Chapter 9. PN domain

Authors: *Thomas M. Parks*

Other Contributors: *Brian Evans*
Christopher Hylands

9.1 Introduction

In the process network model of computation, concurrent processes communicate through unidirectional first-in first-out 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, we want to execute process network programs forever with bounded buffering on the communication channels whenever possible. [Par95]

The process network (PN) domain is an experimental implementation of process network model of computation. The PN domain is a superset of the synchronous dataflow (SDF), Boolean dataflow (BDF), and dynamic dataflow (DDF) domains, as shown in Figure 1-2. Thus, any SDF, BDF, or DDF star can be used in the PN domain. In the dataflow subdomains, stars represent dataflow actors, which consume and produce a finite number of particles when they are fired. In the PN domain, stars represent processes, which consume and produce (possibly infinite) streams of particles. When a dataflow actor from the SDF, BDF or DDF domain is used in a PN system, a *dataflow process* is created that repeatedly fires that actor. A separate thread of execution is created for each process. Thread synchronization mechanisms ensure that a thread attempting to read from an empty input is automatically suspended, and threads automatically wake up when data becomes available.

The current implementation of the PN domain is based on a user-level POSIX thread library called Pthreads [Mue93,Mue95]. By choosing the POSIX standard, we improve the portability of the PN domain. Several workstation vendors already include an implementation of POSIX threads in their operating systems, such as Solaris 2.5 and HP-UX 10. Having threads built into the operating system, as opposed to a user-level library implementation, offers the opportunity for automatic parallelization on multiprocessor workstations. That is, the same program would run on uniprocessor workstations and multiprocessor workstations without needing to be recompiled. When multiple processors are available, multiple threads can execute in parallel. Even on uniprocessor workstations, multi-threaded execution offers the advantage that communication can be overlapped with computation.

9.2 Process networks

Kahn describes a model of computation where processes are connected by communication channels to form a network [Kah74,Kah77]. Processes produce data elements or *tokens* and send them along a unidirectional communication channel where they are stored in a first-in first-out order until the destination process consumes them. Communication channels are the *only* method processes may use to exchange information. A set of processes that communicate through a network of first-in first-out queues defines a *program*.

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 [Kah74]. Therefore, we can apply different scheduling algorithms without affecting the results produced by executing a program.

9.2.1 Dataflow process networks

Dataflow is a model of computation that is a special case of process networks. 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. For example, when applied to an infinite input stream a firing function f may consume just one token and produce one output token:

$$f([x_1, x_2, x_3, \ldots]) = f(x_1)$$

To produce an infinite output stream, the actor must be fired repeatedly. A process formed from repeated firings of a dataflow actor is called a *dataflow process* [Lee95]. The higher-order function *map* converts an actor firing function f into a process:

$$map(f)[x_1, x_2, x_3, \ldots] = [f(x)_1, f(x)_2, f(x)_3, \ldots]$$

A higher-order function takes a function as an argument and returns another function. When the function returned by $map(f)$ is applied to the input stream $[x_1, x_2, x_3, \ldots]$ the result is a stream in which the firing function f is applied point-wise to each element of the input stream. The *map* function can also be described recursively using the stream-building function *cons*, which inserts an element at the head of a stream:

$$map(f)[x_1, x_2, x_3, \ldots] = cons(f(x_1), map(f)[x_2, x_3, \ldots])$$

The use of map can be generalized so that f can consume and produce multiple tokens on multiple streams [Lee95].

Breaking a process down into smaller units of execution, such as dataflow actor firings, makes efficient implementations of process networks possible. The SDF, BDF, and DDF domains implement dataflow process networks by scheduling the firings of dataflow actors. The actor firings of one dataflow process are interleaved with the firings of other processes in a sequence that guarantees the availability of tokens required for each firing. In the PN domain, a dataflow process is created for each dataflow actor. A separate thread of execution is created for each process, and the interleaving of threads is performed automatically. Unlike the dataflow domains, the firing of a dataflow actor is *not* an atomic operation in the PN domain. Because the scheduler does not guarantee the availability of tokens, the firing of an actor can be suspended if it attempts to read data from an empty input channel.

9.2.2 Scheduling dataflow process networks

Because 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 read and write operations are carried out.

For Kahn process networks, no finite-time algorithm can decide whether or not a program will terminate or require bounded buffering. Since we are interested in programs that will never terminate, a scheduler has infinite time to decide these questions. Parks [Par95] developed a scheduling policy that will execute arbitrary Kahn process networks forever with bounded buffering when possible. To enforce bounded buffering, Parks limits channel capacities, which places additional restrictions on the order of read and write operations. Parks reduces the set of possible execution orders to those where the buffer sizes never exceed the capacity limits. In this approach, 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 when writing to a full channel. All scheduling decisions are made dynamically during execution.

9.2.3 Iterations in the PN domain

In a complete execution of a program, the program terminates if and only if all processes block attempting to consume data from empty communication channels. Often, it is desirable to have a partial execution of a process network. An iteration in the PN domain is defined such that no actor in a dataflow process will fire more than once. Some actors may not fire, or may fire partially in an iteration if insufficient tokens are available on their inputs.

9.3 Threads

The PN domain creates a separate thread of execution for each node in the program 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 (shared memory). 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 the monitor. If a thread attempts to lock a monitor that is already locked by another thread, then it will be 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.

9.4 An overview of PN stars

The "open-palette" command in pigi ("O") will open a checkbox window that you can use to open the standard palettes in all the installed domains. For the PN domain, the star library is small enough that it is easily contained entirely in one palette, shown in figure 9-1

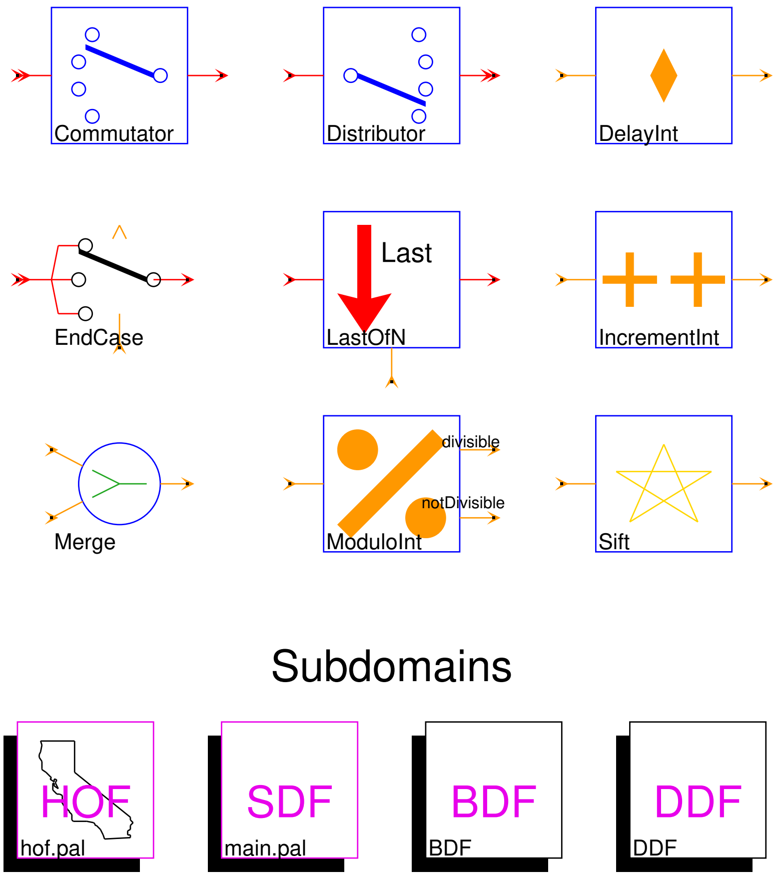

FIGURE 9-1: The palette of stars for the PN domain.

Many of these stars are re-implementations of similarly named stars in the SDF and DDF domains. These implementations take advantage of the multi-threaded nature of execution in the PN domain.

`Commutator` Takes N input streams (where N is the number of inputs) and synchronously combines them into one output stream. It consumes B particles from an input (where B is the blockSize), and produces B particles on the output, then it continues by reading from the next input. The first B particles on the output come from the first input, the next B particles from the next input, etc.

`Distributor` Takes one input stream and splits it into N output streams, where N is the number of outputs. It consumes B input particles, where B = blockSize, and sends them to the first output. It consumes another B input particles and sends them to the next

	output, etc.
`DelayInt`	An initializable delay line.
`EndCase`	Depending on the "control" particle, consume a particle from one of the data inputs and send it to the output. The value of the control particle should be between zero and N-1, where N is the number of data inputs.
`LastOfN`	Given a control input with integer value N, consume N particles from the data input and produce only the last of these at the output.
`Merge`	Merge two increasing sequences, eliminating duplicates.
`IncrementInt`	Increment the input by a constant.
`ModuloInt`	Divides the input stream into a stream of numbers divisible by N and another stream of numbers that are not divisible by N.

9.5 An overview of PN demos

There are two subpalettes of PN domain demos, a palette of examples from papers by Gilles Kahn and David B. MacQueen, and a palette of examples from the Ph.D. thesis of Thomas M. Parks. The top-level palette for demos in the Process Network domain is shown in figure 9-2. The subpalettes are described below.

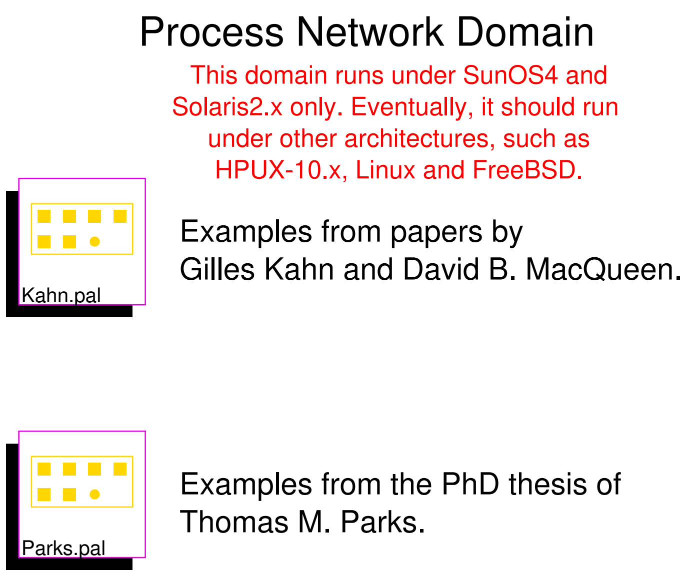

FIGURE 9-2: The top-level palette for PN demos.

9.5.1 Examples from papers by Gilles Kahn and David B. MacQueen

These demos are examples from papers by Gilles Kahn and David B. MacQueen. The palette is shown in figure 9-3.

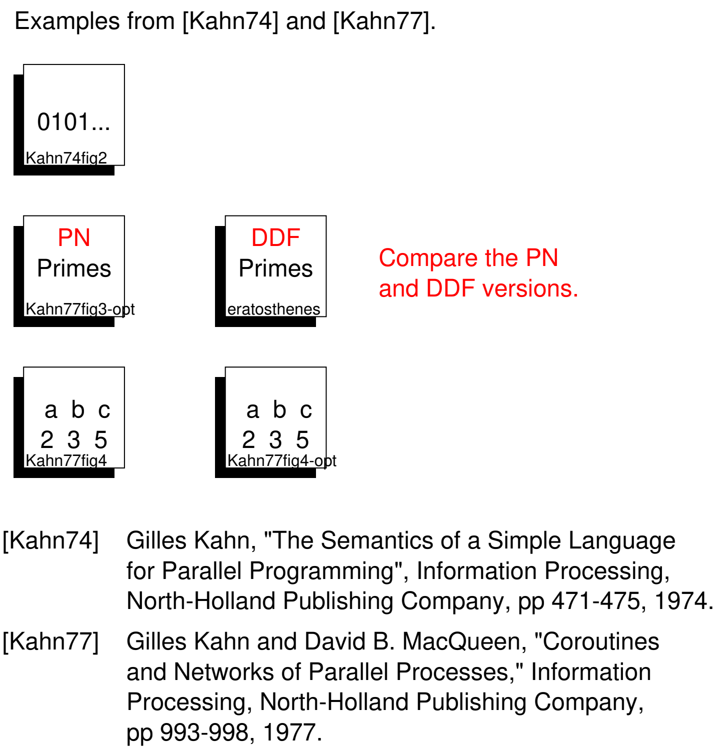

FIGURE 9-3: PN domain demos of examples from papers by Gilles Kahn and David B. MacQueen

`Kahn74fig2` Produce a stream of 0's and 1's. This demo is from figure 2 in [Kah74].

`Kahn77fig3-opt` Sieve of Eratosthenes with non-recursive sift process. This example shows how process networks can change dynamically during execution. The sift process inserts new filter processes to eliminate multiples of newly discovered primes. This demo is from figure 3 in [Kah77].

`eratosthenes` Compare this DDF domain demo with the Kahn77fig3-opt demo above.

`Kahn77fig4` Produce a sequence of integers of the form $2^a 3^b 5^c$. An unbounded number of tokens accumulate in the communication channels as execution progresses. This demo is from figure 4 in [Kah77].

`Kahn77fig4-opt` Produce a sequence of integers of the form $2^a 3^b 5^c$ with optimizations to avoid generating duplicate values. An unbounded number of tokens accumulate in the communication channels as execution progresses. This demo is from figure 4 in [Kah77].

9.5.2 Examples from the Ph.D. thesis of Thomas M. Parks

These demos are examples from the Ph.D. thesis of Thomas M. Parks. The palette is show in figure 9-4.

Examples from [Parks95].

ʼarks95] Thomas M. Parks, Bounded Scheduling of Process Networks, Technichal Report UCB/ERL-95-105, PhD Dissertation, EECS Department, University of California, Berkeley, December 1995.

FIGURE 9-4: PN domain demos of examples from the Ph.D. thesis of Thomas M. Parks.

`Parks95fig3.5` Merge two streams of monotonically increasing integers (multiples of 2 and 3) to produce a stream of monotonically increasing integers with no duplicates. Simple data-driven execution of this example would result in unbounded accumulation of tokens, while demand-driven execution requires that only a small number of tokens be stored on the communication channels. This demo is from figure 3.5 in [Par95].

`Parks95fig3.11` Separate a stream of monotonically increasing integers into those values that are and are not evenly divisible by 3. Simple demand-driven execution of this example would result in unbounded accumulation of tokens, while data-driven execution requires that only a small number of tokens be stored on the communication channels. This demo is from figure 3.11 in [Par95].

`Parks95fig4.1` Separate an increasing sequence of integers into those values that are and are not evenly divisible by 5, then merge these two streams to reproduce a stream of increasing integers. Simple data-driven or demand-driven execution of this example would result in unbounded accumulation of tokens. This demo is from figure 4.1 in [Par95].