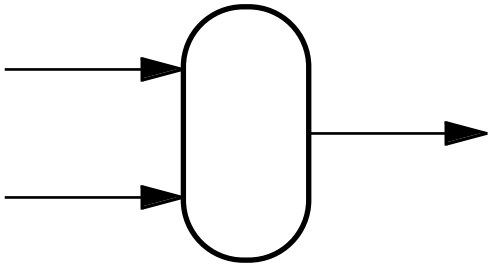

Figure 2.2: The non-determinate merge actor.

is satisfied. The merge actor is non-determinate because the order in which tokens are produced depends on the order in which the inputs become available. The firing rules are not sequential because a blocking read of either input fails to produce the desired behavior. A blocking read on one input causes tokens on the other input to be ignored.

Now we can define map in terms of firing rules:

$$\mathrm{map(f)(cons}(\vec{R},\vec{X})) = \mathrm{cons(f}(\vec{R}), \mathrm{map(f)}(\vec{X})) \tag{2.11}$$

where $\vec{R}$ is a firing rule of f, and cons has been generalized to operate on p-tuples of streams.

Because sequential firing rules can be implemented as a sequence of blocking read operations, dataflow processes are continuous when the firing rules are sequential [33]. Because networks of continuous processes are determinate [24], dataflow process networks are determinate when each actor's firing rules are sequential.

2.1.2 Execution Order

We define the execution order of a process network to be the order of the get and put operations. When an actor fires, it consumes input tokens and produces output tokens. Because an actor firing is atomic, an order on the firings imposes an order on the get and put operations. Thus, we define the execution order of a dataflow program to be the order of actor firings.

When all the firing rule patterns are of the form $[*, *, \ldots, *]$, we can define the *state* of a dataflow graph to be the number of tokens on each arc. When some of the firing rule patterns are data-dependent, as they are for the select actor, then the state of the graph must also include the values of tokens on the control arcs. As actors fire, the graph proceeds from one state to the next. The current state determines which actors are enabled, and thus the set of possible next states. This assumes that actors are defined functionally and have no internal state that affects the firing rules. Actor state must be represented explicitly with an arc connected as a self loop so that its effects on the firing rules become apparent.

2.2 Computation Graphs

Computation graphs [26] are a model of parallel computation similar to process networks. A parallel computation is represented by a finite graph. There is a set of nodes $n_1, \ldots, n_l$, each associated with a function $O_1, \ldots, O_l$. There is a set of arcs $d_1, \ldots, d_t$, where a branch d_p is a queue of data directed from one node n_i to another n_j. Four non-negative integer constants A_p, U_p, W_p and T_p are associated with each arc d_p. These parameters have the following interpretation:

A_p The number of data tokens initially present on the arc d_p.

U_p The number of data tokens produced by the function O_i associated with node n_i.

W_p The number of data tokens consumed by the function O_j associated with node n_j.

T_p A threshold that specifies the minimum number of tokens that must be present on d_p before O_j can be fired. Clearly, $T_p \geq W_p$ must be true.

An operation O_j is enabled when the number of tokens present on d_p is greater than or equal to T_p for every arc leading into n_j. When it is fired, the operation O_j consumes W_p tokens from each input arc d_p, and produces U_q tokens on each output arc d_q. These rules determine which execution sequences are possible. Thus the firing rule patterns are all of the form $[*, *, \ldots, *]$ and any input sequence of length T_p or greater satisfies the firing rules.

The execution of a computation graph is described by a sequence of non-empty sets of operations, $S_1, S_2, \ldots, S_N, \ldots$. Initially, S_1 is a subset of the enabled operations with $A_p \geq T_p$ for each input arc d_p. When the operations in S_1 are fired, the data they consume may prevent them from being enabled, $A_p - W_p < T_p$, and the data they produce may enable other operations, $A_q + U_q \geq T_q$. Each set S_N is a subset of the operations enabled after all of the operations in S_{N-1} have fired.

Due to the restrictions placed on the computation model, Karp and Miller [26] are able to give necessary and sufficient conditions for computation graphs to terminate. They also give necessary and sufficient conditions for the queue lengths to be bounded. Thus the questions of termination and boundedness are decidable for computation graphs, a restricted form of process network.

2.3 Synchronous Dataflow

Synchronous dataflow [31, 32] is a special case of computation graphs where $T_p = W_p$ for all arcs in the graph. Because the number of tokens consumed and produced by an actor is constant for each firing, we can statically construct a finite schedule that is then periodically repeated to implement a dataflow process network that operates on infinite streams of data tokens.

A synchronous dataflow graph can be described by a topology matrix Γ with one row for each arc and one column for each node. This is only a partial description because there is no information regarding the number of initial tokens on each arc. The element Γ_{ij} is defined as the number of tokens produced on the ith

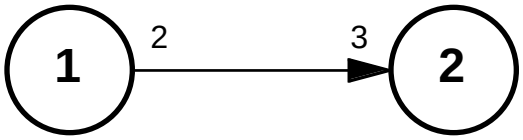

Figure 2.3: A bounded synchronous dataflow program.

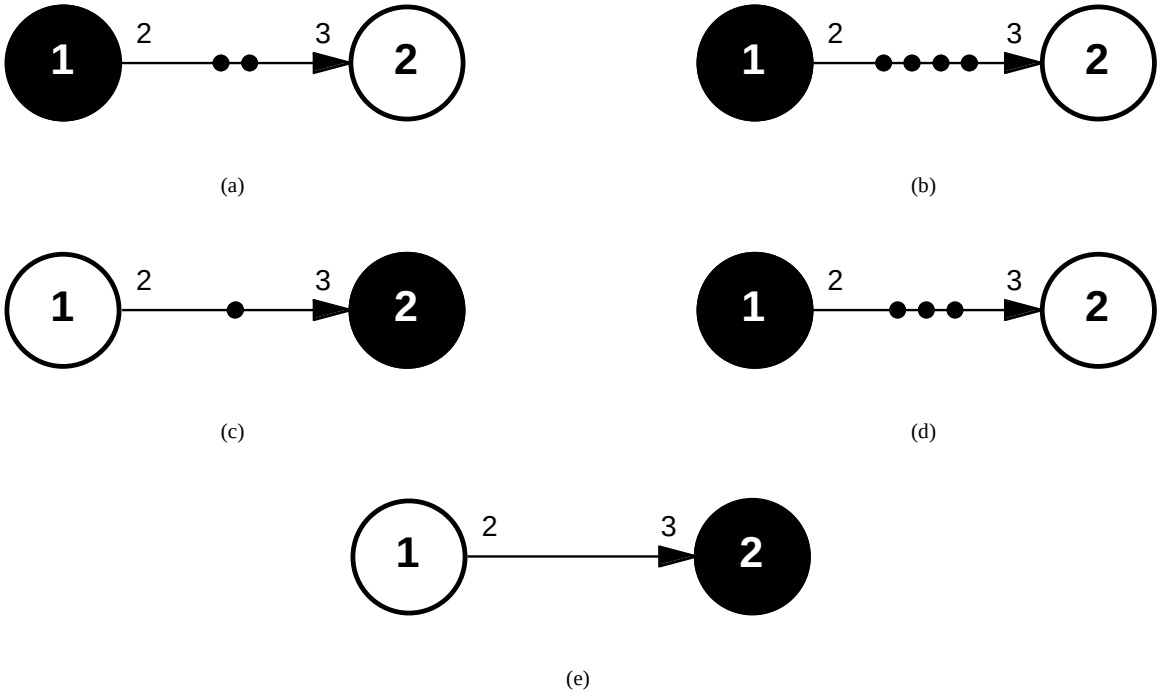

Figure 2.4: A sequence of actor firings that brings the program of figure 2.3 back to its original state.

arc by the jth actor. A negative value indicates that the actor consumes tokens on that arc. Each row of the matrix has one positive element for the actor that produces tokens on the corresponding arc and one negative element for the actor that consumes tokens from the arc. If the same actor consumes and produces tokens on the arc, then entry Γ_{ij} is the difference between the number of tokens produced and the number consumed. All the other elements in the row are zero. For the system to be balanced, a non-trivial, positive repetition vector $\vec{r}$ must be found that satisfies the balance equations:

$$\Gamma\vec{r} = \vec{0} \tag{2.12}$$

where each element r_j of the repetition vector specifies a number of firings for the jth SDF actor, and $\vec{0}$ is the zero vector.

For example, consider the synchronous dataflow graph in figure 2.3. Actor 1 produces 2 tokens each

time it fires, and actor 2 consumes 3 tokens each time it fires. For the system to be balanced, actor 2 must fire 2 times for every 3 firings of actor 1. Actor 1 fires first, producing 2 tokens, shown in figure 2.4(a). Actor 2 is not enabled yet, so we fire actor 1 again. Now there are 4 unconsumed tokens on the arc, shown in figure 2.4(b), and actor 2 can fire. It consumes 3 tokens leaving 1 on the arc, shown in figure 2.4(c). Actor 2 is no longer enabled, so we fire actor 1 a third time. It produces 2 more tokens so that there are 3 on the arc, shown in figure 2.4(d). Now actor 2 fires for a second time and consumes all 3 of the tokens on the arc. At this point the graph has returned to its original state with no tokens on the arc, shown in figure 2.4(e).

The topology matrix for this simple example is

$$\Gamma = \begin{bmatrix} 2 & -3 \end{bmatrix} \tag{2.13}$$

Our intuition tells us that the repetition vector that solves the balance equations should be

$$\vec{r} = \begin{bmatrix} 3 & 2 \end{bmatrix}^T \tag{2.14}$$

This would fire actor 1 three times and actor 2 twice so that the total number of tokens produced is equal to the total number consumed. It is easy to verify that this repetition vector does indeed solve equation 2.12.

The state of a synchronous dataflow graph is the number of tokens on each arc. A *complete cycle* is a sequence of actor firings that returns the graph to its original state. Because the total number of tokens produced in a complete cycle is equal to the total number of tokens consumed, there is no net change in the number of tokens. Because the number of actor firings in the cycle is finite and the number of tokens produced by each firing is finite, the number of unconsumed tokens that can accumulate before an arc returns to its original state is bounded. This finite sequence of firings can be repeated indefinitely so that the program can execute forever with only a bounded number of unconsumed tokens accumulating on the arcs.

There can be many possible firing sequences for a given dataflow graph. In this example we have chosen to fire the actors by repeating the sequence 1,1,2,1,2 but we could just as well have chosen the firing sequence 1,1,1,2,2. If we repeat a complete cycle, a firing sequence that is consistent with the balance equations, then only a bounded number of unconsumed tokens can accumulate before the graph returns to its initial state. The precise value of the bound depends on the firing sequence chosen. For the firing sequence 1,1,2,1,2 the bound is 4 tokens. For the firing sequence 1,1,1,2,2 the bound is 6 tokens. There are also firing sequences that, when repeated, result in unbounded token accumulation. These sequences do not correspond to a solution of the balance equations and thus do not form a complete cycle. The firing sequence 1,1,2 is an example of such a sequence. At the completion of this firing sequence there is a net increase of 1 token on the arc. If this firing sequence is repeated indefinitely, then the number of unconsumed tokens grows without bound. As long as we restrict ourselves to firing sequences that represent finite complete cycles, we know that the program will execute in bounded memory.

Consider the synchronous dataflow graph in figure 2.5. The topology matrix for this example is:

$$\Gamma = \begin{bmatrix} 2 & -3 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \end{bmatrix} \tag{2.15}$$

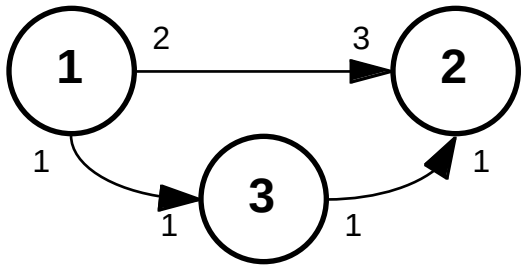

Figure 2.5: An unbalanced synchronous dataflow program.

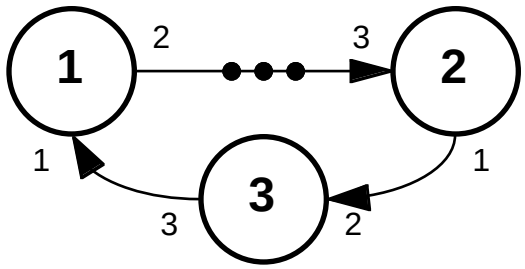

Figure 2.6: A deadlocked synchronous dataflow program.

We can see that this graph is unbalanced because the only solution to the balance equations is the zero vector. This program can still execute forever, but tokens accumulate without bound.

We have seen how the balance equations help us identify systems that can be executed with bounded buffering of tokens on the arcs. However, the balance equations do not completely describe a synchronous dataflow graph. We must also know the initial state of the graph, the number of initial tokens on each arc. Even when the balance equations have a non-trivial positive integer solution, it may not be possible to execute a program indefinitely, or tokens may accumulate without bound if the program is executed indefinitely.

Consider the example in figure 2.6. The topology matrix for this graph is:

$$\Gamma = \begin{bmatrix} 2 & -3 & 0 \\ -1 & 0 & 3 \\ 0 & 1 & -2 \end{bmatrix} \tag{2.16}$$

The smallest integer solution to the balance equations is

$$\vec{r} = \begin{bmatrix} 3 & 2 & 1 \end{bmatrix}^T \tag{2.17}$$

This system is balanced, but it is also deadlocked because there are not enough initial tokens on the arcs in the directed cycle. Initially only actor 2 is enabled. It can fire once, consuming all of the tokens that were initially on its input and producing 1 token on its output. Actor 3 requires 2 tokens in order to fire, but only 1 is available. There are no tokens on any of the other arcs, so we have reached deadlock: none of the actors

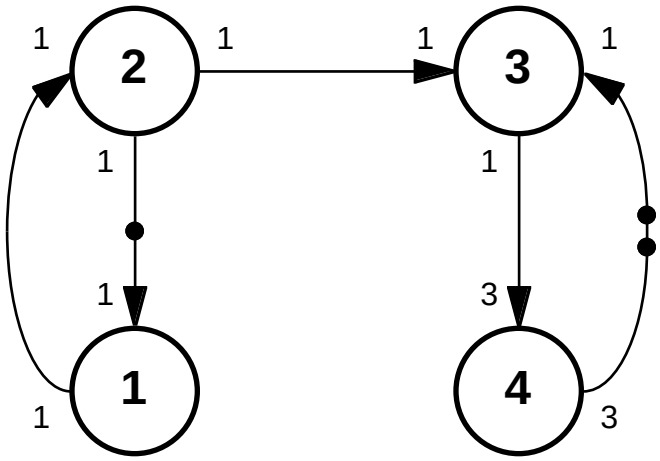

Figure 2.7: A balanced synchronous dataflow program that requires unbounded memory.

can fire. Having a solution to the balance equations is not enough to guarantee the existence of a complete cycle. There must also be enough initial tokens on the directed cycles so that the program does not deadlock.

One simple way to test for the existence of a complete cycle is through simulated execution [31, 32], simply keeping track of the number of tokens on each arc without actually performing any computation. If a complete cycle is found, then the program will not deadlock when executed. However, as we shall see, the existence of a complete cycle is only a sufficient condition for ruling out deadlock. Even if a complete cycle does not exist, the program may not deadlock when executed. Even so, this is not an insurmountable problem. Karp and Miller[26] give necessary and sufficient conditions to detect deadlock in computation graphs.

Consider the example in figure 2.7. Notice that actors 1 and 2 can fire infinitely often, actor 3 can fire only twice, and actor 4 can never fire. This program can run without terminating (actors 1 and 2 can fire indefinitely) but suffers from unbounded accumulation of unconsumed tokens (actor 3 can consume only 2 of the tokens produced by actor 2). The topology matrix for this program is:

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & -1 & 3 \end{bmatrix} \tag{2.18}$$

The smallest, positive integer solution to the balance equations is

$$\vec{r} = \begin{bmatrix} 3 & 3 & 3 & 1 \end{bmatrix}^T \tag{2.19}$$

The balance equations have a non-trivial solution, but a complete cycle does not exist because there are not enough initial tokens to allow actor 4 to fire. If the program is executed indefinitely, it settles into a cycle, 1,2,3,1,2,3,1,2, . . . ,1,2, . . . , that does not return the system to its initial state. Thus the system is unbalanced

despite the fact that the balance equations have a solution. A better definition of a balanced system is one for which the balance equations have a solution *and* a complete cycle exists. An alternate view is to define this as a deadlocked system because a complete cycle does not exist. However this is counter-intuitive because actors 1 and 2 can fire indefinitely.

We have seen that the balance equations help us identify complete cycles. When each actor is fired the number of times specified by $\vec{r}$, the total number of tokens produced on each arc is equal to the total number of tokens consumed. In a complete cycle, there is no net change in the total number of tokens on any arc, so the system returns to its initial state with the same number of tokens on each arc. The total memory required for the buffers associated with the arcs is bounded because there are a finite number of actor firings in a complete cycle (as specified by the repetition vector), and each actor firing produces a finite number of tokens.

The existence of a complete cycle allows us to execute a program forever with bounded buffer sizes. The balance equations specify the number of actor firings in a complete cycle. Finding a non-trivial solution to the balance equations is necessary but not sufficient for a complete cycle to exist. The initial state of the graph is also required to determine whether a complete cycle exists.

2.4 Boolean Dataflow

Synchronous dataflow programs can be completely analyzed because of the restricted nature of the computation model. Balanced synchronous dataflow process networks can be executed forever in bounded memory, which is our ultimate goal. But can we generalize synchronous dataflow to allow conditional, data-dependent execution and still be able to analyze programs with techniques like the balance equations?

Boolean dataflow[9] is an extension of synchronous dataflow that allows conditional token consumption and production. By adding two simple control actors called switch and select, shown in figure 2.1 on page 19, we can build conditional constructs such as if-then-else and do-while loops. The switch actor gets a control token and then copies a token from the input to the appropriate output, which is determined by the Boolean value of the control token. The select actor gets a control token and then copies a token from the appropriate input, which is determined by the Boolean value of the control token, to the output.

Consider the example in figure 2.8, where switch (actor 3) and select (actor 6) are used to build an if-then-else construct. We can write balance equations for this program, but now the topology matrix has

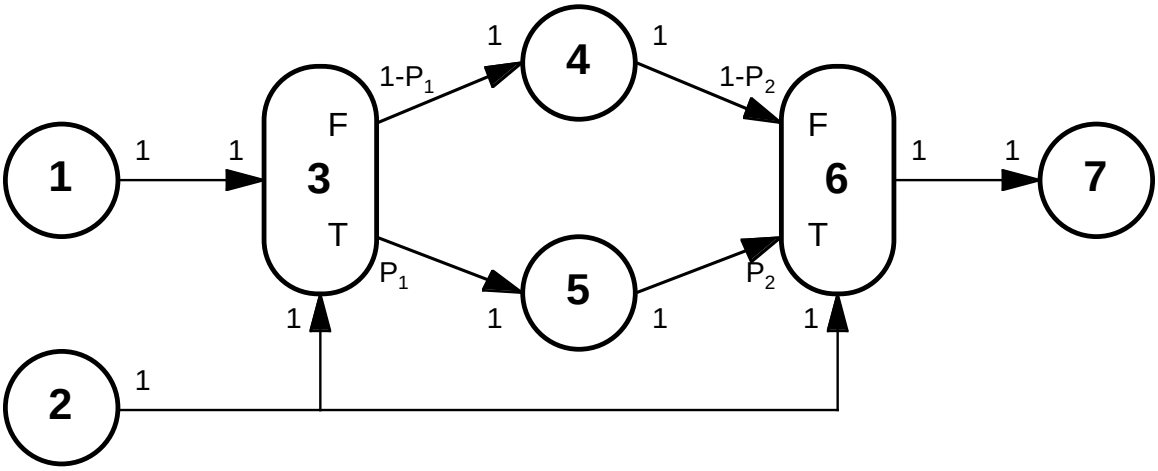

Figure 2.8: A balanced Boolean dataflow program.

symbolic entries:

$$\Gamma = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & (1-P_1) & -1 & 0 & 0 & 0 \\ 0 & 0 & P_1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -(1-P_2) & 0 \\ 0 & 0 & 0 & 0 & 1 & -P_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \tag{2.20}$$

The variables P_1 and P_2 are unknowns, and we can solve the balance equations in terms of these unknowns. For this example, the balance equations $\Gamma\vec{r} = \vec{0}$ have a solution only if $P_1 = P_2$. This restriction is trivially satisfied in this case because the control tokens are derived from the same source (actor 2). Thus the solution is $\vec{r} = [\ 1 \quad 1 \quad 1 \quad (1-P) \quad P \quad 1 \quad 1\ ]^T$ where $P = P_1 = P_2$.

Assume for the moment that this program has a finite complete cycle, just as synchronous dataflow programs have complete cycles. If we let N be the number of control tokens produced by actor 2 in the complete cycle and let T be the number of those tokens that have value "TRUE," then we have $P = \frac{T}{N}$. Now the integer solution to the balance equations has the form $\vec{r} = [\ N \quad N \quad N \quad (N-T) \quad T \quad N \quad N\ ]^T$. The smallest integer solution occurs for $N = 1$, and the system returns to its initial state, with no tokens on any arcs, following a complete cycle.

As in synchronous dataflow, finding a solution to the balance equations is necessary but not sufficient to guarantee the existence of a complete cycle. Consider the example in figure 2.9[16] with the following

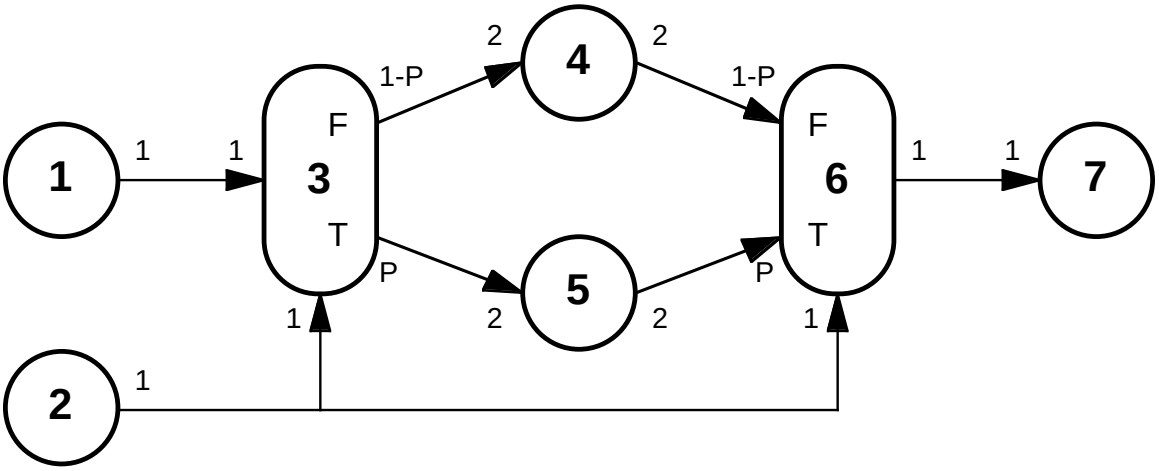

Figure 2.9: A balanced, unbounded Boolean dataflow program.

topology matrix:

$$\Gamma = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & (1-P) & -2 & 0 & 0 & 0 \\ 0 & 0 & P & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & -(1-P) & 0 \\ 0 & 0 & 0 & 0 & 2 & -P & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{bmatrix} \tag{2.21}$$

This is the same system we just studied except that actors 4 and 5 now consume and produce 2 tokens at a time. The solution to the balance equations is $\vec{r} = [\ 2 \quad 2 \quad 2 \quad (1-P) \quad P \quad 2 \quad 2\ ]^T$. Again the existence of a solution does not depend on the value of P, so we might think that this system can be scheduled in bounded memory just as the earlier example. But observe what happens when we attempt to find the minimal integer solution. Using the same definitions of N and T, we find that $\vec{r} = [\ N \quad N \quad N \quad \frac{N-T}{2} \quad \frac{T}{2} \quad N \quad N\ ]^T$. In order for this to have an integer solution we require that N and T be even numbers. But we cannot guarantee this without additional knowledge about the stream of control tokens produced by actor 2. Consider what happens if it produces a "FALSE" token followed by a long stream of "TRUE" tokens. Actor 4 has only one token available, so it is not enabled. Actor 5 fires repeatedly, but all of the tokens on the "TRUE" input of the select (actor 6) accumulate, as do the tokens on its control input. Because the first control token was "FALSE", actor 6 is waiting for tokens on the "FALSE" input.

The answer to the question of whether this system can be executed in bounded memory depends on the values of the tokens produced on the control stream. It may turn out that the sequence of values will allow the system to run in bounded memory, but without special knowledge about this stream, it is impossible to

prove this. We could reject this program because we can't prove that it is bounded, or we could just execute it and hope for the best.

Chapter 3

Dynamic Scheduling of Process Networks

Our goal is to devise a scheduling policy that will execute an arbitrary process network forever with bounded buffering on the communication channels whenever possible. We have seen that for restricted process network models, such as synchronous dataflow, it is possible to construct a finite schedule that can be repeated indefinitely for infinite execution in bounded memory. It is not necessary for synchronous dataflow programs to be strictly bounded where buffer sizes remain bounded for all execution orders. It is enough for the program to be bounded, where at least one execution order yields bounded buffer sizes. Many Boolean dataflow graphs can be similarly analyzed, but this computation model is rich enough that the questions of termination and boundedness are undecidable[9]. Some programs will yield to analysis, but there will always be some graphs that cannot be analyzed in finite time. When static scheduling fails, it becomes necessary to resort to dynamic scheduling of the program. A dynamic scheduler should satisfy two requirements:

Requirement 1 (Complete Execution) *The scheduler should implement a* complete execution *of the of the Kahn process network program. In particular, if the program is non-terminating, then it should be executed forever without terminating.*

Requirement 2 (Bounded Execution) *The scheduler should, if possible, execute the Kahn process network program so that only a bounded number of tokens ever accumulate on any of the communication channels.*

When these requirements conflict, such as for unbounded programs which require unbounded buffering of tokens for a complete execution, then requirement 1 takes precedence over requirement 2. We prefer a complete, unbounded execution to a partial, bounded one.

Dynamic scheduling policies can generally be classified as data driven, demand driven, or some combination of the two. Some work has been done to relate data driven and demand driven scheduling policies. Pingali and Arvind [37, 39, 38] take the approach of transforming program graphs so that a data driven execution of the transformed program is equivalent to a demand driven execution of the original program. Kahn

and MacQueen advocate a demand driven policy [25]. Jagannathan and Ashcroft [22, 23, 2, 3, 21] present an execution policy that they call "eazyflow." It combines aspects of eager (data driven) execution with aspects of lazy (demand driven) execution. We will discuss these policies and some of their shortcomings.

3.1 Decidablilty

Buck showed that Boolean dataflow graphs have computational capability equivalent to a universal Turing machine. Using just the switch and select actors together with actors for performing addition, subtraction, and comparison of integers, it is possible to construct a universal Turing machine [9].

3.1.1 Termination

Theorem 1 *The problem of deciding whether a BDF graph will terminate is undecidable.*

Because a Turing machine can be constructed as a BDF graph, solving the termination decision problem for BDF graphs would allow us to solve the halting problem. But because the halting problem is known to be undecidable, the termination problem for BDF graphs must also be undecidable. Because BDF graphs are a special case of Kahn process networks, we have the following corollary.

Corollary 1 *The problem of deciding whether a Kahn process network will terminate is undecidable.*

3.1.2 Boundedness

Theorem 2 ([9]) *The problem of deciding whether a BDF graph can be scheduled with bounded memory is undecidable.*

Buck proved this by showing that solving the bounded memory decision problem would allow us to solve the halting problem, which is known to be undecidable. Again, because BDF graphs are a special case of Kahn process networks, we have the following corollary.

Corollary 2 *The problem of deciding whether a Kahn process network can be scheduled with bounded memory is undecidable.*

3.1.3 Implications

How does decidability impact our goal of executing arbitrary process networks forever in bounded memory whenever possible? Because termination and boundedness are both undecidable problems, we might wonder if it is even possible to achieve this goal. When a question is undecidable, we cannot devise an algorithm that will always arrive at an answer in finite time [17]. But because we want an infinite execution of a program, we do not need to arrive at an answer in finite time. Our scheduling algorithm can operate as the program executes and need never terminate precisely because we do not want the program to terminate. Because

termination is undecidable, we cannot always determine ahead of time whether or not a particular program terminates. We cannot always identify bounded programs for the same reason. However, we will see that it is simple to devise a scheduling algorithm that will execute process networks forever whenever possible. We can also devise methods of scheduling programs with bounded memory. Satisfying both requirements simultaneously is more difficult, but still possible, as we will show.

3.2 Data Driven Scheduling

Data driven execution, where a process is activated as soon as sufficient data is available, satisfies requirement 1. This policy always results in a complete execution of the program because execution stops if and only if all of the processes are blocked reading from empty communication channels. For strictly bounded programs, where all executions lead to bounded buffer sizes for the communication channels, both requirements 1 and 2 are satisfied: complete, bounded execution is guaranteed. But because the question of boundedness is undecidable, we cannot always classify programs as strictly bounded, bounded or unbounded. For bounded programs, only some of the execution orders lead to bounded buffer sizes. Unfortunately, data driven schedulers do not always find such execution orders when they exist. This can lead to unbounded accumulation of tokens on the communication channels.

Here is an example of a data driven scheduling policy. Let a Kahn process network be described by a connected graph $G = (V, E)$ where V is the set of vertices corresponding to processes, and E is the set of directed edges corresponding to communication channels. Find the set $V_E \subseteq V$ of all *enabled* processes in the program graph G. By definition, a process can be in one of two states: *blocked* attempting to get data from an empty channel, or *enabled*. If V_E is not empty, $V_E \neq \emptyset$, activate all of the processes in V_E. If all of these processes become blocked, repeat by finding a new set V_E' of enabled processes.

It is clear that this policy satisfies requirement 1 and executes a program indefinitely whenever possible. For a non-terminating program, there is always at least one enabled process, so V_E is never empty and execution never terminates. Execution terminates only when $V_E = \emptyset$, which indicates that all processes are blocked. Thus execution under this policy terminates if and only if the program in question terminates. However, it is not necessary to decide whether or not the program will terminate, so the fact that the termination decision problem is undecidable does not affect us.

The process network described in figure 1.4 on page 4 can be executed forever in bounded memory with data driven scheduling, as discussed in section 1.3.3. This is the behavior we desire for a non-terminating, strictly bounded program. However, we cannot classify every program as terminating or non-terminating, bounded or unbounded before beginning execution. How does data driven scheduling behave for programs that are not strictly bounded?

For unbounded programs, where all complete executions lead to unbounded accumulation of tokens, the only way to satisfy requirement 2 is to violate requirement 1 and implement only a partial execution of the program, stopping before too many tokens accumulate on the channels. Because data driven schedul-

```
(int stream V, int stream W) = process d(int stream U)
{
    do
    {
        int u = get(U);
        put(u,V);
        put(u,W);
    } forever;
}
```

Figure 3.1: A process that duplicates a stream.

```
process p(int stream U)
{
    do
    {
        print(get(U));
    } forever;
}
```

Figure 3.2: A process that prints the elements of a stream.

ing always results in a complete execution, it also results in unbounded token accumulation for unbounded programs. Consider again the dataflow process network in figure 2.7 on page 25. This unbounded program executes indefinitely with data driven scheduling, but an unbounded number of tokens accumulate.

For bounded programs, where some (but not all) complete executions lead to unbounded buffer sizes for the communication channels, data driven scheduling can lead to *unnecessary* accumulation of tokens. The dataflow process network in figure 2.3 on page 22 is not strictly bounded, but can be executed forever in bounded memory. In our previous discussion in section 2.3, we considered the firing sequences 1,1,2,1,2 and 1,1,1,2,2. Both of these sequences can be repeated indefinitely with the dataflow process network executing forever in bounded memory. However, the firing sequence 1,1,1, ... is also a possible execution order for this dataflow process network. Because process 1 has no inputs, it never blocks getting data from an empty channel and so it is always enabled. Once activated, this process never suspends. Without parallel execution, there is no opportunity to activate process 2, so every token produced by process 1 accumulates on the channel. Even with the possibility of parallel execution, there is nothing to prevent process 1 from producing data faster than it can be consumed by process 2.

The process network in figure 3.5 is another example of a bounded program where data driven scheduling can lead to unnecessary accumulation of tokens. Figures 3.3, 1.3, 3.1, 3.4 and 3.2 give the definitions of the processes used in this example. In simple data driven execution, all enabled processes are executed until they block. This results in tokens being produced on streams X and Y at the same rate. But because process m does not consume data from each of its inputs at the same rate, tokens will accumulate on one or

```
int stream V = process a(int stream U, int x)
{
    do
    {
        put(get(U) + x, V);
    } forever;
}
```

Figure 3.3: A process that adds a constant to each element of a stream.

```
int stream W = process m(int stream U, int stream V)
{
    int u = get(U);
    int v = get(V);
    do
    {
        if (u < v)
        {
            put(u, W);
            u = get(U);
        }
        else if (u > v)
        {
            put(v, W);
            v = get(V);
        }
        else /* u == v, discard duplicate */
        {
            put(u, W);
            u = get(U);
            v = get(V);
        }
    } forever;
}
```

Figure 3.4: A process that implements an ordered merge. Given two monotonically increasing sequences as input, the output is also monotonically increasing. Values duplicated on the two inputs are removed.

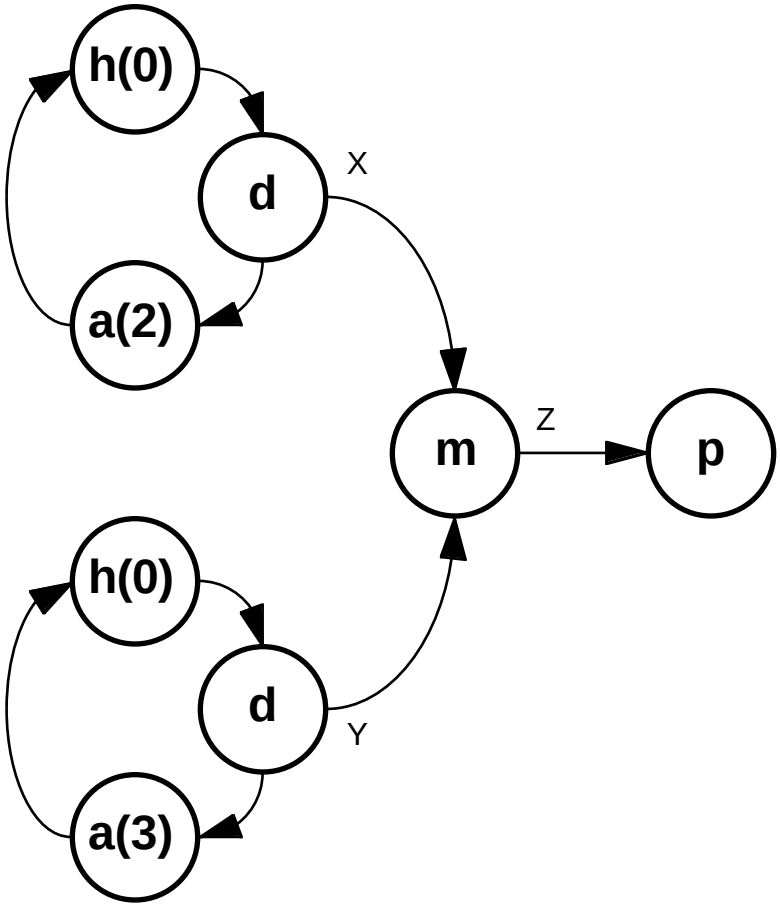

Figure 3.5: A process network that merges two streams of monotonically increasing integers (multiples of 2 and 3) to produce a stream of monotonically increasing integers with no duplicates. The processes are defined in figures 1.3, 3.1, 3.2, 3.3 and 3.4.

the other of the inputs. In this case tokens accumulate without bound on the channel for stream *Y*.

3.3 Demand Driven Scheduling

For bounded execution, we must ensure that data already produced is consumed before even more data is produced. We need a means of regulating data sources so that they produce data at the same rate that it is consumed. Suspending processes only when they attempt to consume data from empty channels is not enough. We must also consider suspending processes when they produce data so that other processes have an opportunity to consume that data. This is the approach taken by Kahn and MacQueen for demand driven execution[25].

Demand driven scheduling policies avoid the unnecessary production of tokens by deferring the activation of a process until its output is needed as input for another process. In this way, we produce only the data needed for the final result. However, we will see examples where unbounded token accumulation is still possible with demand driven scheduling.

3.3.1 Anticipation Coefficients

Kahn and MacQueen describe demand driven execution of process networks[25]. A single process is selected to drive the whole network. This driving process is specified in the program and is usually the process that produces the ultimate output for the program. When a process attempts to consume data from an empty input channel it is suspended, the channel is marked as *hungry*, and the producer process for that channel is activated. When this new process is activated, it may attempt to consume from an empty input which would cause yet another process to be activated in turn. When a process produces data on a hungry channel, it is suspended and the waiting consumer process is activated. Note that there is no transfer of control when consuming from a channel that is not empty or when producing to a channel that is not hungry.

The example in figure 3.5 cannot be executed in bounded memory with data driven scheduling. However, this program can be executed in bounded memory with demand driven scheduling. The subgraphs that act as sources are regulated so that they produce data only when needed. As soon as a process produces data on a hungry channel, it is suspended so that the destination process has an opportunity to consume that data.

In this demand driven scheme, there is never more than one active process at any time. Instead of suspending a process as soon as it produces data on a hungry channel, it could be allowed to continue to run in parallel with the waiting consumer process in *anticipation* of demands for its output. However, there is the danger that the producer process may generate results that are never consumed, just as with data driven execution. Kahn and MacQueen solve this problem by assigning a non-negative integer *A*, called the *anticipation coefficient*, to each channel [25]. Once activated, a producer is not deactivated until there are *A* unconsumed tokens on its output. Kahn and MacQueen suggest that the value of *A* should be set when the channel is passed

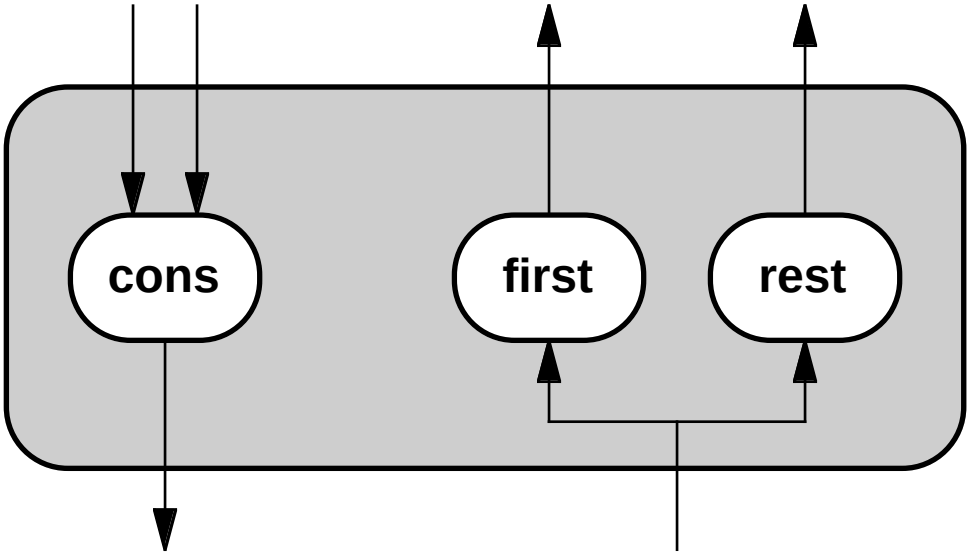

Figure 3.6: The cons operator together with corresponding demand propagation code.

as an input parameter to a new process. As we will show later, it may be necessary to change this value dynamically while the process network is executing in order to avoid causing an artificial deadlock.

3.3.2 Demand Propagation

Pingali and Arvind present a different approach to demand driven execution [37, 39, 38]. They give a method to transform a graph so that a data driven execution of the new graph is equivalent to a demand driven execution of the original graph. For each arc in the original graph, they add a new arc to carry demand tokens in the opposite direction. For every node in the original graph, they add new nodes to propagate demands along the new arcs.

There are only a few operators in the language used by Pingali and Arvind: cons, first, rest, fork, simple functions that consume and produce single tokens, select, and the equivalent of switch. The cons, first, and rest operators are the same ones described earlier in section 1.2.2. The fork operator is equivalent to process *d* in figure 3.1 on page 34. The switch and select operators were discussed in section 2.1.1 and section 2.4. Each operator in the language is transformed into a small program (in a slightly more general language) that includes demand propagation code.

For example, figure 3.6 shows how the cons operator is transformed. The cons operator consumes only one token from its first input, and then consumes tokens from the other input. This is equivalent to a select operator where the control stream is a single “FALSE” token, followed by all “TRUE” tokens. Any tokens after the first will never be consumed from the first input. If data driven execution were used, tokens could be produced despite the fact that they will never be consumed. Thus we need to send a signal to the

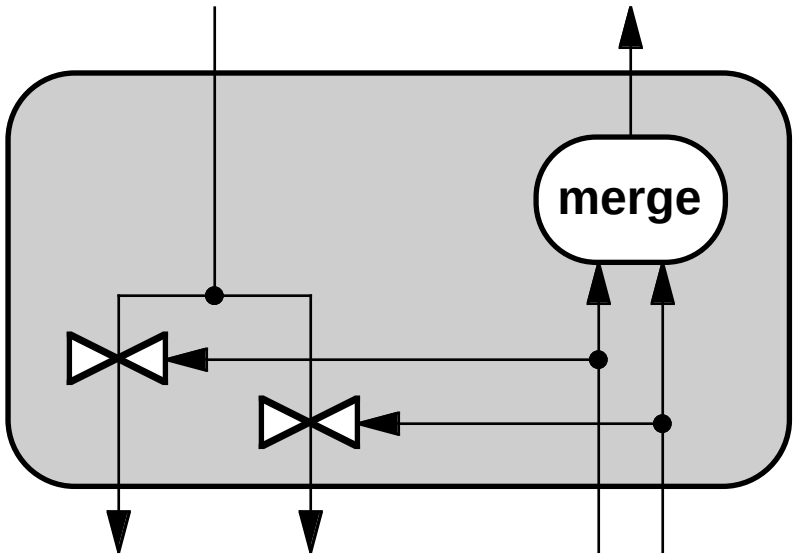

Figure 3.7: The fork operator together with corresponding demand propagation code.

source indicating when tokens are needed. For each input of the cons operator there is a corresponding output that produces these demands. For the output of the cons operator, there is a corresponding input that accepts demands. The first demand token is sent to first demand output. This allows the source to produce the token consumed on the first input of the cons operator. Remaining demand tokens are sent to the other demand output. Thus, no more tokens will be requested for the first input. All remaining requests are for the other input.

Data driven execution of this small program is equivalent to demand driven execution of the cons operator. Demand tokens propagate in a data driven manner, being sent along the appropriate path as soon as they arrive. Once the demands have propagated to the source, tokens will propagate forward in a data driven manner. The cons operator can execute as soon as these tokens arrive.

Figure 3.7 shows how the fork operator is transformed. The fork operator consumes a token from its input and copies it to each output branch. If data driven execution were used, tokens could be produced before they have been demanded. The gate operators on each output of the fork prevent tokens from flowing through until demand tokens have arrived. As demands arrive, they enable the corresponding gate and then are merged into a single output demand stream. Demand tokens can arrive on either input in any order and must be passed on as soon as they arrive. Thus we must use the non-determinate merge operator described earlier in section 2.1.1. Fortunately this use of the merge operator does not introduce nondeterminism. Any reordering that may occur on the merge output stream will not be apparent because the demand tokens do not have values and they are indistinguishable from one another.

Figures 3.8 and 3.9 show how a small program is transformed. Demands for the program output

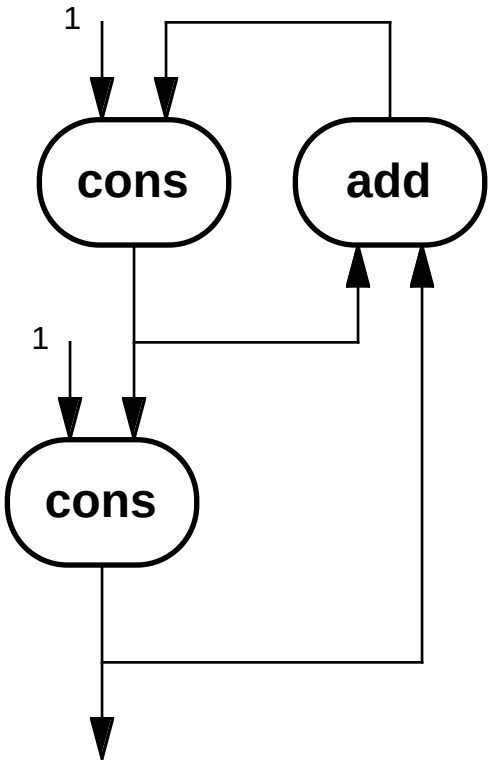

Figure 3.8: A process network program that computes Fibonacci numbers.

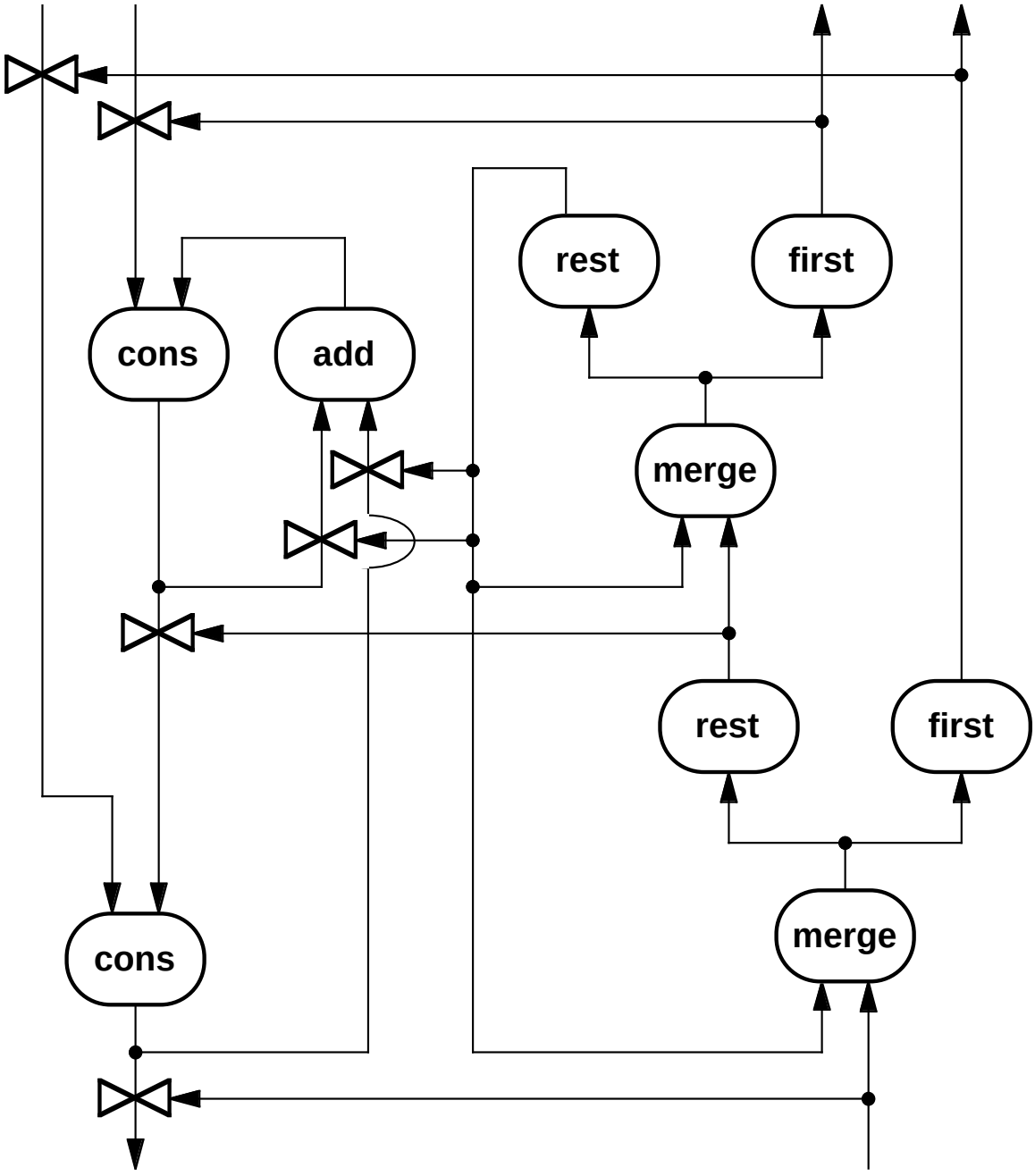

Figure 3.9: The Fibonacci program with demand-propagation code added.

```
(int stream V, int stream W) = process x(int stream U, int y)
{
    do
    {
        int u = get(U);
        if (u mod y == 0) put(u, V);
        else put(u, W);
    } forever;
}
```

Figure 3.10: A process that separates a stream in to those values that are and are not evenly divisible by a constant.

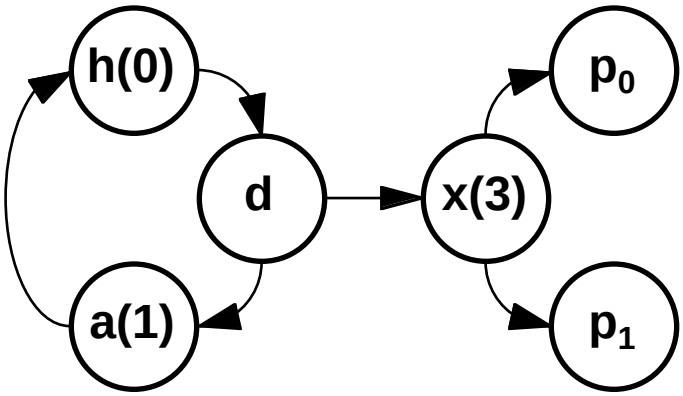

Figure 3.11: A process network that requires unbounded buffering with demand driven execution. The processes are defined in figures 1.3, 3.3, 3.1, 3.2 and 3.10.

flow through the graph to the gate operators at the program inputs where they allow the necessary data to flow forward through the program graph and produce the results that were demanded. One drawback of this approach is the significant overhead required to propagate demands.

3.3.3 Unbounded Execution

Pingali and Arvind prove that a data driven execution of a transformed program is equivalent to demand driven execution and will produce exactly the values required to compute the final output and no more. But this does not preclude unbounded buffering of tokens. For example, if an unequal number of demands arrive for the branches of a fork, then tokens will accumulate at the input of one of the gate operators.

Just as the presence of multiple data sources is a problem in data driven scheduling, the presence of multiple data sinks is a problem in demand driven scheduling. The program shown in figure 3.11 requires unbounded buffering with simple demand driven scheduling. If processes p_0 and p_1 generate demands at the same rate, then tokens will accumulate on one or the other of the outputs of process x because it does not produce tokens on each of its outputs at the same rate. In this case tokens accumulate without bound on the

input to process p_1. If we could switch from demand driven scheduling to data driven scheduling at the point where process x produces undemanded data, then process p_1 would be able to consume that data.

3.4 Combined Data/Demand Driven Scheduling

We have seen examples where data driven scheduling or demand driven scheduling can lead to an unbounded accumulation of tokens. Eazyflow is a hybrid dataflow model that combines aspects of data driven (eager) and demand driven (lazy) scheduling [22, 23, 2, 3, 21]. In eazyflow, execution alternates between demand driven and data driven. Data driven execution is begun when there is a token deficit and continues until there is a token surplus, at which point demand driven execution resumes. Combined approaches, like eazyflow, are more promising than data driven or demand driven scheduling alone, but still fall short of our goal.

In eazyflow, streams are classified as eager or lazy, which determines whether data driven or demand driven execution is used to produce stream values. Streams defined with functions that consume unpredictable amounts of data (such as process *m* in figure 3.4) or produce unpredictable amounts of data (such as process *x* in figure 3.10) are classified as lazy. Streams defined with functions that consume and produce predictable amounts of data (such as synchronous dataflow processes) are classified as eager, unless one or more of the input streams to the function are lazy. If some of the input streams are lazy, then the output stream is also lazy.

For example, stream *Z* in figure 3.5 on page 36 is lazy because process *m* consumes an unpredictable (data dependent) number of tokens from each of its inputs. The streams *X* and *Y* are eager. If there is a token deficit and too few tokens are available when process *m* demands the next value of stream *X*, then data driven execution of the subgraph that produces *X* is triggered. Execution of the subgraph is suspended once enough tokens have been produced to overcome the deficit and to create a surplus.

Fixed thresholds that are parameters of the system define *deficit* and *surplus* [23]. The surplus threshold serves the same purpose as Kahn and MacQueen's anticipation coefficient [24]. As we will show, a fixed threshold is inadequate because it may be necessary to adjust the threshold dynamically in order to avoid causing an artificial deadlock.

Chapter 4

Bounded Scheduling of Process Networks

We now present a scheduling policy that simultaneously satisfies requirement 1 (compete execution) and requirement 2 (bounded execution): arbitrary Kahn process networks execute forever with bounded buffering when possible. We give priority to requirement 1 and prefer a complete, unbounded execution to a partial, bounded execution. As we showed earlier in section 3.2, data driven scheduling satisfies requirement 1 and always yields a complete execution of a Kahn process network. In particular, non-terminating programs execute forever. For a strictly bounded program, which has bounded buffering for *any* execution, any scheduling policy satisfies requirement 2. Thus data driven scheduling satisfies both requirements 1 and 2 when applied to strictly bounded programs.

Not every program is strictly bounded. Some programs are bounded — they *can* be executed with bounded buffer sizes for each of the communication channels, but *some* execution orders lead to unbounded buffer sizes. Other programs are unbounded — *all* complete executions lead to unbounded buffer sizes. We present a scheduling policy that always executes non-terminating, bounded programs forever with bounded buffer sizes. If the program is unbounded, then our policy will still execute it forever (we do not introduce deadlock) but in this case it is not possible to bound the buffer sizes. And of course execution under our policy will terminate given a terminating program.

4.1 Program Graph Transformation

We transform a program graph G to produce a semantically equivalent graph G^0 that is strictly bounded by b^0. This transformation may introduce *artificial deadlock* so that a complete execution of the transformed graph G^0 represents only a partial execution of the original graph G. We execute the transformed program G^0 with data driven scheduling, or any other policy that satisfies requirement 1, until execution stops. If execution of G^0 never stops, then we have succeeded in implementing a complete, bounded execution. If

execution of G^0 stops and we discover that this complete execution of G^0 represents a complete execution of the original program G, then we have also succeeded in implementing a complete, bounded execution. However, if execution of G^0 stops and we discover that this complete execution represents only a partial execution of the original program G, then we have chosen a bound b^0 that is too small. One or more of the channels must buffer more than b^0 tokens in order to implement a complete execution of the program G. Thus we must choose a new larger bound $b^1 > b^0$ and try again.

By definition, a Kahn process network that is bounded by b has at least one complete execution such that every channel is bounded by b. Even if we do not know its value, this bound b exists and is finite. Thus, as we choose successively larger bounds $b^0 < b^1 < b^2 < \cdots$, we will eventually discover a bound b^N that is greater than or equal to b. If the program graph G is bounded by b and we apply our transformation to produce a graph G^N that it is strictly bounded by b^N with $b^N \geq b$, then a complete execution of G^N corresponds to a complete execution of G. Thus we can achieve our goal of complete, bounded execution for any bounded Kahn process network.

We begin with a process network described by a connected graph $G = (V, E)$ with a set of vertices V corresponding to the processes and a set of directed edges E corresponding to the communication channels. For each edge $e_i = (v_m, v_n)$, add a new edge $e_i' = (v_n, v_m)$ in the reverse direction. We call the channels corresponding to these new edges *feedback channels* because they introduce directed cycles, or feedback loops, in the program graph. Let $|e_i|$ be the size of an edge e_i, the number of tokens stored in the buffer for the communication channel. Place $b_i - |e_i|$ tokens on the edge e_i' so that the total number of tokens for the pair of edges is $b_i = |e_i| + |e_i'|$. Frequently there are no tokens initially buffered on the communication channels, with $|e_i| = 0$ and $|e_i'| = b_i$.

Modify each process so that it must consume one token from a feedback channel e_i' for each token that it produces on the corresponding data channel e_i. Also, a process must produce one token on a feedback channel e_i' for each token that it consumes from the corresponding data channel e_i. This requires modified semantics for the get and put operations, with no other modifications to the processes. Thus the number of tokens on the pair of edges remains constant, $b_i = |e_i| + |e_i'|$. In particular, the number of tokens on the data channel is strictly bounded $|e_i| \leq b_i$. The program as a whole is strictly bounded by $b = \max b_i$, the maximum of the bounds for all channels.

As with data channels, processes block when attempting to get tokens from empty feedback channels. The tokens flowing along these feedback channels need not have values. They simply restrict the order of execution and do not affect the values of the tokens on the data channels. This transformation preserves the process network model: a process blocks only when reading from an empty channel and channels are potentially unbounded in size. Instead of adding feedback channels, we could modify the process network model by directly limiting the capacity of the channels and requiring that processes block when writing to a full channel. For the rest of our discussion, we will dispense with the notion of blocking reads from empty feedback channels in favor of blocking writes to full channels.

The feedback channels are similar to the demand arcs in Pingali and Arvind's graph transformation [37, 39, 38] or the acknowledgment arcs of Dennis' static dataflow model [13, 14, 1, 18]. Directly bound-

ing the capacity of a channel is similar to Kahn and MacQueen's anticipation coefficients [25].

Limiting the capacity of the channels (either directly or indirectly with feedback channels) places additional restrictions on the order in which get and put operations can take place. The transformed graph computes the same result as the original graph with the possible exception that we may have introduced deadlock. A complete execution of the transformed graph may be only a partial execution of the original graph.

This graph transformation is one way to divide the set of all possible executions into a hierarchy of subsets. Let O be the set of execution orders for the original program graph, and let O^i be the set of execution orders for the transformed graph that it is strictly bounded by b^i. The set of execution orders for the transformed program is a subset of the execution orders for the original graph, $O^i \subseteq O$. Also, if we transform the graph for two different values b^i and b^j, then $O^i \subseteq O^j$ if $b^i \leq b^j$. If we choose a bound b^i such that there are no infinite executions in O^i even though there are infinite executions in O, then we have introduced artificial deadlock.

4.2 Bounded Scheduling

If a program is bounded, then there exists a finite least upper bound b and an execution order such that the size of each buffer never exceeds b. We choose an initial estimate b^0 of b and transform the program so that it is strictly bounded by b^0. If we happen to choose $b^0 \geq b$, then a complete execution of the transformed program corresponds to a complete execution of the original program. Execution of the transformed program terminates if and only if execution of the original program would also terminate. We call this situation, where all processes are blocked reading from empty channels, *true deadlock*. If we choose $b^0 \leq b$, then execution could also stop if one or more processes are blocked writing to full channels. We call this situation *artificial deadlock*.

We choose an initial bound b^0 and transform the original graph so that it is strictly bounded by limiting the capacity of each communication channel to b^0. We then execute this transformed program graph with an execution policy, such as data driven execution, that satisfies requirement 1. If execution stops due to artificial deadlock, with one or more processes blocked writing to full channels, then we increase the capacities of all channels in the network to $b^1 > b^0$ so that now the program is strictly bounded by b^1. After increasing the channel capacities, we continue execution from the point where we left off. Each time that execution stops due to artificial deadlock, we increase the capacities of all the channels and continue.

If the program is bounded with a finite least upper bound b, then eventually our estimate will increase to meet or exceed that bound $b^N \geq b$, and we will be able to execute the program forever with bounded buffering, simultaneously satisfying both requirements 1 and 2. If the program is unbounded, execution repeatedly stops due to artificial deadlock, and we increase the channel capacities repeatedly and without limit. There is no bound on the buffer requirements for the communication channels, but the execution will continue indefinitely (or until system resources are exhausted). So we see that requirement 1 is given priority over requirement 2: we continue to execute unbounded programs as long as possible, preferring a complete,

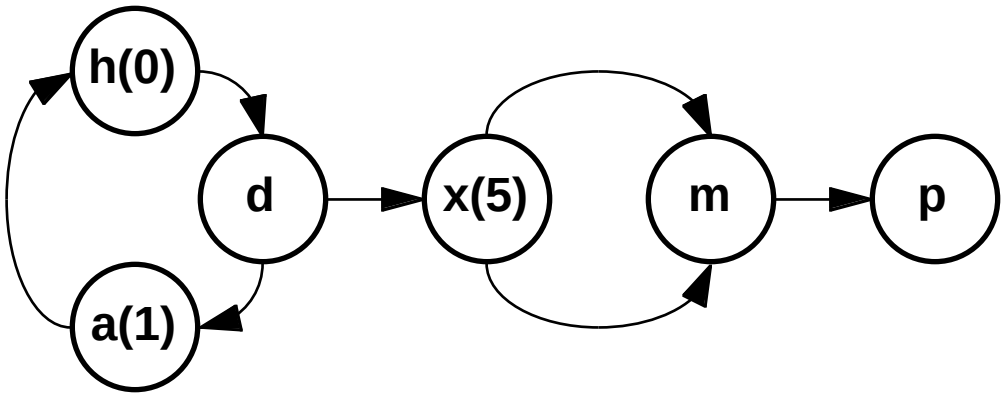

Figure 4.1: A bounded process network in which different channels can have different bounds. The processes are defined in figures 3.3, 1.3, 3.1, 3.10, 3.4 and 3.2.

unbounded execution to a partial, bounded execution. If the original program would terminate, then execution of the transformed program may stop several times due to artificial deadlock, but eventually execution will stop due to a true deadlock where all processes are blocked reading from empty channels and no processes are blocked writing to full channels.

So we see that this bounded scheduling policy has the desired behavior for terminating and non-terminating programs, strictly bounded, bounded and unbounded programs. This is important because termination and boundedness are undecidable. There will always be programs that we cannot classify, so our scheduling policy must have a reasonable behavior for all types of programs.

Part of the elegance of this approach is that *any* scheduling policy that satisfies requirement 1 can be used for the transformed graph: data driven, demand driven or some combination of the two. We have modified the graph is such a way that any scheduler works — any execution leads to bounded buffering on the communication channels.

4.3 Reducing Buffer Bounds

So far we have set all the channel capacities to the same value. Data driven execution could make use of all available capacity, requiring large amounts of storage for unconsumed tokens. Some form of demand driven or hybrid execution policy could be employed to avoid using all available capacity. We can also set different capacity limits for different channels. However, we must be careful when doing this so that we preserve bounded buffer sizes.

Consider the process network in figure 4.1. An increasing sequence of integers is split into two streams: a stream of values that are evenly divisible by 5, and a stream of values that are not evenly divisible by 5. These streams are then merged to again form a stream of increasing integers. Initially, values 0 and 1 are consumed by process *m*. The value 0 is sent to the output, then process *m* waits for the next multiple of 5 to be available. In the meantime, the values 2, 3 and 4 queue up on the other channel. Once the value 5 becomes

available and is consumed by process *m*, the values 2, 3 and 4 are copied from the input to the output. We see that the channel for the stream of values that are not multiples of 5 must have a capacity of at least 3. All other streams can have a capacity as small as 1.

If we begin by setting the initial capacities of all channels to 1, and increase *all* channel capacities by 1 each time the program deadlocks, then every channel ends up with a capacity of 3. There are 7 channels in this example, so the total capacity of the system is 21. This is higher than the minimum of 9 if we allow different channels to have different capacities. If we generalize this example so that the divisor in process *x* is N instead of 5, then we see that storage requirements are $7N$ if we make the capacity limits the same for each channel. The minimum storage required in this example is $N+4$. If N is a large number, then the difference between $7N$ and $N+4$ can be quite large.

It is not strictly necessary to increase the capacity of every buffer. When execution stops due to artificial deadlock, one or more processes are blocked writing to full channels. Increasing the capacity limits of channels that are not full does not allow execution to continue. It is necessary to increase the capacity of one or more of the full channels. It is important not to increase the largest buffer (unless all full buffers are the same size) because this could lead to unbounded growth of that buffer. We will show that it is sufficient to increase the full channel with the smallest capacity.

Instead of increasing the capacity of all the channels, we can increase the capacity of only the full channel with the smallest capacity. One possible sequence of deadlocks that occur with this policy and data driven execution is shown in figure 4.2. Initially all channel capacities are 1 in figure 4.2(a). At several points there is a tie that must be broken, as in figure 4.2(c). Our arbitrary choices led to the distribution of channel capacities shown in figure 4.2(e).

Increasing the smallest full channel guarantees that every full channel will eventually be increased if necessary to unlock the program. If the same channel is increased repeatedly, then eventually it will no longer be smallest. If some full channel other than the smallest is increased, then some buffers could grow without bound. Consider what would happen if only the largest full channel were increased, for example. Choosing the smallest channel prevents this from happening. The advantage of this policy is that some channels have smaller capacities than if all channel capacities were increased.

4.4 Dataflow Scheduling

These results for bounded scheduling of process networks can be applied to dataflow. But because the firing of a dataflow actor is atomic, we cannot directly use the technique of limiting the capacities of channels. When an actor fires, it consumes input tokens and produces output tokens. Once initiated, the firing cannot be suspended. In particular, it cannot be suspended when it produces output tokens. In general we do not know how many tokens a firing will produce, if any. Thus we cannot determine if firing an actor will produce enough tokens to exceed the capacity limit of a channel.

Instead we classify dataflow actors as *deferrable* or *non-deferrable*. We define an actor to be *de-*

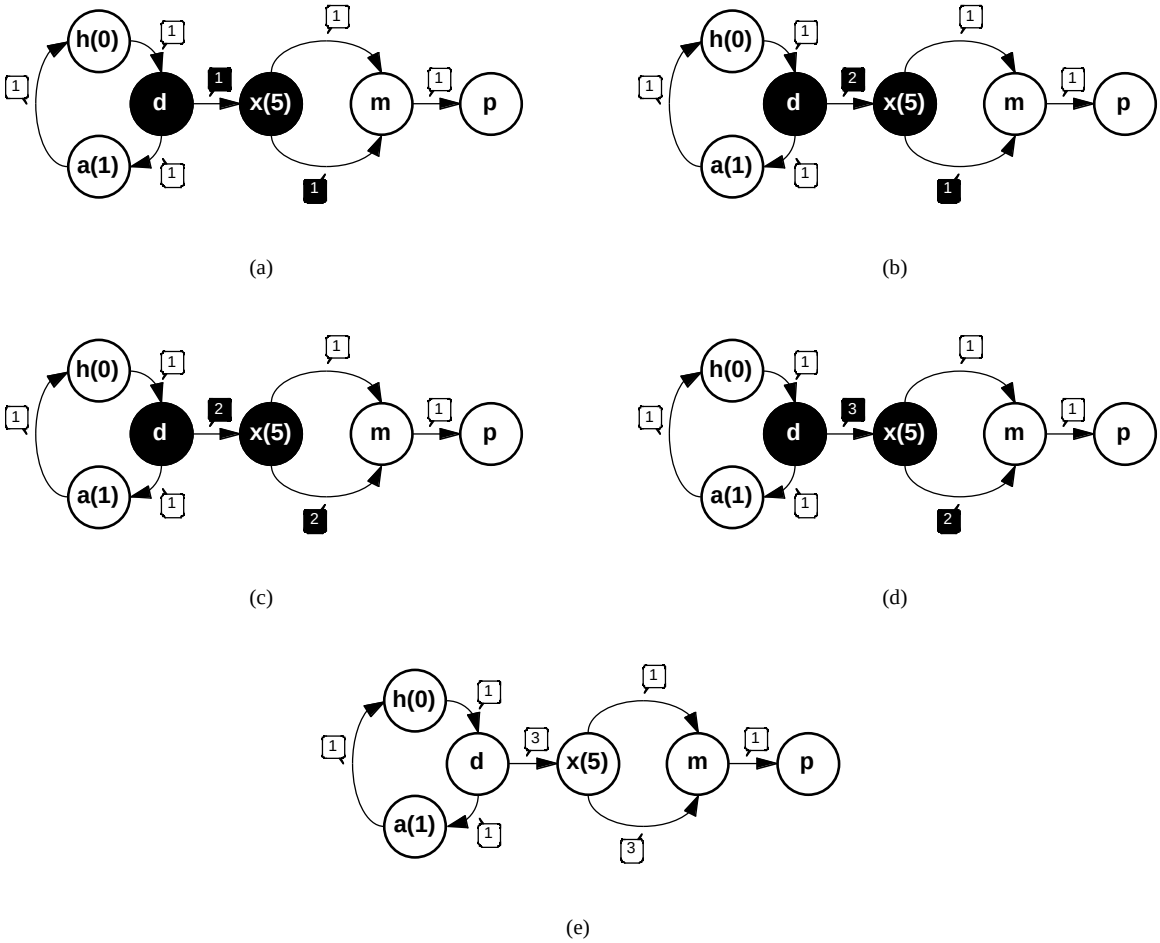

Figure 4.2: The sequence of deadlocks that occur when only the smallest full buffer is increased. At each step, the indicated processes are blocked writing to the indicated full channels. The capacities of the channels are indicated at each step.

ferrable when it is enabled (enough tokens are available at the inputs to satisfy its firing rules), but one or more of its output channels has sufficient tokens to satisfy the demand of the destination actor. In order to preserve bounded buffering when possible, a dataflow actor should not add tokens to an edge when there are already enough tokens to satisfy the demand of the destination actor on that edge (enough tokens are available to match the applicable firing rule pattern for the destination).

If a dataflow program is bounded, then there exists a finite bound b such that no actor consumes or produces more than b tokens in a firing. There could be as many as $b-1$ tokens on a channel with the demand still unsatisfied. An actor could produce as many as b tokens when fired, so there could be as many as $2b-1$ tokens on a channel. If deferrable actors are never fired, then there will never be more than $2b-1$ tokens on any channel.

For a dataflow program described by a graph $G = (V, E)$, data driven execution would find the set V_E of enabled actors and fire all of them. Instead, we find the subset $V_D \subseteq V_E$ of deferrable actors. Only the non-deferrable actors in $V_E - V_D$ are fired. Then the new set V_E' of enabled actors and the new set of deferrable actors are computed, and the actors in $V_E' - V_D'$ are fired. This repeats as long as $V_E' - V_D'$ is not empty. If execution stops, then $V_E' - V_D'$ must be empty. All enabled actors are deferrable, so $V_E' = V_D'$.

At this point, we must fire a deferrable actor to satisfy requirement 1. For each deferrable actor v_j, let c_j be the maximum of the buffer sizes for that actor's output channels that have satisfied demands. If we choose to fire the actor with the minimum value for c_j, then we can also satisfy requirement 2.

The motivation for this is similar to our motivation for increasing the capacity of the smallest full channel. Each deferrable actor has at least one output channel with a satisfied demand. In some sense, such channels are full. In general we do not know which channel(s) an actor firing will produce tokens on, so we must consider the largest such channel buffer for each actor. This determines the value c_j for each deferrable actor, and we choose to fire the actor with the smallest value for c_j. If this actor firing could produce as many as b tokens on a channel that already has a buffer size of c_j. Thus the maximum buffer size for the entire graph would be $\max(\max(c_j), min(c_j) + b)$. If this actor firing produces no tokens, or produces tokens on a channel other than the one that determined c_j, then the maximum buffer size could be smaller.

If the same actor is fired repeatedly, then eventually it will no longer have the smallest value for c_j, or it will no longer be enabled. If some other actor were fired, then some buffers could grow without bound. Consider what would happen if only the actor with the largest value for c_j were fired. Choosing the actor with the smallest value of c_j prevents this from happening.

Chapter 5

Implementation in Ptolemy

Ptolemy [10] is an object oriented simulation and prototyping environment. A basic abstraction in Ptolemy is the domain, which realizes a computational model. Examples of domains include synchronous dataflow (SDF), Boolean dataflow (BDF), dynamic dataflow (DDF) and discrete-event (DE). Subsystems can be described with an appropriate domain and domains can be mixed to realize an overall system simulation. In addition to mixed-domain simulation, Ptolemy also supports code generation for the dataflow domains [40]. The theory presented in this thesis has been implemented in Ptolemy as the Process Network (PN) domain. The PN domain includes all the dataflow domains (SDF, BDF and DDF) as subdomains. This hierarchical relationship among the domains is shown in figure 5.1. The model of computation for each domain is a strict subset of the model for the domain that contains it.

The nodes of a program graph, which correspond to processes or dataflow actors, are implemented in Ptolemy by objects derived from the class `Star`. The firing function of a dataflow actor is implemented by the `run` method of `Star`. The edges of the program graph, which correspond to communication channels, are implemented by the class `Geodesic`. A `Geodesic` is a first-in first-out (FIFO) queue that is accessed by the `put` and `get` methods. The connections between stars and geodesics are implemented by the class `PortHole`. Each `PortHole` has an internal buffer. The methods `sendData` and `receiveData` transfer data between this buffer and a `Geodesic` using the `put` and `get` methods.

Several existing domains in Ptolemy, such as Synchronous Dataflow (SDF) and Boolean Dataflow (BDF), implement dataflow process networks by scheduling the firings of dataflow actors. The firing of a dataflow actor is implemented as a function call to the `run` method of a `Star` object. A scheduler executes the system as a sequence of function calls. Thus, the repeated actor firings that make up a dataflow process are interleaved with the actor firings of other dataflow processes. Before invoking the `run` method of a `Star`, the scheduler must ensure that enough data is available to satisfy the actor's firing rules. This makes it necessary for a `Star` object to inform the scheduler of the number of tokens it requires from its inputs. With this information, a scheduler can can guarantee that an actor will not attempt to read from an empty channel.

By contrast, the PN domain creates a separate thread of execution for each node in the program

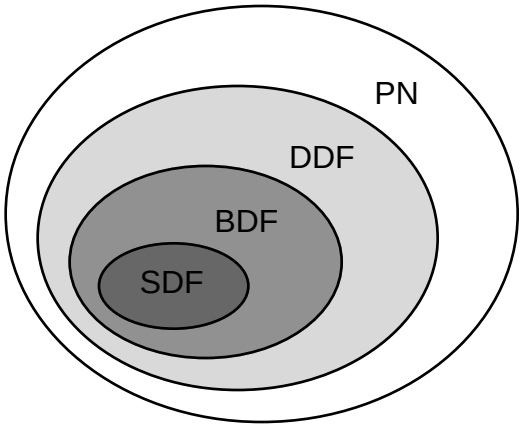

Figure 5.1: The hierarchy of dataflow domains in Ptolemy.

graph. Threads are sometimes called *lightweight processes*. Modern operating systems, such as Unix, support the simultaneous execution of multiple processes. There need not be any actual parallelism. The operating system can interleave the execution of the processes. Within a single process, there can be multiple lightweight processes or threads, so there are two levels of multi-threading. Threads share a single address space, that of the parent process, allowing them to communicate through simple variables. There is no need for more complex, heavyweight inter-process communication mechanisms such as pipes.

Synchronization mechanisms are available to ensure that threads have exclusive access to shared data and cannot interfere with one another to corrupt shared data structures. Monitors and condition variables are available to synchronize the execution of threads. A monitor is an object that can be locked and unlocked. Only one thread may hold the lock on a monitor. If a thread attempts to lock a monitor that is already locked by another thread, it is suspended until the monitor is unlocked. At that point it wakes up and tries again to lock the monitor. Condition variables allow threads to send signals to each other. Condition variables must be used in conjunction with a monitor; a thread must lock the associated monitor before using a condition variable.

The scheduler in the PN domain creates a thread for each node in the program graph. Each thread implements a dataflow process by repeatedly invoking the `run` method of a `Star` object. The scheduler itself does very little work, leaving the operating system to interleave the execution of threads. The `put` and `get` methods of the class `Geodesic` have been re-implemented using monitors and condition variables so that a thread attempting to read from an empty channel is automatically suspended, and threads automatically wake up when data becomes available.

The classes `PtThread`, `PtGate`, and `PtCondition` define the interfaces for threads, monitors, and condition variables in Ptolemy. Different implementations can be used as long as they conform to the interfaces defined in these base classes. At different points in the development of the PN domain, we experimented with implementations based on Sun's Lightweight Process library, AWESIME (A Widely Extensible Simulation Environment) by Dirk Grunwald [20], and Solaris threads[41, 15, 28, 30, 29, 42, 43]. The current

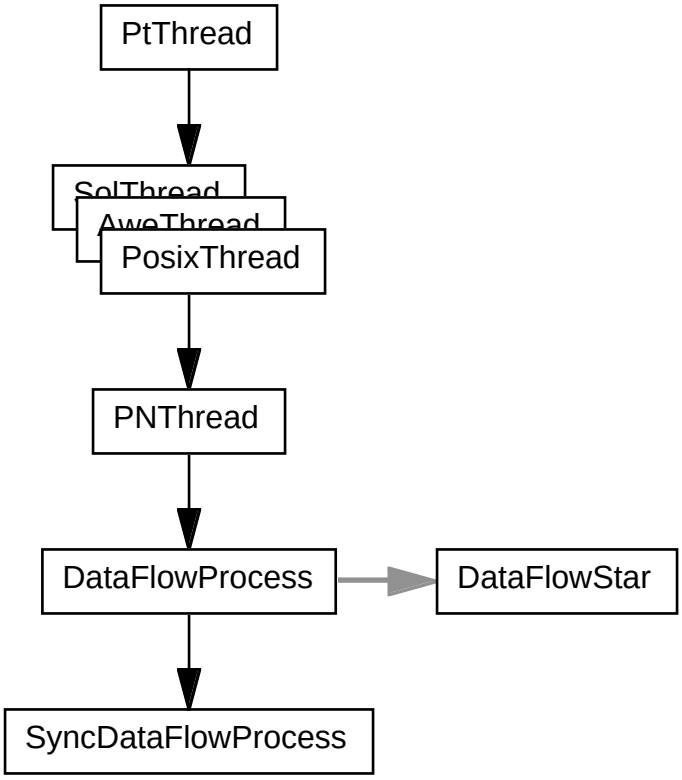

Figure 5.2: The class derivation hierarchy for threads. `PtThread` is an abstract base class with several possible implementations. Each `DataFlowProcess` refers to a `DataFlowStar`.

implementation is based on a POSIX thread library by Frank Müller [34, 35, 19, 36]. This library, which runs on several platforms, is based on Draft 6 of the standard. Parts of our implementation will need to be updated to be compliant with the final POSIX thread standard.

By choosing the POSIX standard, we improve the portability of our code. Sun and Hewlett Packard already include an implementation of POSIX threads in their operating systems, Solaris 2.5 and HPUX 10. Having threads built into the kernel of the operating system, as opposed to a user library implementation, offers the opportunity for automatic parallelization on multiprocessor workstations. Thus, the same program runs properly on uniprocessor workstations and multiprocessor workstations without needing to be recompiled. This is important because it would be impractical to maintain different binary executables of Ptolemy for each workstation configuration.

5.1 Processes

Figure 5.2 shows the class derivation hierarchy for the classes that implement the processes of Kahn process networks. The abstract base class `PtThread` defines the interface for threads in Ptolemy. The class `PosixThread` provides an implementation based on the POSIX thread standard. Other implementations

using AWESIME [20] or Solaris [41] are possible. The class `PNThread` is a `typedef` that determines which implementation is used in the PN domain. Changing the underlying implementation simply requires changing this `typedef`. The class `DataFlowProcess`, which is derived from `PNThread`, implements a dataflow process. The `Star` object associated with an instance of `DataFlowProcess` is activated repeatedly, just as a dataflow actor is fired repeatedly to form a process.

5.1.1 PtThread

`PtThread` is an abstract base class that defines the interface for all thread objects in Ptolemy. Because it has pure virtual methods, it is not possible to create an instance of `PtThread`. All of the methods are virtual so that objects can be referred to as a generic `PtThread`, but with the correct implementation-specific functionality.

The class `PtThread` has two public methods.

```
virtual void initialize() = 0;
```

This method initializes the thread and causes it to begin execution.

```
virtual void terminate() = 0;
```

This method causes execution of the thread to terminate.

The class `PtThread` has one protected method.

```
virtual void run() = 0;
```

This method defines the functionality of the thread. It is invoked when the thread begins execution.

5.1.2 PosixThread

The class `PosixThread` provides an implementation for the interface defined by `PtThread`. It does not implement the pure virtual method `run`, so it is not possible to create an instance of `PosixThread`. This class adds one protected method, and one protected data member to those already defined in `PtThread`.

```
static void* runThis(PosixThread*);
```

This static method invokes the `run` method of the referenced thread. This provides a C interface that can be used by the POSIX thread library.

```
pthread_t thread;
```

A handle for the POSIX thread associated with the `PosixThread` object.

```
pthread_attr_t attributes;
```

A handle for the attributes associated with the POSIX thread.

```
int detach;
```

A flag to set the detached state of the POSIX thread.

```
void PosixThread::initialize()
{
    // Initialize attributes.
    pthread_attr_init(&attributes);

    // Detached threads free up their resources as soon as they exit.
    // Non-detached threads can be joined.
    detach = 0;
    pthread_attr_setdetachstate(&attributes, &detach);

    // New threads inherit their priority and scheduling policy
    // from the current thread.
    pthread_attr_setinheritsched(&attributes, PTHREAD_INHERIT_SCHED);

    // Set the stack size to something reasonably large. (32K)
    pthread_attr_setstacksize(&attributes, 0x8000);

    // Create a thread.
    pthread_create(&thread, &attributes, (pthread_func_t)runThis, this);

    // Discard temporary attribute object.
    pthread_attr_destroy(&attributes);
}
```

Figure 5.3: The `initialize` method of `PosixThread`.

```
void PosixThread::terminate()
{
    // Force the thread to terminate if it has not already done so.
    // Is it safe to do this to a thread that has already terminated?
    pthread_cancel(thread);

    // Now wait.
    pthread_join(thread, NULL);
    pthread_detach(&thread);
}
```

Figure 5.4: The `terminate` method of `PosixThread`.

```
DataFlowProcess(DataFlowStar& s)
    : star(s) {}
```

Figure 5.5: The constructor for `DataFlowProcess`.

The `initialize` method, shown in figure 5.3, initializes attributes, then creates a thread. The thread is created in a non-detached state, which makes it possible to later synchronize with the thread as it terminates. The controlling thread (usually the main thread) invokes the `terminate` method of a thread and waits for it to terminate. The priority and scheduling policy for the thread are inherited from the thread that creates it, usually the main thread. A function pointer to the `runThis` method and the `this` pointer, which points to the current `PosixThread` object, are passed as arguments to the `pthread_create` function. This creates a thread that executes `runThis`, and passes `this` as an argument to `runThis`. Thus, the `run` method of the `PosixThread` object is the main function of the thread that is created. The `runThis` method is required because it would not be good practice to pass a function pointer to the `run` method as an argument to `pthread_create`. Although the `run` method has an implicit `this` pointer argument by virtue of the fact that it is a class method, this is really an implementation detail of the C++ compiler. By using the `runThis` method, we make the pointer argument explicit and avoid any dependencies on a particular compiler implementation.

The `terminate` method, shown in figure 5.4, causes the thread to terminate before deleting the `PosixThread` object. First it requests that the thread associated with the `PosixThread` object terminate, using the `pthread_cancel` function. Then the current thread is suspended by `pthread_join` to give the cancelled thread an opportunity to terminate. Once termination of that thread is complete, the current thread resumes and deallocates resources used by the terminated thread by calling `pthread_detach`. Thus one thread can cause another to terminate by invoking the `terminate` method of that thread.

```
void DataFlowProcess::run()
{
    // Configure the star for dynamic execution.
    star.setDynamicExecution(TRUE);

    // Fire the Star ad infinitum.
    do
    {
        if (star.waitPort()) star.waitPort()->receiveData();
    } while(star.run());
}
```

Figure 5.6: The `run` method of `DataFlowProcess`.

5.1.3 DataFlowProcess

The class `DataFlowProcess` is derived from `PosixThread`. It implements the map higher order function described in section 2.1. A `DataFlowStar` is associated with each `DataFlowProcess` object.

`DataFlowStar& star;`

This protected data member refers to the dataflow star associated with the `DataFlowProcess` object.

The constructor, shown in figure 5.5, initializes the `star` member to establish the association between the thread and the star.

The `run` method, shown in figure 5.6, is defined to repeatedly invoke the `run` method of the star associated with the thread, just as the map function forms a process from repeated firings of a dataflow actor. Some dataflow stars in the BDF domain can operate with static scheduling or dynamic, run-time scheduling. Under static scheduling, a BDF star assumes that tokens are available on control inputs and appropriate data inputs. This requires that the scheduler be aware of the values of control tokens and the data ports that depend on these values. Because our scheduler has no such special knowledge, these stars must be properly configured for dynamic, multi-threaded execution in the PN domain. Stars in the BDF domain that have been configured for dynamic execution, and stars in the DDF domain dynamically inform the scheduler of data-dependent firing rules by designating a particular input `PortHole` with the `waitPort` method. Data must be retrieved from the designated input before invoking the star's `run` method. The star's `run` method is invoked repeatedly, until it indicates an error by returning "FALSE."

5.2 Communication Channels

Figure 5.7 shows the class derivation hierarchy for the classes that implement the communication channels of Kahn process networks. The classes that implement the communication channels provide the synchronization necessary to enforce the blocking read semantics of Kahn process networks. The classes

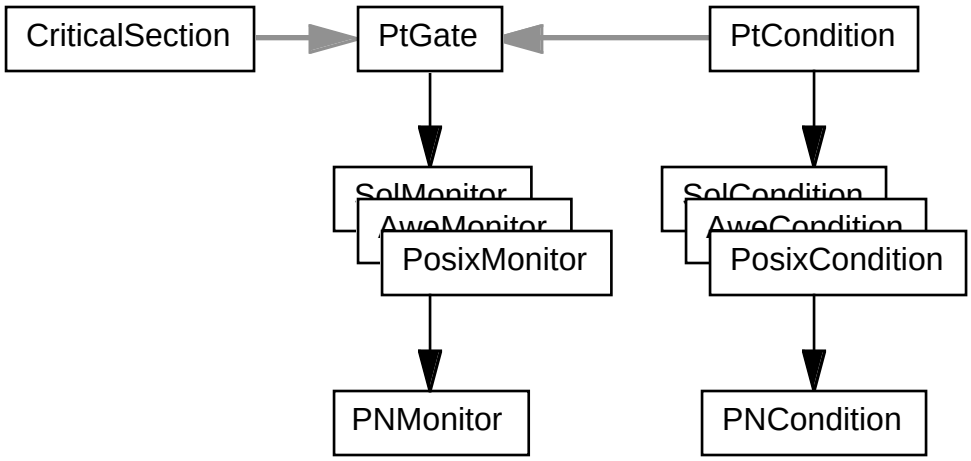

Figure 5.7: The class derivation hierarchy for monitors and condition variables. `PtGate` and `PtCondition` are abstract base classes, each with several possible implementations. Each `CriticalSection` and `PtCondition` refers to a `PtGate`.

`PtGate`, `PosixMonitor` and `CriticalSection` provide a mutual exclusion mechanism. The classes `PtCondition` and `PosixCondition` provide a synchronization mechanism. The class `PNGeodesic` uses these classes to implement a communication channel that enforces the blocking read operations of Kahn process networks and the blocking write operations required for bounded scheduling.

The abstract base class `PtGate` defines the interface for mutual exclusion objects in Ptolemy. The class `PosixMonitor` provides an implementation of `PtGate` based on the POSIX thread standard. Other implementations are possible. The class `PNMonitor` is a `typedef` that determines which implementation is used in the PN domain. Changing the underlying implementation simply requires changing this `typedef`.

The abstract base class `PtCondition` defines the interface for condition variables in Ptolemy. The class `PosixCondition` provides an implementation based on the POSIX thread standard. Other implementations are possible. The class `PNCondition` is a `typedef` that determines which implementation is used in the PN domain. Changing the underlying implementation simply requires changing this `typedef`.

The class `CriticalSection` provides a convenient method for manipulating `PtGate` objects, preventing some common programming errors. The class `PNGeodsesic` uses all of these classes to implement a communication channel.

5.2.1 PtGate

A `PtGate` can be locked and unlocked, but only one thread can hold the lock. Thus if a thread attempts to lock a `PtGate` that is already locked by another thread, it is suspended until the lock is released.

```
virtual void lock() = 0;
```

This protected method locks the `PtGate` object for exclusive use by one thread.

```
void PosixMonitor::lock()
{
    pthread_mutex_lock(&mutex);

    // Guarantee that the mutex will not remain locked
    // by a cancelled thread.
    pthread_cleanup_push((void(*)(void *))pthread_mutex_unlock, &mutex);
}
```

Figure 5.8: The `lock` method of `PosixMonitor`.

```
void PosixMonitor::unlock()
{
    // Remove cleanup handler and unlock.
    pthread_cleanup_pop(TRUE);
}
```

Figure 5.9: The `unlock` method of `PosixMonitor`.

`virtual void unlock() = 0;`

This protected method releases the lock on the `PtGate` object.

5.2.2 PosixMonitor

The class `PosixMonitor` provides an implementation for the interface defined by `PtGate`. It has a single protected data member.

`pthread_mutex_t thread;`

A handle for the POSIX monitor associated with the `PosixMonitor` object.

The implementations of the `lock` and `unlock` methods are shown in figures 5.8 and 5.9.

5.2.3 CriticalSection

The class `CriticalSection` provides a convenient mechanism for locking and unlocking `PtGate` objects. Its constructor, shown in figure 5.10 locks the gate. Its destructor, shown in figure 5.11

```
CriticalSection(PtGate* g) : mutex(g)
{
    if (mutex) mutex->lock();
}
```

Figure 5.10: The constructor of `CriticalSection`.

```
~CriticalSection()
{
    if (mutex) mutex->unlock();
}
```

Figure 5.11: The destructor of `CriticalSection`.

unlocks the gate. To protect a section of code, simply create a new scope and declare an instance of `CriticalSection`. The `PtGate` is locked as soon as the `CriticalSection` is constructed. When execution of the code exits scope, the `CriticalSection` destructor is automatically invoked, unlocking the `PtGate` and preventing errors caused by forgetting to unlock it. Examples of this usage are shown in figures 5.16 and 5.15. Because only one thread can hold the lock on a `PtGate`, only one section of code guarded in this way can be active at a given time.

5.2.4 PtCondition

The class `PtCondition` defines the interface for condition variables in Ptolemy. A `PtCondition` provides synchronization through the `wait` and `notify` methods. A condition variable can be used only when executing code within a critical section (i.e. when a `PtGate` is locked).

`PtGate& mon;`

This data member refers to the gate associated with the `PtCondition` object.

`virtual void wait() = 0;`

This method suspends execution of the current thread until notification is received. The associated gate is unlocked before execution is suspended. Once notification is received, the lock on the gate is automatically reacquired before execution resumes.

`virtual void notify() = 0;`

This method sends notification to one waiting thread. If multiple threads are waiting for notification, only one is activated.

`virtual void notifyAll() = 0;`

This method sends notification to all waiting threads. If multiple threads are waiting for notification, all of them are activated. Once activated, all of the threads attempt to reacquire the lock on the gate, but only one of them succeeds. The others are suspended again until they can acquire the lock on the gate.

5.2.5 PosixCondition

The class `PosixCondition` provides an implementation for the interface defined by `PtCondition`. The implementations of the `wait`, `notify` and `notifyAll` methods are shown

```
void PosixCondition::wait()
{
    pthread_cond_wait(&condition, &mutex);
}
```

Figure 5.12: The `wait` method of `PosixCondition`.

```
void PosixCondition::notify()
{
    pthread_cond_signal(&condition);
}
```

Figure 5.13: The `notify` method of `PosixCondition`.

in figures 5.12, 5.13 and 5.14.

5.2.6 PNGeodesic

The class `PNGeodesic`, which is derived from the class `Geodesic` defined in the Ptolemy kernel, implements the communication channels for the PN domain. In conjunction with the `PtGate` member provided in the base class `Geodesic`, two condition variables provide the necessary synchronization for blocking read and blocking write operations.

`PtCondition* notEmpty;`

This data member points to a condition variable used for blocking read operations when the channel is empty.

`PtCondition* notFull;`

This data member points to a condition variable used for blocking write operations when the channel is full.

`int cap;`

This data member represents the capacity of the communication channel and determines when it is full.

The `slowGet` method, shown in figure 5.15, implements the get operation for communication channels. The entire method executes within a critical section to ensure consistency of the object's data mem-

```
void PosixCondition::notifyAll()
{
    pthread_cond_broadcast(&condition);
}
```

Figure 5.14: The `notifyAll` method of `PosixCondition`.

```
Particle* PNGeodesic::slowGet()
{
    // Avoid entering the gate more than once.
    CriticalSection region(gate);
    while (sz < 1 && notEmpty) notEmpty->wait();
    sz--; Particle* p = pstack.get();
    if (sz < cap && notFull) notFull->notifyAll();
    return p;
}
```

Figure 5.15: The `slowGet` method of `PNGeodesic`.

```
void PNGeodesic::slowPut(Particle* p)
{
    // Avoid entering the gate more than once.
    CriticalSection region(gate);
    while (sz >= cap && notFull) notFull->wait();
    pstack.putTail(p); sz++;
    if (notEmpty) notEmpty->notifyAll();
}
```

Figure 5.16: The `slowPut` method of `PNGeodesic`.

bers. If the buffer is empty, then the thread that invoked `slowGet` is suspended until notification is received on `notEmpty`. Data is retrieved from the buffer, and if it is not full notification is sent on `notFull` to any other thread that may have been waiting.

The `slowPut` method, shown in figure 5.16, implements the put operation for communication channels. The entire method executes within a critical section to ensure consistency of the object's data members. If the buffer is full, then the thread that invoked `slowPut` is suspended until notification is received on `notFull`. Data is placed in the buffer, and notification is sent on `notEmpty` to any other thread that may have been waiting.

The `setCapacity` method, shown in figure 5.17, is used to adjust the capacity limit of communication channels. If the capacity is increased so that a channel is no longer full, notification is sent on `notFull` to any thread that may have been waiting.

```
void PNGeodesic::setCapacity(int c)
{
    cap = c;
    if (sz < cap && notFull) notFull->notifyAll();
}
```

Figure 5.17: The `setCapacity` method of `PNGeodesic`.

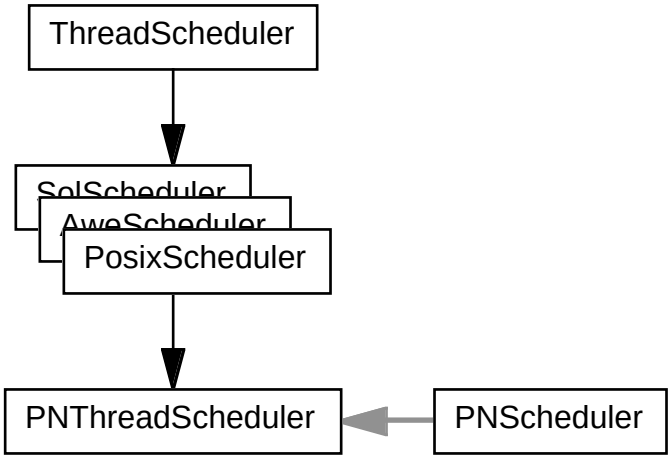

Figure 5.18: The class derivation hierarchy for schedulers. `ThreadScheduler` is an abstract base class with several possible implementations. Each `PNScheduler` refers to a `PNThreadScheduler`.

5.3 Scheduling

Figure 5.18 shows the class derivation hierarchy for the classes that implement the dynamic scheduling of Kahn process networks. The classes `ThreadScheduler` and `PosixScheduler` provide mechanisms for initiating and terminating groups of threads. These classes are used by `PNScheduler` to create threads for each node in the program graph. The class `SyncDataFlowProcess` implements the threads for the nodes.

5.3.1 ThreadScheduler

The abstract base class `ThreadScheduler` defines the interface of a container class for manipulating groups of threads. It has three public methods.

```
virtual void add(PtThread*) = 0;
```

This method adds a `PtThread` object to the container.

```
virtual void run() = 0;
```

This method causes all threads in the container to begin execution. It is intended that only the threads belonging to a particular `ThreadScheduler` object be affected by this method. This would permit multiple `ThreadScheduler` objects to exist and operate without interfering with each other. However, in practice this may not be possible. Often this causes all threads belonging to all `ThreadScheduler` objects to be activated.

```
virtual ~ThreadScheduler();
```

This method terminates and deletes all threads in the container.

```
// Start or continue the running of all threads.
void PosixScheduler::run()
{
    // Initialize attributes.
    pthread_attr_t attributes;
    pthread_attr_init(&attributes);

    // Lower the priority to let other threads run.
    pthread_getschedattr(mainThread, &attributes);
    pthread_attr_setprio(&attributes, minPriority);
    pthread_setschedattr(mainThread, attributes);

    // When control returns, restore the priority of this thread
    // to prevent others from running.
    pthread_getschedattr(mainThread, &attributes);
    pthread_attr_setprio(&attributes, maxPriority);
    pthread_setschedattr(mainThread, attributes);

    // Discard temporary attribute object.
    pthread_attr_destroy(&attributes);
}
```

Figure 5.19: The `run` method of `PosixScheduler`.

5.3.2 PosixScheduler

The class `PosixScheduler` provides an implementation of `ThreadScheduler` based on the POSIX thread standard. Other implementations are possible. The class `PNThreadScheduler` is a `typedef` that determines which implementation is used in the PN domain. Changing the underlying implementation simply requires changing this `typedef`.

The `add` method of `PosixScheduler` simply a thread to an internal list implemented by the class `ThreadList`. The `run` method, which is shown in figure 5.19, allows *all* threads (not just those in the list) to run by lowering the priority of the main thread. If execution of the threads ever stops, control returns to the main thread and its priority is raised again to prevent other threads from continuing. The `PosixScheduler` destructor invokes the `ThreadList` destructor, which is shown in figure 5.20. It causes all threads in the list to terminate.

5.3.3 PNScheduler

The class `PNScheduler` controls the execution of a process network. Three data members support synchronization between the scheduler and the processes.

`PNThreadScheduler* threads;`

A container for the threads managed by the scheduler.

```
// Destructor.  Delete all threads.
// This assumes that the threads have been dynamically allocated.
ThreadList::~ThreadList()
{
    // Delete all the threads in the list.
    for ( int i = size(); i > 0; i--)
    {
        PosixThread* t = (PosixThread*)getAndRemove();
        t->terminate();
        LOG_DEL; delete t;
    }
}
```

Figure 5.20: The destructor of `ThreadList`.

`PNMonitor* monitor;`

A monitor to guard the scheduler's condition variable.

`PNCondition* start;`

A condition variable for synchronizing with threads.

`int iteration;`

A counter for regulating the execution of the processes.

The `createThreads` method, shown in figure 5.21, creates one process for each node in the program graph. A `SyncDataFlowProcess` is created for each `DataFlowStar` and added to the `PNThreadScheduler` container.

It is often desirable to have a partial execution of a process network. The class `SyncDataFlowProcess`, which is derived from `DataFlowProcess`, supports this by synchronizing the execution of a thread with the `iteration` counter that belongs to the `PNScheduler`. The `run` methods of `PNScheduler` and `SyncDataFlowProcess` implement this synchronization. The `PNScheduler` `run` method, shown in figure 5.22, increments the `iteration` count to give every process an opportunity to run. The `SyncDataFlowProcess` `run` method, shown in figure 5.23, ensures that the number of invocations of the star's `run` method does not exceed the `iteration` count.

The `increaseBuffers` method is used during the course of execution to adjust the channel capacities according to the theory presented previously in chapter 4. Each time execution stops, the program graph is examined for full channels. If there are any full channels, then the capacity of the smallest one is increased.

```
// Create threads (dataflow processes).
void PNScheduler::createThreads()
{
    GalStarIter nextStar(*galaxy());
    DataFlowStar* star;

    LOG_NEW; threads = new PNThreadScheduler;

    // Create Threads for all the Stars.
    while((star = (DataFlowStar*)nextStar++) != NULL)
    {
        LOG_NEW; SyncDataFlowProcess* p
            = new SyncDataFlowProcess(*star,*start,iteration);
        threads->add(p);
        p->initialize();
    }
}
```

Figure 5.21: The `createThreads` method of `PNScheduler`.

```
// Run (or continue) the simulation.
int PNScheduler::run()
{
    if (SimControl::haltRequested())
    {
        Error::abortRun(*galaxy(), " cannot continue.");
        return FALSE;
    }

    while((currentTime < stopTime) && !SimControl::haltRequested())
    {
        // Notify all threads to continue.
        {
            CriticalSection region(start->monitor());
            iteration++;
            start->notifyAll();
        }
        threads->run();
        while (increaseBuffers() && !SimControl::haltRequested())
        {
            threads->run();
        }
        currentTime += schedulePeriod;
    }
    return !SimControl::haltRequested();
}
```

Figure 5.22: The `run` method of `PNScheduler`.

```
void SyncDataFlowProcess::run()
{
    int i = 0;

    // Configure the star for dynamic execution.
    star.setDynamicExecution(TRUE);

    // Fire the star ad infinitum.
    do
    {
        // Wait for notification to start.
        {
            CriticalSection region(start.monitor());
            while (iteration <= i) start.wait();
            i = iteration;
        }
        if (star.waitPort()) star.waitPort()->receiveData();
    } while (star.run());
}
```

Figure 5.23: The `run` method of `SyncDataFlowProcess`.

```
// Increase buffer capacities.
// Return number of full buffers encountered.
int PNScheduler::increaseBuffers()
{
    int fullBuffers = 0;
    PNGeodesic* smallest = NULL;

    // Increase the capacity of the smallest full geodesic.
    GalStarIter nextStar(*galaxy());
    Star* star;
    while ((star = nextStar++) != NULL)
    {
        BlockPortIter nextPort(*star);
        PortHole* port;
        while ((port = nextPort++) != NULL)
        {
            PNGeodesic* geo = NULL;
            if (port->isItOutput()
                && (geo = (PNGeodesic*)port->geo()) != NULL)
            {
                if (geo->size() >= geo->capacity())
                {
                    fullBuffers++;
                    if (smallest == NULL
                        || geo->capacity() < smallest->capacity())
                        smallest = geo;
                }
            }
        }
    }
    if (smallest != NULL)
        smallest->setCapacity(smallest->capacity() + 1);

    return fullBuffers;
}
```

Figure 5.24: The `increaseBuffers` method of `PNScheduler`.

Chapter 6

Conclusion

We have presented a scheduling policy for Kahn process networks that simultaneously satisfies our two requirements of non-termination and bounded buffering when possible. We do this by limiting the capacities of all communication channels and then increasing these capacities as needed to avoid deadlock.

We rely on the fact that Kahn process networks are determinate. The results produced by executing a program are unaffected by the order in which operations are carried out. In particular, deadlock is a property of the program itself and does not depend on the details of scheduling. Buffer sizes for the communication channels, on the other hand, do depend on the order in which get and put operations are carried out. By limiting the channel capacities, we place additional restrictions on the order of get and put operations. We have reduced the set of possible execution orders to those where the buffer sizes never exceed the capacity limits. If our model of computation were nondeterminate, the channel histories could be affected by scheduling decisions. In particular, one wrong scheduling decision could cause the system to deadlock or require unbounded buffering on one or more channels.

Our approach has some drawbacks. Execution of the entire program comes to a stop each time we encounter artificial deadlock, which can severely limit parallelism. Artificial deadlock occurs when the capacity limits are set too low, causing some processes to block writing to a full channel. All scheduling decisions are made dynamically during execution. We now discuss some topics for future research that may improve upon our policy.

6.1 Static Capacity Assignments

For simple process network models, such as synchronous dataflow process networks [31, 32] we can completely analyze a program and determine exactly what buffer sizes are required for the communication channels. By solving the balance equations, as described in section 2.3, we determine how many times each dataflow actor is fired in a complete cycle. We also know how many tokens are consumed and produced by each actor firing, so we immediately have a bound on the buffer sizes for the communication channels. Similar

analysis can sometimes (but not always) be done for Boolean dataflow process networks [9], as discussed in section 2.4. We could apply these analysis techniques to entire process networks, or to subsystems that obey the restricted synchronous or Boolean dataflow models. This would allow us to assign static capacity limits for some or all of the communication channels. There would be no need to adjust these capacities dynamically during execution, and we can also pre-allocate memory so that there is no run-time overhead associated with memory allocation.

6.2 Simple Directed Cycles

Limiting the channel capacities is equivalent to adding feedback channels, converting every connection into a directed cycle. It is the presence of the directed cycles that limits token production to give us strictly bounded buffer sizes. Some systems, such as the one in figure 1.4 on page 4 are already strictly bounded because every process is part of a directed cycle. There is no need to create additional directed cycles by adding feedback channels to every connection. This suggests that we could add fewer feedback channels and still have strictly bounded programs. This could give us more parallelism because there would be fewer directed cycles in the program graph.

Also, just because there is a directed cycle in the program graph, there is no guarantee that token production depends on token consumption. When we added feedback channels to every connection, we set up rules about how reads and puts proceed. Before writing to a data channel, a process must read from the corresponding feedback channel. By design there is a dependency between a given output and the corresponding feedback input. There is a similar dependency between inputs and their corresponding feedback outputs. Thus we know that this directed cycle does actually limit token production. For less general process models, such as synchronous dataflow and Boolean dataflow, the dependencies between inputs and outputs are known. In these cases, directed cycles can be analyzed to determine if they limit token production (and thus bound token accumulation) or if they introduce deadlock.

In our implementation we directly limit channel capacities and have blocking write and blocking read operations that simply increment or decrement a counter to keep track of the number of tokens buffered on a channel. If we reduce the number of feedback connections, then we cannot use this optimization. Instead we must make actual connections and send tokens across these feedback channels. This adds overhead that could exceed any savings realized by reducing the number of directed cycles. Future research could examine the effectiveness of the “optimizations” we are about to discuss.

6.2.1 K-Bounded Loops

In Culler’s work on K-bounded loops [11, 12], he forms a directed cycle around the body of a loop in order to limit the number of unconsumed tokens that can accumulate in a dataflow program. Thus, instead of forming a directed cycle for every connection, he forms a directed cycle from the outputs of a group of dataflow actors to their inputs. Figure 6.1 shows the original loop structure, and figure 6.2 shows Culler’s K-

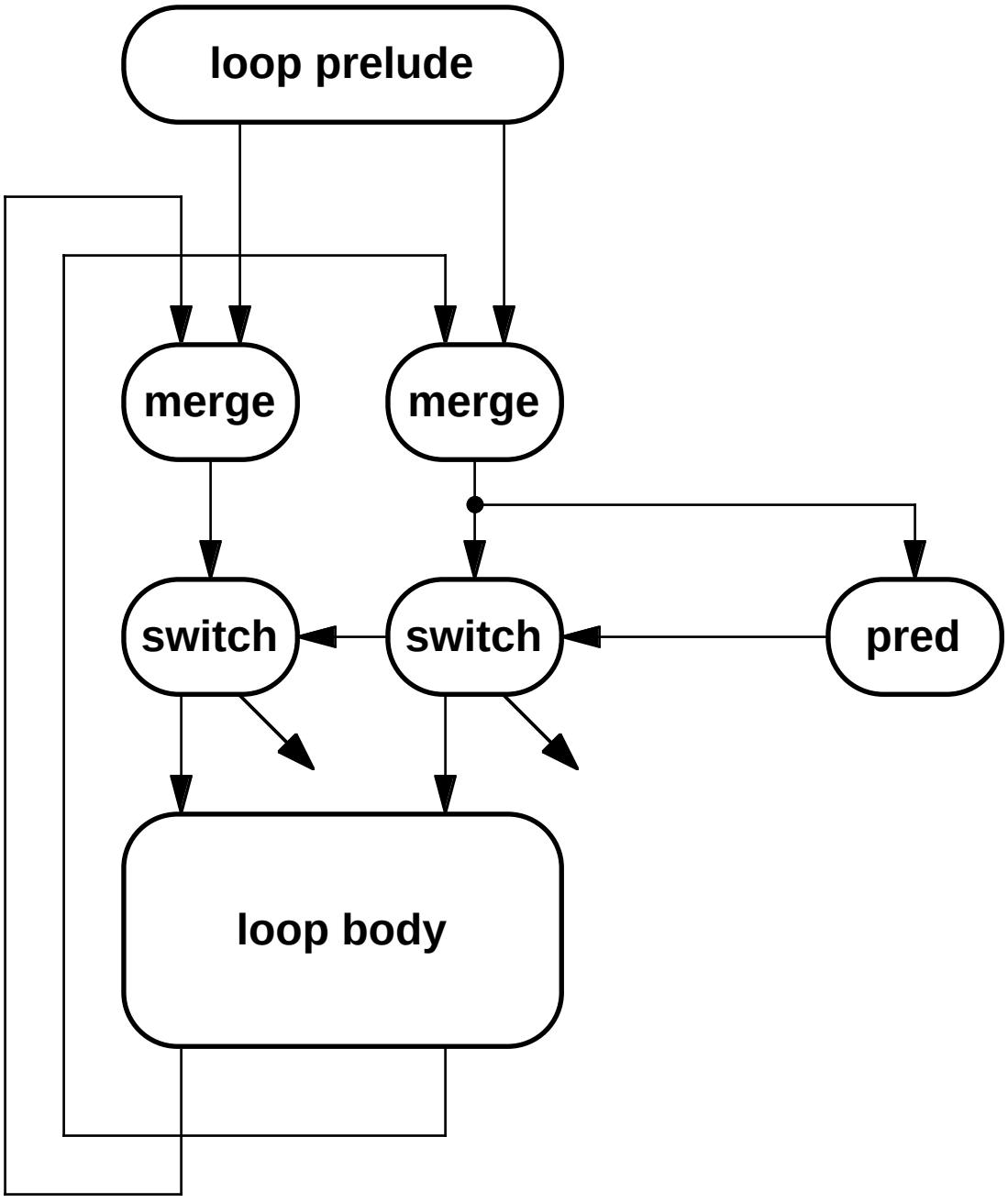

Figure 6.1: Loop graph schema.

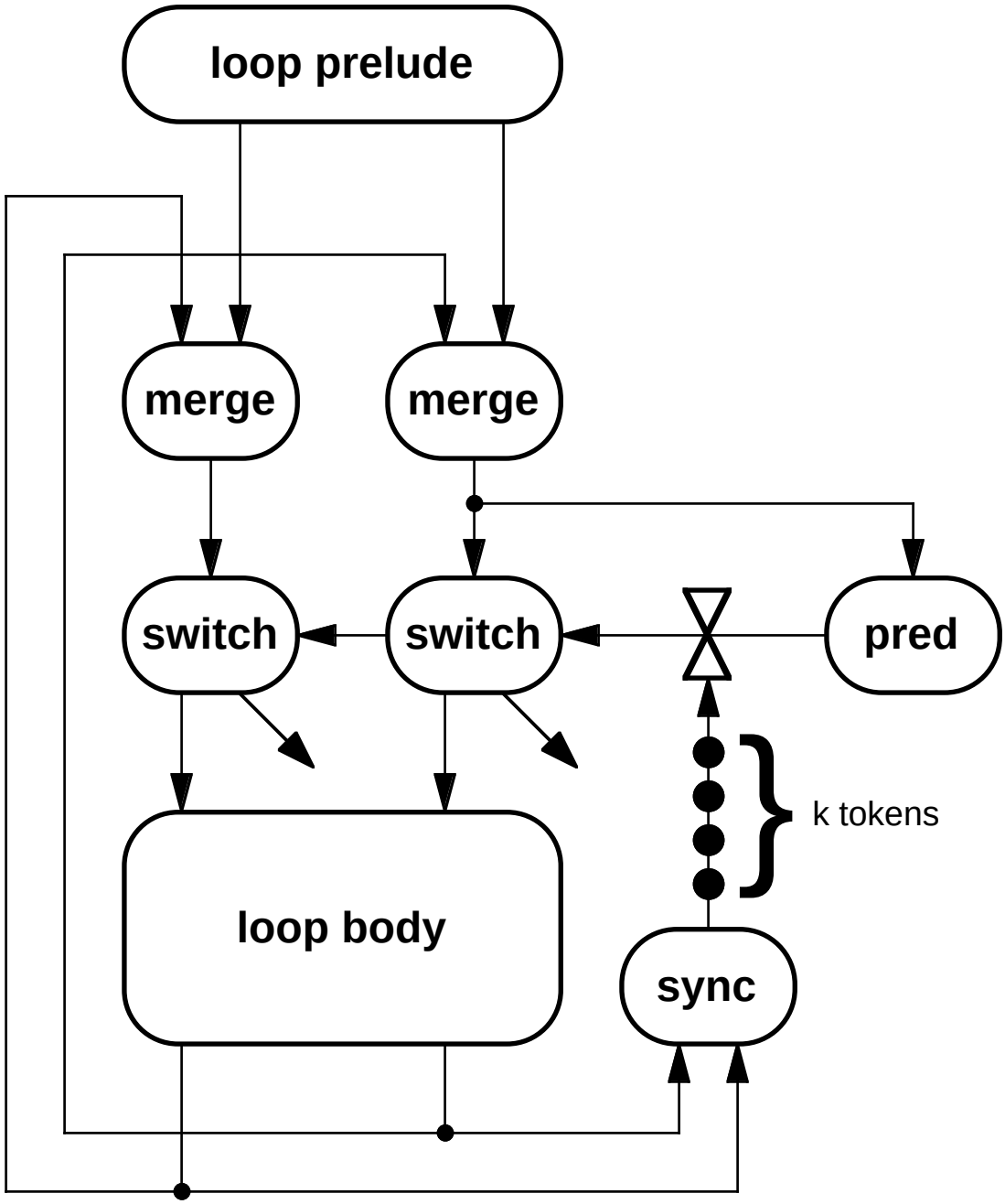

Figure 6.2: Culler's K-bounded loop graph schema.

bounded loop structure. A sync actor is inserted to to collect the outputs of the loop body and provide trigger tokens to a gate that is inserted to regulate the flow of tokens to the inputs. The number of initial trigger tokens on the gate's input, k, bounds the number of loop invocations that can proceed in parallel. His technique is only applied in cases where the program subgraph of the loop body is simple enough that the bound could be as low as $k = 1$ without introducing deadlock. But setting the bound so low limits buffering requirements on the communication channels, but it also severely limits parallelism. He examines strategies for setting k that balance parallelism and resource requirements for token storage.

One interesting difference with our approach is that no processes (or dataflow actors) need to be modified. We modified the semantics of the get and put operations. Culler's approach is to splice a few new actors into the graph, requiring no modification of the semantics of the model of computation.

6.2.2 Macroscopic Demand Propagation

Pingali's microscopic demand propagation algorithm (Micro-Prop [37]), described in section 3.3.2, is very similar to our approach in that feedback channels are added for every connection. He describes a macroscopic demand propagation algorithm (Macro-Prop [39]) which adds fewer feedback channels. Pingali identifies so called "steady state" sections of program loops. These sections are acyclic graphs that produce one set of output tokens for each set of input tokens. Thus there is no data-dependent consumption or production of tokens. Demand propagation code is added for the program sections as a whole instead of for individual operations as in the Micro-Prop algorithm. The Micro-Prop algorithm is used for demand-propagation code between the program sections identified by Macro-Prop. Pingali proves that programs produced by the Macro-Prop algorithm have the same input/output behavior as programs produced by the Micro-Prop algorithm. Thus we get equivalent programs with less overhead for demand propagation.

We could use some of these graph transformation techniques to identify sections of process networks which will remain strictly bounded if surrounded by a feedback connection.

6.3 Hybrid Static/Dynamic Scheduling

Whether we set some channel capacities statically, or introduce simple directed cycles, we are still scheduling all processes dynamically. Instead, we could analyze program graphs (or sections of graphs) that conform the the synchronous dataflow model or Boolean dataflow model. By solving the balance equations we determine how many times each dataflow actor is fired in a complete cycle. We can then construct a static schedule for a complete cycle and define the firing function of a large-grain dataflow actor to be one cycle of this schedule. We can now construct a single dataflow process from this large-grain actor. This gives us a hybrid scheduling framework where a fine-grain program graph is converted into a large-grain graph where scheduling of the large-grain graph is done dynamically, but the schedule within each large-grain process has been constructed statically.

This ensures that communication channels connecting statically scheduled actors remain bounded,

and the overhead of dynamic scheduling and synchronization is reduced. Similar clustering techniques are already used in Ptolemy for synchronous and Boolean dataflow [4, 5, 9]. However, these techniques require that the resulting large-grain dataflow actor still obey the same dataflow model as its fine-grain components: SDF actors combine to produce a large-grain SDF actor and BDF actors combine to produce a large-grain BDF actor. This restriction makes it possible to derive firing rules for the large-grain actors so that they can be treated the same as fine-grain actors by schedulers. We make no such restriction because we are scheduling the large-grain actor as a process and have no need to define firing rules for it. We simply rely on the blocking read semantics of Kahn process networks.

6.4 Code Generation

The PN domain implementation described in chapter 5 is a simulation domain in Ptolemy. An execution of a process network runs as part of Ptolemy. Instead, we could generate a C program that implements the process network. This requires only that a C compiler and suitable multi-threading mechanism be available for the target platform; there is no need to support all of Ptolemy. This makes it feasible to use the process network model to program embedded systems with limited resources.

Ptolemy already supports the generation of programs from dataflow graphs [40]. For synchronous dataflow process networks and some boolean dataflow process networks, a static interleaving of process execution can be compiled into the C program itself so that true multi-threading is not necessary. Separate C programs are generated for each processor of a multiprocessor system. This code generation mechanism could be extended to create programs with multiple threads that are dynamically scheduled.

6.4.1 Reactive C

One advantage of a multi-threaded programming style is that when control returns to a suspended process, it resumes execution at the point where it left off. It does not have to begin from a procedure entry point. When implemented in C++, the firing function for a dataflow actor is complicated by the need to test and maintain state, as shown in figure 6.3. This is required so that the actor performs a different function each time it is fired. Reactive C [6, 8] is a C pre-processor that automatically generates the `switch` statement to achieve this functionality. Figure 6.4 shows how the same function might be implemented if Reactive C were extended to the C++ language. If get and put operations were written in Reactive C++, as in figures 6.5 and 6.6, then this function could be implemented in a multi-threaded style as in figure 6.7. Reactive C [6, 8] has been used as the basis of a process network language [7] that is very similar to Kahn and MacQueen's language [25]. The implementation of the `GET` operation is similar to the example in figures 6.5. Figure 6.8 shows how a select process would be implemented in this language.

Reactive C is simply a pre-processor that generates C programs. These programs interleave the executions of multiple processes to simulate parallelism. There very little overhead for context switching, and there is no need to maintain a separate stack for each thread, as in POSIX threads. It would be interesting to

```
void go()
{
    switch(state)
    {
    case 0:
        control.receiveData();
        if (control%0) input = trueInput;
        else input = falseInput;
        waitFor(input);
        state = 1;
        break;

    case 1:
        input.receiveData();
        output%0 = input%0;
        output.sendData();
        waitFor(control);
        state = 0;
    }
}
```

Figure 6.3: The C++ code for the select dataflow actor.

```
void go()
{
    control.receiveData();
    if (control%0) input = trueInput;
    else input = falseInput;
    waitFor(input);

    stop;

    input.receiveData();
    output%0 = input%0;
    output.sendData();
    waitFor(control);
}
```

Figure 6.4: The Reactive C++ code for the select dataflow actor.

```
Particle* PNGeodesic::slowGet()
{
    while (sz < 1) stop;
    sz--; Particle* p = pstack.get();
    return p;
}
```

Figure 6.5: The Reactive C++ code for the get operation.

```
void PNGeodesic::slowPut(Particle* p)
{
    while (sz >= cap) stop;
    pstack.putTail(p); sz++;
}
```

Figure 6.6: The Reactive C++ code for the put operation.

```
go
{
    control.receiveData();
    if (control%0) input = trueInput;
    else input = falseInput;
    input.receiveData();
    output%0 = input%0;
    output.sendData();
}
```

Figure 6.7: The multi-threaded code for the select dataflow actor.

```
PROCESS select(Channel control,Channel t,Channel f, Channel out)
{
    VAR int val;

    for(;;)
    {
        GET(control,val);
        if (val) PUT(out,GET(t));
        else PUT(out,GET(f));
    }
}
```

Figure 6.8: A reactive select process.

extend this to the C++ language (Reactive C++) and to implement the PUT operation as in figure 6.6 to support the bounded scheduling theory developed in this thesis. Reactive C++ could be used to implement the PN domain in Ptolemy, and Reactive C could be used as a target language for code generation.

6.4.2 POSIX Threads

One advantage of using POSIX threads is the opportunity for parallel execution. When threads are built into the operating system, programs can be automatically parallelized. The same program can execute on uniprocessor workstations and multiprocessor workstations without the need to recompile. When multiple processors are available, multiple threads can execute in parallel. Even on uniprocessor workstations there is an advantage to multi-threaded execution: the possibility to overlap communication with computation. While one thread is blocked waiting for a file access to complete, another thread can continue to do useful work.

We could generate C programs that use POSIX threads to implement process networks. If we generate C++ programs instead of C, then we can used exactly the same classes as Ptolemy uses to implement threads.

Bibliography

[1] Arvind, L. Bic, and T. Ungerer. Evolution of data-flow computers. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 1, pages 3–33. Prentice Hall, Englewood Cliffs, 1991.

[2] E. A. Ashcroft, A. A. Faustini, and B. Huey. Eduction: A model of parallel computation and the programming language Lucid. In *Phoenix Conf. Comput. Comm.*, pages 9–15, Scottsdale, Arizona, Mar. 1985.

[3] E. A. Ashcroft, R. Jagannathan, A. A. Faustini, and B. Huey. Eazyflow engines for Lucid: A family of supercomputer architectures based upon demand driven and data driven computation. In *Int. Conf. Supercomput. Syst.*, pages 513–523, St. Petersburg, Florida, Dec. 1985.

[4] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee. A scheduling framework for minimizing memory requirements of multirate DSP systems represented as dataflow graphs. In *IEEE Workshop VLSI Sig. Proc.*, pages 188–196, Veldhoven, Netherlands, Oct. 1993.

[5] S. S. Bhattacharyya and E. A. Lee. Scheduling synchronous dataflow graphs for efficient looping. *J. VLSI Sig. Proc.*, 6(3):271–288, Dec. 1993.

[6] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Softw. Pract. Exp.*, 21(4):401–428, 1991.

[7] F. Boussinot. Nets of reactive processes. ftp://cma.cma.fr/pub/rc/nrp/doc/nrp.ps.Z, Dec. 1994.

[8] F. Boussinot and G. Doumenc. *RC Reference Manual*. Ecole Nationale Supérieure des Mines de Paris, Sophia-Antipolis, France, 1992. ftp://cma.cma.fr/pub/rc/reports/rapport16-92.ps.Z.

[9] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, U. C. Berkeley, 1993. http://ptolemy.eecs.berkeley.edu/papers/jbuckThesis.

[10] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. J. Comput. Sim.*, 4:155–182, Apr. 1994. http://ptolemy.eecs.berkeley.edu/papers/JEurSim.

[11] D. Culler. Resource requirements of dataflow programs. In *Int. Symp. Comput. Arch.*, pages 141–150, Honolulu, Hawaii, 1988.

[12] D. E. Culler. Managing parallelism and resources in scientific dataflow programs. Tech. Report MIT/LCS/TR-446, MIT Lab. Comput. Sci., 545 Technology Square, Cambridge, MA 02139, Nov. 1989. PhD Thesis.

[13] J. Dennis. First version of a data flow procedure language. In *Prog. Symp.*, pages 362–376, Paris, France, Apr. 1974.

[14] J. B. Dennis. The evolution of "static" data-flow architecture. In *Advanced Topics in Data-Flow Computing*, chapter 2, pages 35–91. Prentice Hall, Englewood Cliffs, 1991.

[15] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond multiprocessing: Multithreading the SunOS kernel. In *USENIX*, pages 11–18, San Antonio, Texas, June 1992. http://www.Sun.COM/sunsoft/Developer-products/sig/threads/papers/beyond_mp.ps.

[16] G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved dataflow programs for DSP computation. In *ICASSP*, pages 561–564, San Francisco, California, Mar. 1992.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[18] J.-L. Gaudiot and L. Bic, editors. *Advanced Topics in Data-Flow Computing*. Prentice Hall, Englewood Cliffs, 1991.

[19] E. W. Giering, F. Müller, and T. P. Baker. Implementing Ada 9X features using POSIX threads: Design issues. In *TRI-Ada*, pages 214–228, Sept. 1993. ftp://ftp.cs.fsu.edu/pub/PART/publications/triada93.ps.gz.

[20] D. Grunwald. A users guide to AWESIME: An object oriented parallel programming and simulation system. Tech. Report CU-CS-552-91, Department of Computer Science, University of Colorado, Boulder, Nov. 1991. ftp://cs.colorado.edu/pub/cs/distribs/Awesime/users-guide.ps.Z.

[21] B. M. Huey, A. A. Faustini, and E. A. Ashcroft. An eduction engine for Lucid: A highly parallel computer architecture. In *Phoenix Conf. Comput. Comm.*, pages 156–160, Scottsdale, Arizona, Mar. 1985.

[22] R. Jagannathan and E. A. Ashcroft. Eazyflow: A hybrid for parallel processing. In *Int. Conf. Par. Proc.*, pages 514–523, Bellaire, Michigan, Aug. 1984.

[23] R. Jagannathan and E. A. Ashcroft. Eazyflow engine architecture. In *Phoenix Conf. Comput. Comm.*, pages 161–165, Scottsdale, Arizona, Mar. 1985.

[24] G. Kahn. The semantics of a simple language for parallel programming. In *Info. Proc.*, pages 471–475, Stockholm, Aug. 1974.

[25] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Info. Proc.*, pages 993–998, Toronto, Aug. 1977.

[26] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, and queueing. *SIAM J. Applied Math.*, 14(6):1390–1411, Nov. 1966.

[27] P. H. J. Kelly. Programming a network of processing elements using an extended functional programming language. In *CONPAR*, pages 556–565, Manchester, Sept. 1988.

[28] S. Khanna, M. Sebree, and J. Zolnowsky. Realtime scheduling in SunOS 5.0. In *USENIX*, pages 375–390, San Francisco, California, Jan. 1992. http://www.Sun.COM/sunsoft/Developer-products/sig/threads/papers/rt_sched.ps.

[29] S. Kleiman, B. Smaalders, D. Stein, and D. Shah. Writing multithreaded code in Solaris. In *COMPCON*, pages 187–192, San Francisco, Californa, Feb. 1992. http://www.Sun.COM/sunsoft/Developer-products/sig/threads/papers/writing_mt_code.ps.

[30] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner. Symmetric multiprocessing in Solaris 2.0. In *COMPCON*, pages 181–186, San Francisco, California, Feb. 1992. http://www.Sun.COM/sunsoft/Developer-products/sig/threads/papers/symmetric_mp.ps.

[31] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, C-36(1):24–35, Jan. 1987.

[32] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, Sept. 1987.

[33] E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. IEEE*, pages 773–799, May 1995. http://ptolemy.eecs.berkeley.edu/papers/processNets.

[34] F. Müller. Implementing POSIX threads under UNIX: Description of work in progress. In *Softw. Eng. Research Forum*, pages 253–261, Nov. 1992. ftp://ftp.cs.fsu.edu/pub/PART/publications/pthreads_serf92.ps.gz.

[35] F. Müller. A library implementation of POSIX threads under UNIX. In *USENIX*, pages 29–41, Jan. 1993. ftp://ftp.cs.fsu.edu/pub/PART/publications/pthreads_usenix93.ps.gz.

[36] F. Müller. Pthreads library interface. ftp://ftp.cs.fsu.edu/pub/PART/publications/pthreads_interface.ps.gz, July 1995.

[37] K. Pingali and Arvind. Efficient demand-driven evaluation. part 1. *ACM Trans. Prog. Lang. Syst.*, 7(2):311–333, Apr. 1985.

[38] K. Pingali and Arvind. Clarification of "feeding inputs on demand" in efficient demand-driven evaluation part 1. *ACM Trans. Prog. Lang. Syst.*, 8(1):140–141, Jan. 1986.

[39] K. Pingali and Arvind. Efficient demand-driven evaluation. part 2. *ACM Trans. Prog. Lang. Syst.*, 8(1):109–139, Jan. 1986.

[40] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software synthesis for dsp using Ptolemy. *J. VLSI Sig. Proc.*, 9(1):7–21, Jan. 1995. http://ptolemy.eecs.berkeley.edu/papers/jvsp_codegen.

[41] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *USENIX*, pages 65–79, Dallas, Texas, Jan. 1991. http://www.Sun.COM/sunsoft/Developer-products/sig/threads/papers/sunos_mt_arch.ps.

[42] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX*, pages 1–9, San Antonio, Texas, June 1992. http://www.Sun.COM/sunsoft/Developer-products/sig/threads/papers/impl_threads.ps.

[43] SunSoft, Inc. Solaris 2.4 multithreaded programming guide, Aug. 1994. http://www.Sun.COM/sunsoft/Developer-products/sig/threads/doc/MultithreadedProgrammingGuide_Solaris24.ps.

[44] W. W. Wadge. An extensional treatment of dataflow deadlock. In *Semantics of Concurrent Computations*, pages 285–299, Evian, France, July 1979.