DATAFLOW PROCESS NETWORKS

by

Edward A. Lee

Memorandum No. UCB/ERL M94/53

19 July 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DATAFLOW PROCESS NETWORKS

by

Edward A. Lee

Memorandum No. UCB/ERL M94/53

19 July 1994

DATAFLOW PROCESS NETWORKS

by

Edward A. Lee

Memorandum No. UCB/ERL M94/53

19 July 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DATAFLOW PROCESS NETWORKS

Edward A. Lee

Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

This paper reviews a model of computation used in industrial practice in signal processing software environments and experimentally in other contexts. It gives this model the name "dataflow process networks," and studies its formal properties as well as its utility as a basis for programming language design. Variants of this model are used in commercial visual programming systems such as SPW from the Alta Group of Cadence (formerly Comdisco Systems), COSSAP from Synopsys (formerly Cadis), the DSP Station from Mentor Graphics, and Hypersignal from Hyperception. They are also used in research software such as Khoros from the University of New Mexico and Ptolemy from the University of California at Berkeley.

Dataflow process networks are shown to be a special case of Kahn process networks, a model of computation where a number of concurrent processes communicate through unidirectional FIFO channels, where writes to the channel are non-blocking, and reads are blocking. In dataflow process networks, each process consists of repeated "firings" of a dataflow "actor". An actor defines a (usually functional) quantum of computation. By dividing processes into actor firings, the considerable overhead of context switching incurred in most implementations of Kahn process networks is avoided.

This paper relates the dataflow process networks to other dataflow models, including those used in dataflow machines, such as static dataflow and the tagged-token model. It also relates dataflow process networks to functional languages such as Haskell, and shows that modern language concepts such as higher-order functions and polymorphism can be used very effectively in dataflow process networks. A number of programming examples using a visual syntax are given.

This research in sponsored in part by ARPA, under the RASSP program, in cooperation with the United States Air Force, and by the National Science Foundation (MIP9201605).

1.0 Motivation

This paper concerns programming methodologies commonly called "graphical dataflow programming" that are used extensively for signal processing and experimentally for other applications. In this paper, "graphical" means simply that the program is explicitly specified by a directed graph where the nodes represent computations and the arcs represent streams. The graphs are typically hierarchical, in that a node in a graph may represent another directed graph. The nodes in the graph can be either language primitives or subprograms specified in another language, such as C or Fortran.

It is common in the signal processing community to use a visual syntax to specify such graphs, in which case the model is often called "visual dataflow programming." But it is by no means essential to use a visual syntax. At least one commercial graphical programming environment (Mentor Graphics' DSP Station) allows an arbitrary mixture of visual and textual specification, both based on the applicative language Silage [35]. Several other languages with related semantics, such as SIGNAL [9][52] and Sisal [57] are used primarily or exclusively with textual syntax. The language LUCID [75][77], while primarily used with textual syntax, has experimental visual forms [7].

Hierarchy in graphical program structure can be viewed as an alternative to the more usual abstraction of subprograms via procedures, functions, or objects. It is better suited than any of these to a visual syntax, and also better suited to signal processing.

Some examples of graphical dataflow programming environments intended for signal processing (including image processing) are Khoros, from the University of New Mexico [67], Ptolemy, from the University of California at Berkeley [15], the signal processing worksystem (SPW), from the Alta Group at Cadence (formerly Comdisco Systems), COSSAP, from Synopsys (formerly Cadis), and the DSP Station, from Mentor Graphics (formerly EDC). A survey of graphical dataflow languages for other applications is given by Hills [36]. These software environments all claim variants of dataflow semantics, but a word of caution is in order. The term "dataflow" is often used very loosely for semantics that bear little resemblance to those outlined

by Dennis in 1975 [23]. A major motivation of this paper is to point out a rigorous formal underpinning for dataflow graphical languages, to establish precisely the relationship between such languages and functional languages, and to show that such languages benefit significantly from such modern programming concepts as polymorphism, strong typing, and higher-order functions. Although it has been rarely exploited in visual dataflow programming, I also show that such languages can make very effective use of recursion.

Most graphical signal processing environments do not define a language in any strict sense. In fact, some designers of such environments advocate minimal semantics [60], arguing that the graphical organization by itself is sufficient to be useful. The semantics of a program in such environments is determined by the contents of the graph nodes, either subgraphs or a subprograms. Subprograms are usually specified in a conventional programming language such as C. Most such environments, however, including Khoros, SPW, and COSSAP, take a middle ground, permitting the nodes in a graph or subgraph to contain arbitrary subprograms, but defining precise semantics for the interaction between nodes. Following Halbwachs [32], I call the language used to define the subprograms in nodes the *host language*. Following Jagannathan, I call the language defining the interaction between nodes the *coordination language* [41].

Many possibilities have been explored for precise semantics of coordination languages, including for example the computation graphs of Karp and Miller [46], the synchronous dataflow graphs of Lee and Messerschmitt [50], the Processing Graph Method (PGM) of Kaplan, *et al.* [45], Granular Lucid [41], and others [2][18][21][41][76]. Many of these limit expressiveness in exchange for considerable advantages such as compile-time predictability.

Graphical programs can be either interpreted or compiled. It is common in signal processing environments to provide both options. The output of compilation can be a standard procedural language, such as C, assembly code for programmable DSP processors [63], or even specifications of silicon implementations [22]. Often, considerable effort is put into optimized compilation (see for example [10][26][64][71]).

2.0 Formal Underpinnings

In most graphical programming environments, the nodes of the graph can be viewed as processes that run concurrently and exchange data over the arcs of the graph. However, these processes and their interaction are usually much more constrained than those of CSP [37] or SCCS [58]. A better (and fortunately much simpler) formal underpinning is the Kahn process network [43].

2.1 Kahn Process Networks

In a process network, concurrent processes communicate only through one-way FIFO channels with unbounded capacity. Each channel carries a possibly infinite *sequence* (a *stream*) that we denote $X = [x_1, x_2, ...]$, where each x_i is an atomic data object, or *token*. Each token is written (produced) exactly once, and read (consumed) exactly once. Writes to the channels are *non-blocking* (they always succeed immediately), but reads are *blocking*. This means that a process that attempts to read from an empty input channel stalls until the buffer has sufficient tokens to satisfy the read. Lest the reader protest, I will show that this model of computation does not actually require either multitasking or parallelism, although it is certainly capable of exploiting both. It also usually does not require infinite queues, and indeed can be much more efficient in its use of memory than comparable methods in functional languages, as we will see.

A process in the Kahn model is a mapping from one or more input sequences to one or more output sequences. The process is usually constrained to be *continuous* in a rather technical sense. To develop this idea, we need a little notation.

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, $[x_1, x_2] \sqsubseteq [x_1, x_2, x_3]$. Consider a (possibly infinite) ordered set of sequences $\chi = \{ X_0 \sqsubseteq X_1 \sqsubseteq ... \}$. Such an ordered set of sequences can have one or more upper bounds Y, where $X_i \sqsubseteq Y$ for all $X_i \in \chi$. The *least upper bound* X_{LUB} of χ is an upper bound such for any other upper bound Y, $X_{LUB} \sqsubseteq Y$. To ensure that any ordered set of sequences always has an upper bound, we include in our algebra the ficti-

tious sequence $\top$ (*top*) defined so that $X \sqsubseteq \top$ for all sequences X. The empty sequence is denoted $\bot$ (*bottom*), and is obviously a prefix of any other sequence.

A *process* f maps an input sequence into an output sequence. Given an ordered set of sequences χ, it will map this set into another (possibly ordered) set of sequences Ψ. Let $\sqcap \chi$ denote the least upper bound (in the prefix order sense) of the set χ. Then a process f is said to be *continuous* if for all such sets χ,

$$f(\sqcap \chi) = \sqcap f(\chi). \tag{1}$$

This is analogous to the conventional notion of continuity for conventional functions, if the least upper bound is interpreted as a limit, as in

$$\sqcap \chi = \sqcap \{ X_0 \sqsubseteq X_1 \sqsubseteq \ldots \} = \lim_{i \to \infty} X_i \, . \tag{2}$$

Kahn sketches a proof that networks of continuous processes have a more intuitive property called *monotonicity* [43]. A process f is monotone if given two sequences X and X', then $X \sqsubseteq X' \Rightarrow f(X) \sqsubseteq f(X')$. This can be thought of as a form of causality, but one that does not invoke time. Moreover, in signal processing, it provides a useful abstract analog to causality that works for multirate discrete-time systems without requiring the invocation of continuous time.

For completeness, I now prove Kahn's claim that a continuous process is monotonic [43]. To do this, I prove that if a process is not monotonic, then it cannot be continuous. If the process f is not monotonic, then there exist sequences X and X' where $X \sqsubseteq X'$, but $f(X) \not\sqsubseteq f(X')$. Let $\chi = \{ X_0 \sqsubseteq X_1 \sqsubseteq \ldots \}$ be any prefix ordered sequence such that $X_0 = X$ and $\sqcap \chi = X'$. Then note that $f(\sqcap \chi) = f(X')$. But this cannot be equal to $\sqcap f(\chi)$ because $X \in \chi$ and $f(X) \not\sqsubseteq f(X')$. This concludes the proof.

A key consequence of these properties is that a process can be computed iteratively [54]. This means that given a prefix of the final input sequences, it is possible to compute part of the output sequences. In other words, a monotonic process is non-strict (its inputs need not be complete before it can begin computation). In addition, a continuous process will not wait forever before producing an output (i.e., it will not wait for completion of an infinite input sequence).

A network of processes is, in essence, a set of simultaneous relations between sequences. Any set of sequences that forms a solution is called a *fixed point*. Kahn argues in [43] that continuity of the processes implies that there will be exactly one "minimal" fixed point (where minimal is in the sense of prefix ordering). The minimal solution is the solution resulting from null sequences at the system inputs. Other solutions can then be found from this one by iterative computation, which works because of the monotone condition.

Note that continuity implies monotonicity, but not the other way around. One process that is monotone but not continuous is given by

$$f(X) = \begin{cases} \{0\}; & \text{if } X \text{ is finite} \\ \{0, 1\}; & \text{otherwise} \end{cases} \tag{3}$$

To show that this is monotone, note that if X is infinite and $X \sqsubseteq X'$, then $X = X'$, so

$$Y = f(X) \sqsubseteq Y' = f(X') \ . \tag{4}$$

If X is finite, then $Y = f(X) = \{0\}$, which is a prefix of all possible outputs. To show that it is not continuous, consider the sequence

$$\chi = \{ X_0 \sqsubseteq X_1 \sqsubseteq \ldots \}, \tag{5}$$

where each X_i has exactly i elements in it. Then $\sqcap \chi$ is infinite, so

$$f(\sqcap \chi) = \{0,1\} \neq \sqcap f(\chi) = \{0\}. \tag{6}$$

Iterative computation of this function is clearly problematic.

A useful property is that a network of monotone processes itself defines a monotone process. This property is valid even for process networks with feedback loops, as is formally proven using induction by Panagaden and Shanbhogue [62]. It is possible to use this to show that networks of monotone processes are *determinate*.

2.2 Nondeterminism

A useful property in some modern languages is an ability to express nondeterminism. This can be used to construct programs that respond to unpredictable sequences of events, or to build incomplete programs, deferring portions of the specification until more complete information

about the system implementation is available. Although this capability can be extremely valuable, it needs to be balanced against the observation that for the vast majority of programming tasks, programmers need determinism. Unfortunately, by allowing too much freedom in the interaction between nodes, some graphical programming environments can surprise the user with nondeterminate behavior. Nondeterminate operations can be a powerful programming tool, but they should be used only when such a powerful programming tool is necessary.

Taking a Bayesian perspective, a system is random if the information *known* about the system and its inputs is not sufficient to determine its outputs. The semantics of the programming language may determine what is known, since some properties of the execution may be unspecified. However, since most graphical programming environments do not define complete languages, it is easy (and dangerous) to circumvent what semantics there are by using the host language. In fact, the common principle of avoiding overspecifying programs leaves aspects of the execution unspecified, and hence opens the door to nondeterminate behavior. Any behavior that depends on these unspecified aspects will be nondeterminate.

For example, in the process network shown in figure 1, nothing in the graph specifies the relative timing of the processing in nodes B and C. Suppose that nodes B and C each modify a variable that they share. Then the order in which they access this variable could certainly affect the outcome of the program. The problem here is that the process network semantics, which specify a communication mechanism, have been circumvented using a shared variable in the host language. While this may be a powerful and useful capability, it should be used with caution, and in particular, it should not surprise the unwary programmer. Such a capability has been built into the PGM specification [45] in the form of what are called "graph variables."

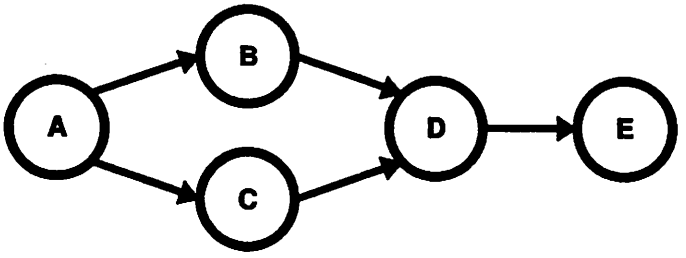


Figure 1. This process network does not specify the relative timing of the processing in nodes B and C. If D is a nondeterminate merge, it does not specify in which order the results should appear at E.

If B and C share a variable as described above, then they are possibly not monotone. Knowing that $f(X_0) = Y_0$, $f(X_1) = Y_1$, and $X_0 \sqsubseteq X_1$ is not enough to conclude that $Y_0 \sqsubseteq Y_1$ because the extended inputs might somehow affect the order in which the shared variable is accessed. However, they could be monotone if, for example, the discipline used to access the shared variable is equivalent to implementing a Kahn channel.

As a rather different example, suppose that actor D in figure 1 is a *nondeterminate merge*. (any of the three variants discussed by Panagaden and Shanbhogue [62]). Its behavior is that if a data value (a *token*) is available on either input, it can immediately move that token to its output. Now, the output depends on the order in which B and C produce their outputs, and on the timing with which D examines its inputs. It is easy to show that a nondeterminate merge is not monotonic, and hence not continuous.

Arvind and Brock [4] argue that the nondeterminate merge is practically useful for resource management problems. A resource manager accepts requests for a resource (e.g. money in a bank balance), arbitrates between multiple requests, and returns a grant or deny, or some related data value. It is observed that such a resource manager can be used to build a memory cell, precisely the type of resource that functional programming is trying to get away from.

A network with a nondeterminate merge clearly might be nondeterminate, but it might also be determinate. For example, suppose that C in figure 1 never actually produces any outputs. Then the nondeterminate merge in D will not make the network nondeterminate.

The nondeterminate merge does not satisfy one of Kahn's conditions for a process network, that reads from channels be blocking. This constraint makes it impossible for a process to test an input for the presence of data. Thus, if D is a nondeterminate merge, then the graph in figure 1 is not, strictly speaking, a Kahn process network.

We have been using the term "determinate" loosely. If we now formally define determinism in the context of process networks, then the main result of this section follows immediately. Define the *history* of a channel to be the sequence of tokens that have traversed the channel (i.e. have been both written and read). A Kahn process is said to be *determinate* if the histories of all the internal and output channels depend only on the histories of the input channels. A monotone

process is clearly determinate. Since a network of monotone processes is monotone [62], then a network of monotone processes is also determinate.

2.3 Streams

The graphical programming environments that we are concerned with are most often used to design or simulate real-time signal processing systems. Real-time signal processing systems are reactive, in that they respond to a continual stream of stimuli from an environment with which they cannot synchronize [8]. Skillcorn [75] argues that streams and functions on them are a natural way to model reactive systems. Streams are such a good model for signals that the signal processing community routinely uses them even for non-real-time systems.

Wendelborn and Garsden [78] observe that there are different ideas in the literature of what a stream is. One camp defines streams recursively, using cons-like list constructors, and treats them functionally using lazy semantics. This view is apparently originally due to Landin [47]. Lazy semantics ensure that the entire stream need not be produced before its consumer operates on it. For example, Burge [16] describes streams as the functional analog of coroutines that "may be considered to be a particular method of representing a list in which the creation of each list element is delayed until it is actually needed." As another example, in Scheme, streams are typically implemented as a two-element cell where one element has the value of the head of the stream and the other has the procedure that computes the rest of the stream [1]. Recursive operations on streams require use of a special "delay" operator that defers the recursive call until access to the "cdr" of the stream element is attempted. This ad-hoc mechanism makes recursive streams possible in a language without lazy semantics.

Another camp sees streams as channels, just like the channels in a Kahn process network. A channel is not functional, because it is modified by appending new elements to it. Kahn and MacQueen outline in [44] a demand-driven multitasking mechanism for implementing such channels. Ida and Tanaka argue for the channel model for streams, observing that it algorithmically transforms programs from a recursive to an iterative form [40]. Dennis, by contrast, argues for the recursive-cons representation of streams in Sisal 2 for program representation, but suggests translating them into non-recursive dataflow implementations using the channel model [25]. Franco, *et*

al. also argue in [28] for using the channel model, with a demand-driven execution style, and propose an implementation in Scheme. The channels are implemented using a "call with current continuation" mechanism in Scheme. This mechanism essentially supports process suspension and resumption, although the authors admit that at the time of their writing, no Scheme implementation supported this without the considerable expense of a control-stack copy.

A unique approach implemented in the language Silage [35] blends the benefits of a declarative style with the simplicity the channel model. In Silage, a symbol "x" represents an infinite stream. The language has the notion of a global cycle, and a simple reference to a symbol "x" can be thought of as referring to the "current value" of the stream x. An implicit infinite iteration surrounds every program. This language is being used successfully for both software and hardware synthesis in the Mentor Graphics DSP Station, the Cathedral project at IMEC [22], and in the Hyper project at U. C. Berkeley [66].

A more general approach is to associate with each stream a "clock," as done in Lustre [31] and Signal [9]. A clock is a logical signal that defines the alignment of stream tokens in different streams. For example, one could have a stream y where only every second token in y aligns with a token in another stream x. Although both streams may be infinite, one can view x as having twice as many tokens as y. A powerful algebraic methodology has been developed to reason about relationships between clocks, particularly for the Signal language [9][52].

I prefer the channel model for streams for a number of reasons. Stylistically, unlike the recursive-cons model, it puts equal emphasis on destruction (consumption of data from the stream) as construction (production of data onto the stream). Moreover, it does not require costly lazy evaluation. While a demand-driven style of control is popular among theoreticians, no established signal processing programming environment uses it, partly because of the cost, and partly because the same benefits (avoiding unnecessary computation) can usually be obtained more efficiently through compile-time analysis [12][50]. Unlike Silage, Lustre, and Signal, there is no concept of simultaneity of tokens (tokens in different streams lining up). Instead, tokens are queued using a FIFO discipline.

It is especially important in signal processing applications to recognize that streams can carry truly vast amounts of data. A real-time digital audio stream, for instance, might carry 44,100 samples per second per channel, and might run for hours. Video sequences carry much more. Viewing a stream as a conventional data structure, therefore, gets troublesome very quickly. It may require storing forever all of the data that ever enters the stream. Any practical implementation must instead store only a sliding window into the stream, preferably a small window. But just by providing a construct for random access of elements of a stream, for example, the language designer can make it difficult or impossible for a compiler to bound the size of the window.

A useful stream model in this context must be as good at losing data (and recycling its memory) as it is at storing data. The prefix-ordered sequences carried by the channels in the Kahn process networks are an excellent model for streams because the blocking reads remove data from the stream. However, special care is still required if the memory requirements of the channels in a network are to remain bounded. This problem will be elaborated below.

In [68][69][70], Reekie *et al.* consider the problem of supporting streams in the functional programming language Haskell [38]. They propose some interesting extensions to the language, and motivate them with a convincing discussion of the information needed by a compiler to efficiently implement streams. To do this, they use the Kahn process network model for Haskell programs, and class them into *static* and *dynamic*. In static networks, all streams are infinite. In dynamic networks, streams can come and go, and hence the structure of the network can change. Mechanisms for dealing with these two types of networks are different. Static networks are much more common in signal processing, and fortunately much easier to implement efficiently, although I will consider both types below.

For efficiency, Reekie *et al.* wish to evaluate the process networks eagerly, rather than lazily as normally required by Haskell [70]. They propose eager evaluation whenever strictness analysis [39] reveals that a stream is "head strict", meaning that every element in the stream will be evaluated. This is similar to the optimization embodied in the Eazyflow execution model for dataflow graphs, which combines data-driven and demand-driven evaluation of operator nets by

partitioning the net into subnets that can be evaluated eagerly without causing any wasteful computation [42].This, in effect, translates the recursive-cons view of streams into a channel view.

Reekie, *et al.* also point out that if analysis reveals that a subgraph is synchronous (in the sense of "synchronous dataflow" [50][51]), then very efficient evaluation is possible. While this latter observation has been known for some time in signal processing circles, putting it into the context of functional programming has been a valuable contribution. To clarify this point, I can establish a clear relationship between dataflow, functional languages, and Kahn process networks.

Streams can be generalized to higher dimensionality, as done in Lucid [75] and Ptolemy [49][19]. This, however, is beyond the scope of this paper.

2.4 Dataflow, functional languages, and process networks

A dataflow *actor*, when it fires, maps input tokens into output tokens. Thus, an actor, applied to one or more streams, will fire repeatedly. A set of *firing rules* specify when an actor can fire. Specifically, these rules dictate precisely what tokens must be available at the inputs for the actor to fire. A firing *consumes* input tokens and *produces* output tokens. A sequence of such firings is a particular type of Kahn process that we might call a *dataflow process.*

More specialized dataflow models, such as Dennis' static dataflow [24] or synchronous dataflow [50][51] can be described in terms of dataflow processes. The models used by all signal processing environments mentioned above can also be described in terms of dataflow processes. The tagged token model of Arvind and Gostelow [5][6] is related, but not identical, as I will show. Signal [9] and Lustre [31], which are called "synchronous dataflow languages," do not form dataflow processes at all because they lack the FIFO queues of the communication channels.

A sufficient condition for a dataflow process to be continuous is that the actors are *functional*, and that the set of firing rules is *sequential*. "Functional" means that the actors lack side effects and that the outputs are purely a function of the inputs. "Sequential" means that the firing rules can be tested in a pre-defined order using only blocking reads. A little notation will help make this rather technical definition precise.

2.4.1 Firing rules

An actor with M input streams can have N firing rules

$$F = \{F_1, F_2, ..., F_N\} . \tag{7}$$

The actor can fire if and only if at least one of the firing rules is satisfied, where each firing rule constitutes a set of *patterns,* one for each of M inputs,

$$F_i = \{P_{i,1}, P_{i,2}, ..., P_{i,M}\} . \tag{8}$$

A pattern $P_{i,j}$ is a (typically finite) sequence. For firing rule i to be satisfied, each pattern $P_{i,j}$ must form a prefix of the sequence of unconsumed tokens at input j .

For some firing rules, some patterns might be empty lists, $P_{i,j} = \perp$. This means that any available sequence at input j is acceptable. In particular, it does *not* mean that input j must be empty.

To be able accommodate the usual dataflow firing rules, we need a slight generalization of prefix ordering algebra. The special symbol "*" will denote a token wildcard. Thus, the sequence [*] is a prefix of any sequence with at least one token. The sequence [*,*] is a prefix of any sequence with at least two tokens. The only sequence that is a prefix of [*] is $\perp$, however. Notice therefore, that the statement $[*] \sqsubseteq P$ is *not* saying that any one-token sequence is a prefix of P. All it says is that P has at least one token.

Let A_j, for $j = 1, ..., M$, denote the list of available unconsumed tokens on the j^{th} input. Then the firing rule F_i is enabled if

$$P_{i,j} \sqsubseteq A_j, \text{ for all } j = 1, ..., M . \tag{9}$$

We can write condition (9) using the shorthand

$$F_j \sqsubseteq A \tag{10}$$

where $A = \{A_1, A_2, ..., A_M\}$, and it is understood that each sequence in the list F_j is a prefix of the corresponding sequence in the sequence of available tokens A.

For many actors, the firing rules are very simple. Consider an adder with two inputs. It has a single firing rule, $F = \{ \{ [*], [*] \} \}$, meaning that each of the two inputs must have at least one token. More generally, synchronous dataflow actors [50][51], always have a single firing rule, and each pattern in the firing rule is of the form $[*, *, ..., *]$, with some fixed number of wild-cards. In other words, an SDF actor is enabled by a fixed number of tokens at each input.[1]

A more interesting actor is the select actor in figure 2a, has the firing rules

$$F_1 = \{ [*], \bot, [T] \} \tag{11}$$

$$F_2 = \{ \bot, [*], [F] \} \tag{12}$$

where T and F match true and false-valued Booleans, respectively. The behavior of this actor is to read a Boolean control input, then read a token from the specified data input and copy that token to the output. The firing rules are sequential, in that a blocking read of the control input, followed by a blocking read of the appropriate data input, will invoke the appropriate firing rule.

The nondeterminate merge with two inputs, also shown in figure 2b, has the firing rules

$$F_1 = \{ [*], \bot \} \tag{13}$$

$$F_2 = \{ \bot, [*] \} . \tag{14}$$

These rules are not sequential. A blocking read of either input fails to produce the desired behavior, as illustrated in figure 3. In figure 3a, a blocking read of the top input will never unblock. In figure 3b, a blocking read of the bottom input will never unblock. In both cases, the behavior is

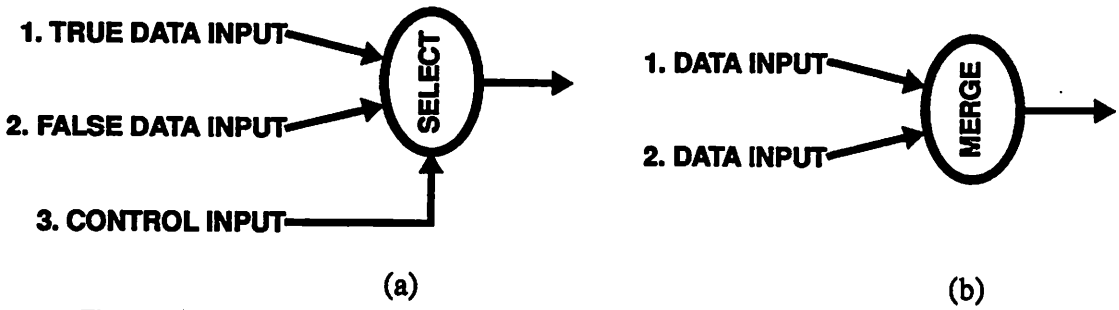


Figure 2. The select and nondeterminate merge actors each combine two data streams into one, but the select actor uses a Boolean control signal to determine how to accomplish the merge.

1. An SDF actor also produces a fixed number of tokens when it fires, but this is not captured in the firing rules.

incorrect. Note that with any correct implementation of the nondeterminate merge, both networks in figure 3 are nondeterminate. It is unspecified how many times a given token will circulate around the feedback loop between arrivals of tokens from the left.

2.4.2 Identifying sequential firing rules

In general, a set of firing rules is sequential if the following procedure succeeds:

1. Find an input j such that $[*] \sqsubseteq P_{i,j}$ for all $i = 1, ..., N$. That is, find an input such that all the firing rules require at least one token from that input. If no such input exists, fail.
2. For the choice of input j, divide the firing rules into subsets, one for each specific token value mentioned in the first position of $P_{i,j}$ for any $i = 1, ..., N$. If $P_{i,j} = [*, ...]$, then the firing rule F_i should appear in all such subsets.
3. Remove the first element of $P_{i,j}$ for all $i = 1, ..., N$.
4. If all subsets have empty firing rules, then succeed. Otherwise, repeat these four steps for any subset with any non-empty firing rules.

The first step identifies an input where a token is required by all firing rules. The idea of the second step is that reading a token from that particular input will often at least partially determine which firing rules apply. Observing its value, therefore, will often reduce the size of the set of applicable firing rules.

Consider the select actor in figure 2. The above steps become

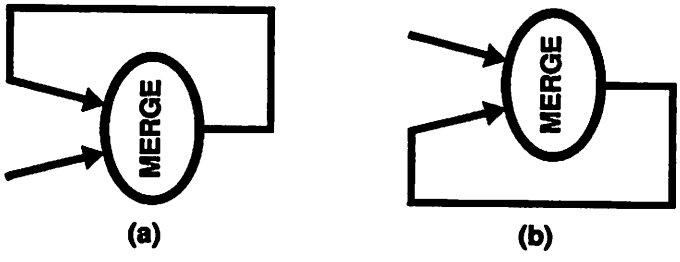


Figure 3. Illustration that the firing rules of the nondeterminate merge are not sequential. A blocking read of either input will cause one of these two networks to deadlock inappropriately.

1. $j = 3$.
2. The firing rules divide into two sets, $\{F_1\}$ and $\{F_2\}$, each with only one rule.
3. The new firing rules become $F_1 = \{[*], \perp, \perp\}$ in the first subset and $F_2 = \{\perp, [*], \perp\}$ in the second subset.
4. The procedure repeats trivially for each subset, and in step 3, the modified firing rules become empty.

For the nondeterminate merge, the procedure fails immediately, in the first application of step 1.

2.4.3 Relationship to higher-order functions

Constraining the actors to be functional makes a dataflow process roughly equivalent to the function "maps" used by Burge [16] and Reekie [68]. It is similar to the "map" function in Haskell and the "mapcar" function in Lisp, except that it introduces the notion of consuming the tokens that match the firing rule, and hence easily deals with infinite streams.

All of these variants of "map" are *higher-order functions*, in that they take functions as arguments and return functions [55]. For example, *maps f*, where *f* is a function, returns a function that applies *f* to each element of a stream. The function *f* might take a scalar argument, but the function returned by *maps f* takes a stream argument. Thus, *maps f* is a dataflow process, where each firing consists of one application of *f*.

2.4.4 A nondeterminate example

An example that combines many of the points made so far can be constructed using the nondeterminate operator introduced by McCarthy [56] and used by Hudak [38]:

amb $(x, \perp) = x$

amb $(\perp, y) = y$

amb $(x, y) = x$ or y chosen randomly

These three declarations define the output of the *amb* function under three firing rules. The symbols "*x*" or "*y*" translate into a firing rule that requires one data token, and the value of the data is given symbol *x* or *y*. A dataflow process could be constructed by repeatedly firing this function on

stream inputs. McCarthy points out that the expression *amb* (1, 2) + *amb* (1, 2) could take on the value 3, and uses this to argue that nondeterminism implies a loss of referential transparency[1].

When used to create a dataflow process, this example actually mixes two distinct causes for nondeterminism. Random behavior in an actor acting alone is sufficient to lose determinacy and referential transparency. The simpler definition:

amb (x, y) = x or y chosen randomly

is sufficient for *amb* (1, 2) + *amb* (1, 2) to take on the value 3. If the choice of random number is made using a random number generator, then normally the random number generator has state, initialized by a seed. Perhaps the seed should be shown explicitly as an argument to the function:

amb (x, y, s) = x or y chosen by generating a random number from seed s.

Suddenly, we regain referential transparency and determinacy. It would not be possible for *amb* (1, 2, 3) + *amb* (1, 2, 3) to equal 3, for example. Without giving the seed as an argument, *amb* is not functional.

Consider the simplified definition:

amb2 $(x, \perp) = x$

amb2 $(\perp, y) = y$

amb2 $(x, y) = y$

This definition has no random numbers in it, but in a dataflow process network, it is still possible for *amb2* (1, 2) + *amb2* (1, 2) to equal 3. The firing rules are not sequential. The output depends on how the choice between firing rules is made, something not specified by the language semantics.

We can show directly that a dataflow process constructed with the *amb2* function is not monotonic, and hence is not continuous. Let $f(X, Y)$ represent the dataflow process made with actor "*amb2*" applied to sequences X and Y. It is easy to show that the process is not monotonic, and hence is not continuous. Consider the sequences

$$X_1 = [1], X_2 = [1, 1], \text{and } Y_1 = \perp, Y_2 = [2], \tag{15}$$

1. A basic notion dating back to the lambda calculus [20], referential transparency means that any two identical expressions have identical values. If *amb*(1,2)+*amb*(1,2)=3, then clearly the two instances of *amb*(1,2) cannot have taken on the same value.

where Y_1 is the empty sequence. Clearly, $X_1 \sqsubseteq X_2$ and $Y_1 \sqsubseteq Y_2$. However,

$$f(X_1, Y_1) \not\sqsubseteq f(X_2, Y_2) \ . \tag{16}$$

We get $f(X_1, Y_1) = [1]$, while $f(X_2, Y_2)$ can take on any of the following possible values: $[2, 1]$, $[1, 2]$, $[1, 2, 1]$, $[1, 1, 2]$, or $[2, 1, 1]$. This is clearly nondeterminate. Only three of the five possible outcomes satisfy the monotonicity constraint. And these choose rather arbitrarily from among the firing rules. If we were to make a policy of these choices, it would be easy to construct other example inputs that would violate monotonicity.

One might argue for a different interpretation of the firing rules, in which a $\perp$ in a firing rule pattern matches only an empty input (no tokens available). Under this interpretation, we get $f(X_1, Y_1) = [1]$ and $f(X_2, Y_2) = [2, 1]$. While not monotonic, this might appear to be determinate (recall that we've only argued that continuity is *sufficient* for determinacy, not that it is *necessary*). But further examination reveals that I have made some implicit assumptions about synchronization between the input streams. To see this, consider the prefix ordered sequences

$$X_1 = [1] \ , X_2 = [1] \ , X_3 = [1, 1] \ , \text{and } Y_1 = \perp, Y_2 = [2] \ , Y_3 = [2] \ . \tag{17}$$

It would seem reasonable to argue that these are in fact exactly the same sequences as in (15). We are just looking at the value of the sequences more often. However, under the same implicit synchronization assumptions, the output is different:

$$f(X_1, Y_1) = [1] \ , f(X_2, Y_2) = [2] \ , f(X_3, Y_3) = [2, 1] \ . \tag{18}$$

These outputs are not prefix ordered, as they would be for a continuous process.

This issue becomes much clearer if one considers a more complete dataflow process network, as shown in figure 4. The dataflow processes A and B have no inputs, so their firing rule is

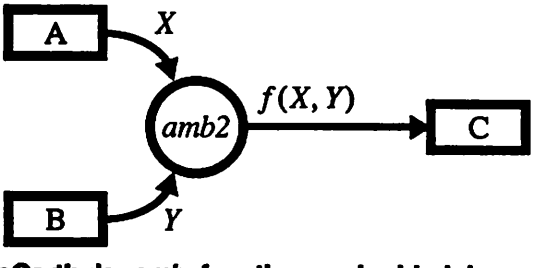


Figure 4. A variant of McCarthy's *amb* function embedded in a dataflow process network.

simple; they are always enabled. They produce at their outputs the streams X and Y. The problems addressed above, in this context, refer to the relative timing of token production at A and B compared to the timing of the firings of the *amb2* actor. In dataflow process network semantics, this timing is not specified.

2.4.5 Firing rules and template matching

Some functional languages use template matching in function definitions the way I have been using firing rules. Consider the following Haskell example (with slightly simplified syntax):

fac 0 = 1

fac n = n*fac(n-1)

This defines a factorial function. If the argument is 0, the result is 1. If the argument is n, the result is n*fac(n-1). These are not ambiguous because the semantics of Haskell gives priority to the first template, removing any ambiguity. The second template is really a shorthand for "any n except 0." These two templates, therefore, viewed as firing rules, are naturally sequential, since each rule consumes one token and implicitly states: "use me if no previously declared firing rule applies and the inputs match my pattern." Of course, this does not remove ambiguities due to function arguments where no data is needed. (Haskell has lazy semantics, deferring the evaluation of function arguments until the data is needed, so a function may be invoked that will decide it does not need data from one its arguments).

Embedding this example, the factorial function, in a dataflow process network introduces new and interesting problems. Consider *fac X*, where X is a stream. Each firing of the actor can trigger the creation of new streams, so this process network is not static. I will consider more interesting recursive examples than this is considerable detail below, so I defer further discussion.

2.4.6 Relationship to Kahn Process Networks

Dataflow process networks are a special case of Kahn process networks. They construct a process as a sequence of atomic actor invocations. Instead of suspending a process on a blocking read or non-blocking write, threads can be freely interleaved by a *scheduler*, which determines the sequence of actor firings. Since the actors are functional, no state needs to be stored when one

actor terminates and another fires. The biggest advantage, therefore, is that the context switch overhead of process suspension and resumption is entirely avoided.

The offsetting cost is the cost of scheduling. However, for most programs, this cost can be entirely shifted to the compiler [50] [12]. While it is impossible to always shift all costs to the compiler [12], large clusters within a process network can be scheduled at compile time, greatly reducing the number of dataflow processes that must be dynamically scheduled. As a consequence of this efficiency, much finer granularity is practical, with processes often being as simple as to just add two streams. We will now consider execution models in more detail.

2.5 Execution models

Given a dataflow process network, a surprising variety of execution models can be associated with it. This variety is due, in no small part, to the fact that a dataflow process network does not overspecify an algorithm the way non-declarative semantics do. Execution models have different strengths and weaknesses, and there is, to date, no clear winner.

2.5.1 Concurrent processes

Kahn and MacQueen propose an implementation of Kahn process networks using multitasking with a primarily demand-driven style [44]. A single "driver" process (one with no outputs) demands inputs. When it suspends due to an input being unavailable, the input channel is marked "hungry" and the source process is activated. It may in turn suspend, if its inputs are not available. Any process that issues a "put" command to a hungry channel will be suspended and the destination process restarted where it left off, thus injecting also a data-driven phase to the computation. If a "get" operation suspends a process, and the source process is already suspended waiting for an input, then deadlock has been detected.

In the Kahn and MacQueen schema, configuration of the network on the fly is allowed. This allows for recursive definition of processes. Recursive definition of streams (data) is also permitted in the form of directed loops in the process graph.

The repeated task suspension and resumption in this style of execution is relatively expensive, since it requires a context switch. It suggests that the granularity of the processes should be

relatively large. For dataflow process networks, the cost can be much lower, and hence the granularity can be smaller.

2.5.2 Dynamic scheduling of dataflow process networks

Dataflow process networks have other natural execution models due to the breakdown of a process into a sequence of actor firings. A firing of an actor provides a different quantum of execution than a process that suspends on blocking reads. Using this quantum avoids the complexities of task management (context switching and scheduling) that are implied by Kahn and MacQueen [44] and explicitly described by Franco, *et al.* [28]. Instead of context switching, dataflow process networks are executed by scheduling the actor firings. This scheduling can be done at compile time or at run time, and in the latter case, can be done by hardware or by software.

The most widely known execution models for dataflow process networks have emerged from research into computer architectures for executing dataflow graphs [3]. This association may be unfortunate, since the performance of such architectures has yet to prove competitive [34]. In such architectures, actors are fine-grained, and scheduling is done by hardware. Although there have been some attempts to apply these architectures to signal processing [61], the widely used dataflow programming environments for signal processing have nothing to do with dataflow architectures.

Some signal processing environments, for example COSSAP from Cadis (now Synopsys) and the dynamic dataflow domain in Ptolemy, use a run-time scheduler implemented in software. This performs essentially the same function performed in hardware by dataflow machines, but is usually used with actors that have larger granularity. The scheduler tracks the availability of tokens on the inputs to the actors, and fires actors that are enabled.

2.5.3 Static scheduling of dataflow process networks

For many signal processing applications, the firing sequence can be determined statically (at compile-time). The class of dataflow process networks for which this is always possible is called *synchronous dataflow* [46][50][51]. In synchronous dataflow, the solution to a set of *balance equations* relating the production and consumption of tokens gives the relative firing rates of

the actors. These relative firing rates combined with simple precedence analysis allows for the static construction of periodic schedules. Synchronous dataflow is used in COSSAP (for code generation, not for simulation), in the multirate version of SPW from the Alta Group of Cadence (formerly Comdisco), and in the synchronous dataflow domain in Ptolemy. These methods have recently been extended to cover most dynamic dataflow graphs [12][48], and have been implement in the Boolean dataflow domain in Ptolemy. For fully general dataflow models, it is still necessary to have some responsibilities deferred to a runtime scheduler [12].

2.5.4 The tagged-token model

An execution model developed by Arvind and Gostelow [5][6] actually generalizes the dataflow process network model. In this model, each token has a tag associated with it, and firing of actors is enabled when inputs with matching tags are available. Outputs to a given stream are produced with distinct tags. An immediate consequence is that there is no need for a FIFO discipline in the channels. The tags keep track of the ordering. More importantly, there is no need for the tokens to be produced or consumed in order. The possibility for out-of-order execution allows us to construct dataflow graphs that would deadlock under the FIFO scheme but not under the tagged-token scheme. We will consider a detailed example below, after developing a usable language.

3.0 Experimenting with Language Design

The dataflow process network model, as defined so far, provides a framework within which we can define a language. To define a complete language, we would need to specify a set of primitive actors. Instead, I will outline a coordination language, leaving the design of the primitives somewhat arbitrary. There are often compelling reasons to leave the primitives unspecified. Many graphical dataflow environments rely on a host language for specification of these primitives, and allow arbitrary granularity and user extensibility. Depending on the design of these primitives, the language may or may not be functional, may or may not be able to express nondeterminism, and may or may not be as expressive other languages.

Granular Lucid, for example, is a coordination language with the semantics of Lucid [41]. Coordination languages with dataflow semantics are described by Suhler *et al.* [76], Gifford and Lucassen [29], Onanian [61], Printz [65], and Rasure and Williams [67]. Contrast these to the approach of Reekie [68] and the DSP Station from Mentor Graphics [26], where new actors are defined in a language with identical semantics to the visual language. There are compelling advantages to that approach, in that all compiler optimizations are available down to the level of the host language primitives. But the hybrid approach, in which the host language has imperative semantics, gives the user more flexibility. Since our purpose in this paper is to explore the data-flow process networks model fully, this flexibility is essential.

3.1 The Ptolemy system

To make the discussion concrete, I will use the Ptolemy software environment [15] to illustrate some of the tradeoffs. It is well suited for several reasons:

- It has both a visual ("block diagram") and a textual interface; the visual interface similar in principle to many of those used in signal processing software environments.
- It does not have any model of computation built into the kernel, and hence can be used to experiment with variations on the model of computation.
- Three dataflow process network "domains" have already been built in Ptolemy, precisely to carry out such experiments. I can use, compare, and extend these.
- The set of primitive actors is easily extended (using C++ as the host language). This gives us more than enough freedom to test the limits of the dataflow process networks model of computation.

A *domain* in Ptolemy is a user-defined subsystem implementing a particular model of computation. Three Ptolemy domains have been constructed with dataflow semantics, and one with more general process network semantics. The *synchronous dataflow* domain (SDF) [50][51] is particularly well suited to signal processing [14], where low-overhead execution is imperative. The SDF domain makes all scheduling decisions at compile time. The *dynamic dataflow* domain (DDF) makes all scheduling decisions at runtime, and is therefore much more flexible. The *Boolean*

dataflow domain (BDF) attempts to make scheduling decisions for dynamic dataflow graphs at compile time, using the so-called token-flow formalism [12][48]. It resorts to run-time scheduling only when its analysis techniques break down. The *communicating processes* domain (CP) uses a multitasking kernel to manage process suspension and resumption.

Ptolemy supports two distinct execution models, *interpreted* and *compiled*. Compilation can be implemented using a simple code generation mechanism, allowing for quick experimentation, or it can be implemented using more sophisticated transformation and optimization techniques. Such optimization may require more knowledge about the primitives than the simple code generation mechanism, which simply stitches together code fragments defining each actor.

3.2 Visual hierarchy — the analog to procedural abstraction

In keeping with the majority of signal processing programming environments, I will use a visual syntax for the interconnection of dataflow processes. In fact, in Ptolemy, a program is not entirely visual, since the actors and data structures are defined textually, using C++. Only the gross program structure is described visually. The visual equivalent of an expression, of course, is a subgraph. Subgraphs can be encapsulated into a single node, thus forming a larger dataflow process by composing smaller ones. This is analogous to procedural abstraction in imperative languages and functional abstraction in functional languages.

3.2.1 Determinacy and referential transparency

To make the dataflow process network determinate, as discussed above, it is sufficient for the actors to have two properties; their mappings from input tokens to output tokens should be functional (free from side effects), and the firing rules for each actor should be sequential, in the technical sense given above. If our actors have these properties, then our language has referential transparency, meaning that syntactically identical expressions have the same value regardless of their lexical position in the program.

With referential transparency, the two subgraphs shown in figure 5 are equivalent. The two inputs to the identical dataflow processes *A* are identical streams, so the outputs will be identical.

If the primitive actors are functional, then hierarchical actors may be functional as well, but there are some complications due to scheduling, directed loops in the graph, and *delays*.

3.2.2 Functional behavior and hierarchy

In modern languages, it is often considered important that abstractions be semantically little different from language primitives. Thus, if the primitive actors are functional, the hierarchical nodes should be functional. If the primitive actors have firing rules, then the hierarchical nodes should have firing rules. We will find this goal problematic.

A hierarchical node in a dataflow process network has a subnetwork and input/output ports, as shown in the examples in figure 6. If we wish for the subnetwork to fire as a unit, as if it

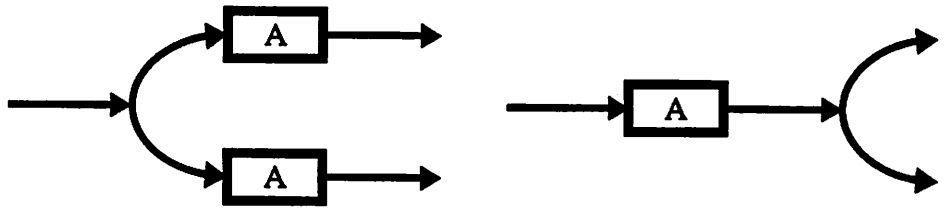


Figure 5. Referential transparency implies that these two dataflow process networks are equivalent.

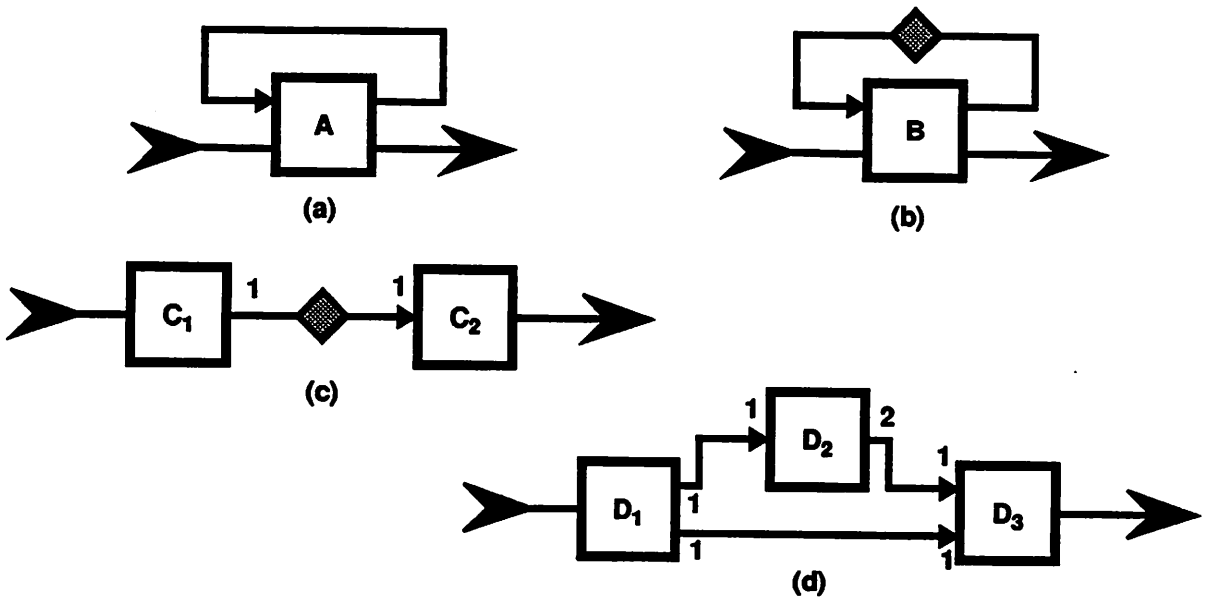


Figure 6. Hierarchical nodes in a dataflow process network may not be functional even if the primitives are functional. The large arrowheads indicate input and output for the hierarchical node.

were a primitive, then the firing must be free of side effects. This is not always possible. Consider the example in figure 6a. Note first that to avoid deadlock, it must be possible for actor *A* to fire (at least the first time) without any tokens at its top input port. Such a feedback loop will typically be used to implement a recurrence, in which case the feedback channel will store tokens from one firing of the hierarchical node for use in the next firing. With this usage, the hierarchical node has state, and is therefore not functional.

3.2.3 Delays, state, recurrences, and recursion

The hierarchical node shown in figure 6b is more typical. The shaded diamond is a *delay*, which is typically implemented as an initial token in the channel. This initial token enables the first firing of actor *A* if it requires a token on the top input. It is called a "delay" because for any channel with a unit delay, the n-th token read from the channel is the $(n-1)$-th written to it. A feedback loop with delay effectively stores state, making the hierarchical node non-functional.

The delay shown in figure 6(b) is typically implemented using the "cons" operator to initialize streams when streams are based on the recursive-cons model [47]. It is roughly equivalent to the "D" operator in the tagged-token model [6]. It is the visual equivalent of "fby" (followed-by) in Lucid [75] and the "->" operator in Lustre [32]. In the single assignment language Silage, developed for signal processing [35], a delay is written "x@1". This expression refers to the stream "x" delayed by one token, with the initial token value defined by a declaration like "x@@1 = *value*." For example,

```
x = 1 + x@1;
x@@1 = 0;
```

defines a stream consisting of all non-negative integers, in order.

In functional languages, instead of using a recurrent construct like a delay, state is usually carried in the program using recursion. Consider, for example, the following Haskell program, which adds the elements of a list or stream, given by Reekie [68]:

```
integrate xs = scanl (+) 0 xs
```

where *scanl* is a higher order function defined in Haskell as follows:

scanl (*f*, *init*, ⊥) = *init*

scanl (f, *init*, (x:xs)) = f (x, scanl (f, *init*, xs))

These two definitions use template matching; the first is invoked if the third argument is an empty list. The *init* gives the initial value for the sum, equivalent to the value of the initial token in a delay. The syntax (x:xs) divides a list into the first element (x) and the rest (xs). The program uses recursion to carry state, via the higher-order function *scan*. It has been observed that for efficiency this recursion must be translated into an iterative implementation [40][25][28]. For streams this is mandatory, since otherwise the depth of the recursion could become extremely large.

Delays in a hierarchical node can make the node non-functional even if it is not in a feedback loop. Consider the example in figure 6c. Following Lee and Messerschmitt [50], the "1" symbol next to the output of C_1 means that it produces one token when it fires. The "1" next to the input of C_2 means that it consumes one token when it fires. A reasonable firing of the hierarchical node would therefore consist of one firing of C_1 and one of C_2. But under this policy, state will have to be preserved on the arc connecting the two actors between firings, again making the hierarchical node non-functional.

3.2.4 Firing subgraphs — the balance equations

This last example raises the question of how to determine how many firings of the constituent actors make up a "reasonable" firing of a hierarchical node. One approach would be to solve the *balance equations* of [48][50][51] to determine how many firings of each actor are needed to return a subsystem to its original state. By "original state" we mean that the number of unconsumed tokens on each internal channel (arc) should be the same after the firing as before. For the example in figure 6c, the single balance equation is

$$r_{C_1} \times 1 = r_{C_2} \times 1, \tag{19}$$

where r_{C_i} is the number of firings of C_i that returns the subsystem to its original state (and thus keeps it "in balance"). For dynamic dataflow graphs, these balance equations are a bit more complicated, but often lead to definitive conclusions about the relative number of firings of the actors that are required to maintain balance.

Unfortunately, two problems arise. First, some useful systems have balance equations with no solution [12][13]. Such systems are said to be *inconsistent*, and generally have unbounded memory requirements. A simplified (and probably not useful) example is shown in figure 6d. The balance equations for this subsystem are (one for each arc)

$$r_{C_1} \times 1 = r_{C_2} \times 1, \tag{20}$$

$$r_{C_1} \times 1 = r_{C_3} \times 1, \tag{21}$$

$$r_{C_2} \times 2 = r_{C_3} \times 1. \tag{22}$$

These equations have no solution. Indeed, any set of firings of these actors will leave state in the subsystem, so no firing pattern would result in a functional hierarchical node.

To hint that inconsistent systems are, in fact, useful, consider an algorithm that computes an ordered sequence of integers of the form $2^a 3^b 5^c$ for all $a, b, c \geq 0$. This problem has been considered by Dijkstra [27] and Kahn and MacQueen [44]. A dataflow implementation equivalent to the first of two by Kahn and MacQueen is shown in figure 7a. The "merge" block is an ordered merge [48]; given a nondecreasing sequence of input values on two streams, it merges them into a single stream of nondecreasing values, and removes duplicates. An more efficient implementation that does not generate such duplications (and hence does not need to eliminate them) is given in figure 7b. It is also inconsistent. Neither of these implementations has bounded memory requirements.

3.2.5 Side effects and state

Some of the problems with directed loops could be solved by requiring all delays to appear only at the top level of the hierarchy, as was done for example in the BOSS system [72]. This is awkward, however, and anyway provides only a partial solution. A better solution is simply to reconcile the desire for functional behavior with the desire to maintain state. This can be done simultaneously for hierarchical nodes and primitives, greatly increasing the flexibility and convenience of the language, while still maintaining the desirable properties of functional behavior.

The basic observation is that internal state in a primitive or a hierarchical node is *syntactic sugar* (a convenient syntactic shorthand) for feedback loops at the top level of the graph. In other words, there is no reason to actually put all such feedback loops at the top level if semantics can be maintained with a more convenient syntax. With this observation, we can now allow actors with state. These become more like *objects* than *functions*, since they represent both data and methods for operating on the data. The (implicit) feedback loop around any actor or hierarchical node with state also establishes a precedence relationship between successive firings of the actor. This precedence serializes the actor firings, thus ensuring proper state updates.

Once we allow actors with state, it is a simple extension to allow actors with other side effects, such as those handling I/O. The inherently sequential nature of an actor that outputs a stream to a file, for example, is simply represented by a feedback loop that does not carry any meaningful data, but establishes precedences between successive firings of the actor.

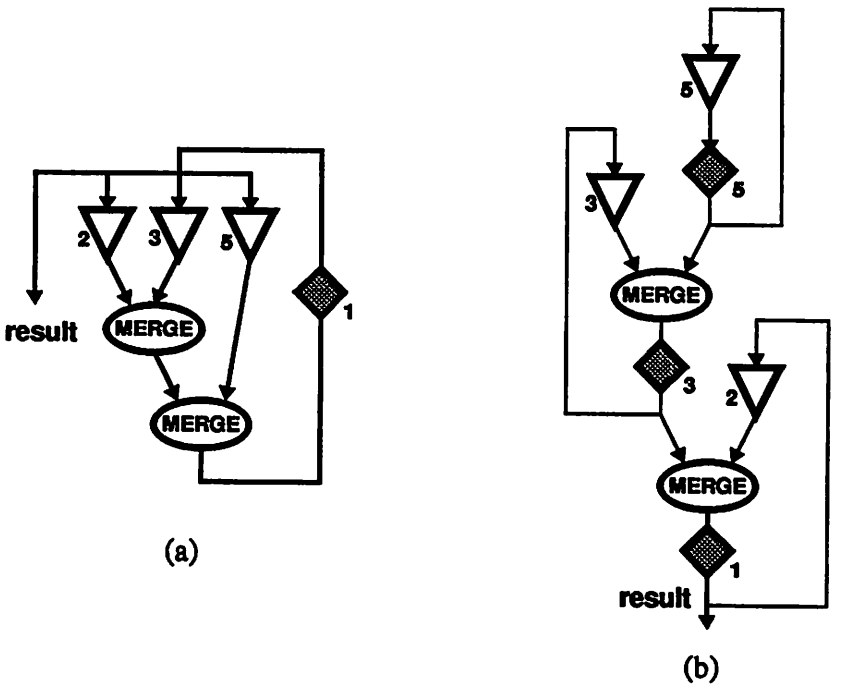


Figure 7. Two inconsistent dataflow graphs that compute an ordered sequence of integers of the form $2^a3^b5^c$. The triangular icons multiply their inputs by the indicated constant. The delay icon (a diamond) represents an initial token with value 1, 3, or 5, as annotated.

3.3 Function arguments — parameters and input streams

In Ptolemy, as in many software environments of this genre, there are three phases to the execution of a program. The *setup* phase makes a pass over the hierarchical program graph initializing delays, initializing state variables, evaluating *parameters*, evaluating whatever portion of the schedule is pre-computed, and performing whatever other setup functions the program modules require. The *run* phase involves executing either the pre-computed schedule or a dynamic schedule that is computed on-the-fly. If the run is finite (it often is not), there is a *wrapup* phase, in which allocated memory is freed, final results are presented to the user, and any other required cleanup code is executed.

The *parameters* that are evaluated during the setup phase are often related to one another via an expression language. Thus, parameters represent the part of the computation that does not operate on streams, in which values that might be used during stream processing are computed. Some simple examples are the gain values associated with the triangular icons in figure 7 or the initial values of the delays in the same figure. In principle, these values may be specified as arbitrarily complex expressions.

The gain blocks in figure 7 may be viewed as functions of arity two, the multiplying constant and the input stream. But unlike any functional language that I know of, a clear distinction is made between *parameter arguments* and *stream arguments*. This distinction is both syntactic and semantic. The syntax in Ptolemy is to use a textual expression language to specify the value of the parameters, using a parameter screen like that in figure 8. This expression language has some of the trappings of a standard programming languages, including types and scoping rules. It could be entirely replaced by a standard programming language, although preferably one with declarative semantics.

Parameters are still formally viewed as arguments to the function represented by the actor. But the syntactic distinction between parameters and stream arguments is especially convenient in visual programming. It avoids cluttering a diagrammatic program representation with a great many arcs representing streams that never change in value. Moreover, it makes the job of a compiler or interpreter simpler, removing the optimization step of identifying such static streams. In

Ptolemy, when compiled mode is used for implementation, code generation occurs *after* the parameters have been evaluated, thus allowing highly-optimized, application-specific code to be generated. For example, instead of a single telephone channel simulator subroutine capable of simulating any combination of impairments, a optimized code that takes advantage of the fact that the third harmonic distortion is set to zero (see figure 8) can be synthesized. This becomes particularly important when the implementation is via hardware synthesis, as is becoming increasingly common in signal processing systems.

Sometimes, all of the arguments to a function are parameters, in which case we call the actor a *source*, since it has no dynamic inputs (see, for example, the A and B actors in figure 4).

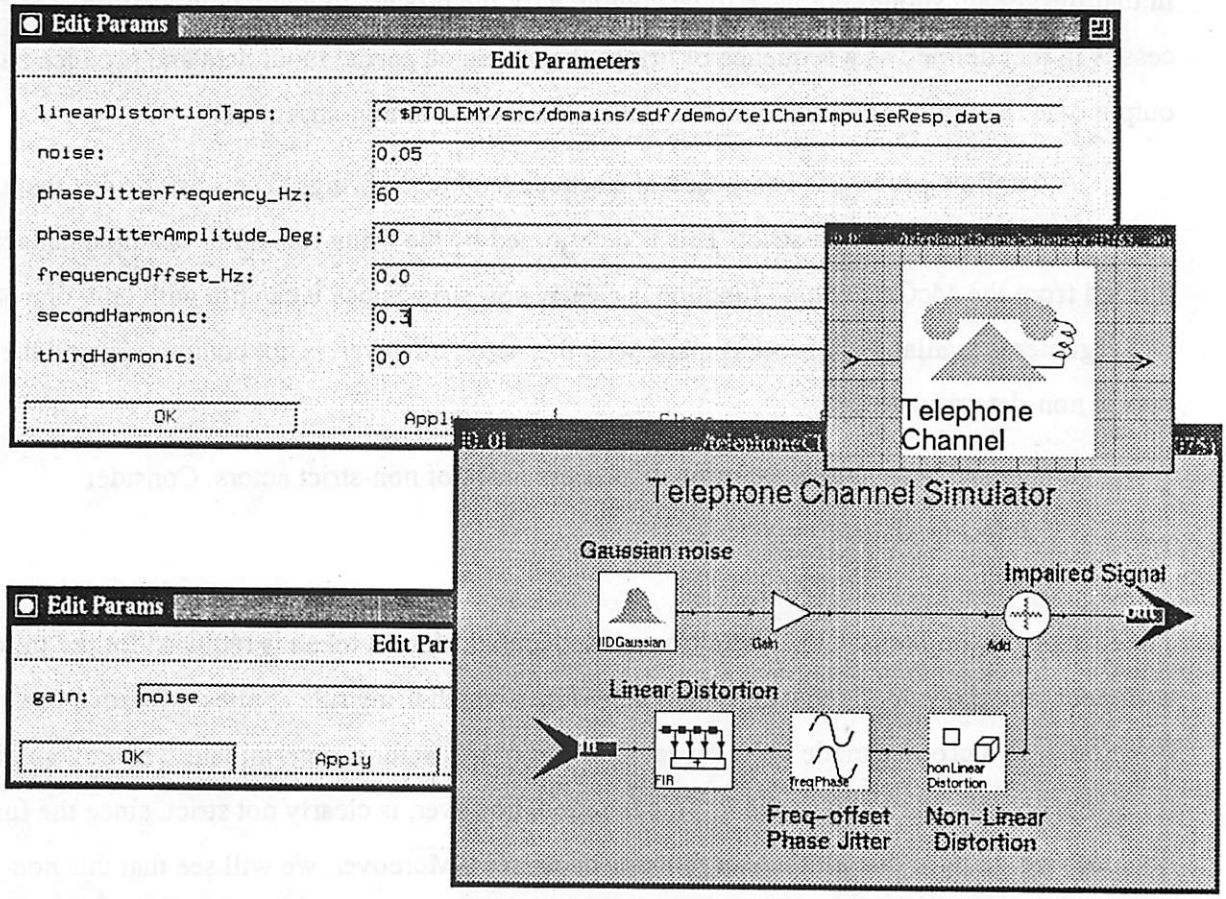


Figure 8. Top: A typical parameter screen in Ptolemy for a hierarchical node that models a telephone channel. The first parameter is given as a reference to a file. The icon for the node is shown to the right. The next level down in the hierarchy is shown in the lower right window. At the lower left, the parameter screen shows that the parameter for the Gain actor inherits its value from the "noise" parameter above it in the hierarchy. Parameter values can also be expressions.

Referential transparency for source actors is also preserved, as long as the parameters are considered. Thus, the transformation shown in figure 5 is now possible only if the actors or subgraphs being consolidated have identical parameters. Thus, with these syntactic devices (actors with state, delays, and actors with parameters as well as inputs), referential transparency is still possible. I call such actors *generalized functional actors.*

3.4 Firing rules and strictness

A function is strict if it requires that all its arguments be present before it can begin computation. A dataflow process, viewed as a function applied to a stream, clearly should not be strict, in that the stream should not have to be complete for the process to begin computation. The process is in fact defined as a sequence of firings that consume partial input data and produce partial output data. But in our context, this is a rather trivial form of non-strictness.

A dataflow process is composed of a sequence of actor firings. The actor firings themselves might be strict or non-strict. This is determined by the firing rules. For example, an actor formed from the McCarthy *amb* function is clearly non-strict, since it can fire with only one of the two arguments available. A process made with this actor, however, is not continuous, and the process is non-determinate.

It is possible to have a determinate process made of non-strict actors. Consider

$$select\ (x, \perp, true) = x$$

$$select\ (\perp, y, false) = y$$

The firing rules implied by this definition are sequential, since a token is required for the third argument, and the value of that argument determines which firing rule applies. Moreover, *select* is functional, so a process made up of repeated firings of this actor is determinate. The Ptolemy icon for this process is shown in figure 9. This function, however, is clearly not strict, since the function does not require that all three arguments be present. Moreover, we will see that this non-strictness is essential for the most general form of recursion. The fact that non-strictness is essential for recursion has been observed before, of course [38].

The next natural question is whether hierarchical nodes should be strict. The example shown in figure 10 suggests a definitive "no" for the answer. A hierarchical node A is composed of subprocesses B and C as shown in the figure. When considering only the expanded definition in figure 10b, we might identify a strict actor consisting of a firing of each of B and C, in either order, once both inputs to the meta actor were available. However, when connected as shown in figure 10a, the network deadlocks, quite unnecessarily.

All three dataflow domains in Ptolemy have non-strict hierarchical nodes. To implement this, most schedulers used in these domains take a simple approach; they flatten the hierarchy before constructing a schedule. This approach may be expensive for large programs with repeated use of the same hierarchical nodes, but it does the job. At least one more sophisticated scheduler [11] constructs strict hierarchical nodes (when this is safe) through a clustering process, in order to build more compact schedules. It ignores the user-specified hierarchy in doing this.

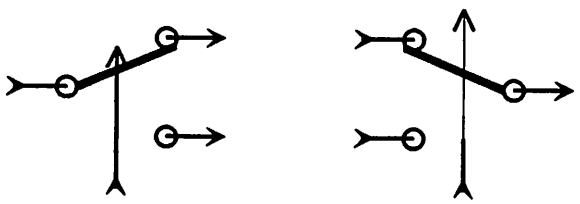


Figure 9. Switch and Select actors in the dynamic dataflow domains of Ptolemy. These are determinate actors that merge or split streams under the control of a Boolean stream.

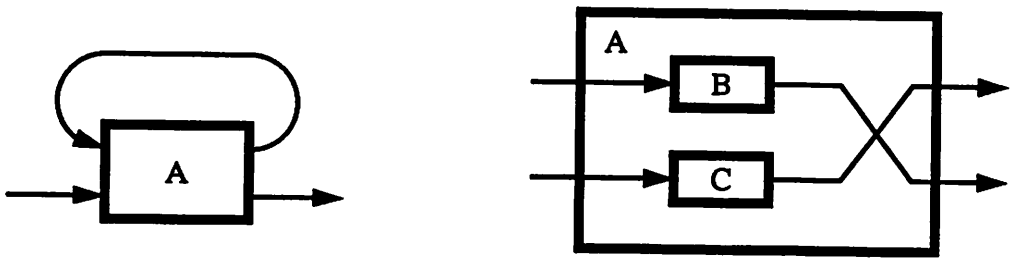


Figure 10. A hierarchical node A in a simple subnetwork (a) and its expanded definition (b). If the actor A is strict, the subnetwork in (a) deadlocks.

3.5 Recurrences and Recursion

Functional languages such as Haskell commonly use recursion to carry state. The comparable mechanism for dataflow process networks is feedback loops, usually with initial tokens, as shown in figure 6a and 6b. These feedback loops specify recurrence relations rather than recursion. Ida and Tanaka have also noted the advantages of this representation [40]. A consequence of this is that recursion plays a considerably reduced role in dataflow process networks compared to functional languages. But this does not mean that recursion is not useful.

Consider the "sieve of Eratosthenes," an algorithm considered by Kahn and MacQueen [44]. It computes prime numbers by constructing a chain of "filters", one for each prime number it has found so far. Each filter removes from the stream any multiple of its prime number. The algorithm starts with a single filter for the prime number 2 in the chain and runs each successively larger integer through the chain of filters. Each time a number gets through to the end of the chain, it must be prime, so a new filter is created and added to the chain. A recursive implementation of this algorithm is concise, convenient, and elegant, although of course we can express any recursive algorithm iteratively [38].

A recursive implementation in the dynamic dataflow domain of Ptolemy is shown in figure 11. The icon with the concentric squares is actually a higher-order function (explained further below) that invokes a named hierarchical node (*sift*) when it fires. In this case, the named hierarchical node is a recursive reference to the very hierarchical node in which the icon appears. More direct expression of recursion is not yet supported by the Ptolemy graphical interface, although it is supported in the underlying kernel.

Note that recursion in figure 11 expresses a "mutable graph", in that the structure of the graph changes as the program executes. Such dynamics are also permitted by Kahn and MacQueen [44] and in TLDF [76]. Mutability, however, considerably complicates compile-time analysis of the graph. The compile-time scheduling methods in [12] and [50] have yet to be extended to recursive graphs. This raises the interesting question of whether recursion precludes compile-time scheduling. We find, perhaps somewhat surprisingly, that often it does not. To illustrate this point, we will derive a recursive implementation of the fast Fourier transform (FFT) in the syn-

chronous dataflow domain in Ptolemy, and show that it can be completely scheduled at compile time. It can even be statically parallelized, with the recursive description imposing no impediment. The classic derivation of the FFT leads directly to a natural and intuitive recursive representation. For completeness, we repeat this simple derivation here.

The N-th order discrete Fourier transform DFT of a sequence $x(n)$ is given by

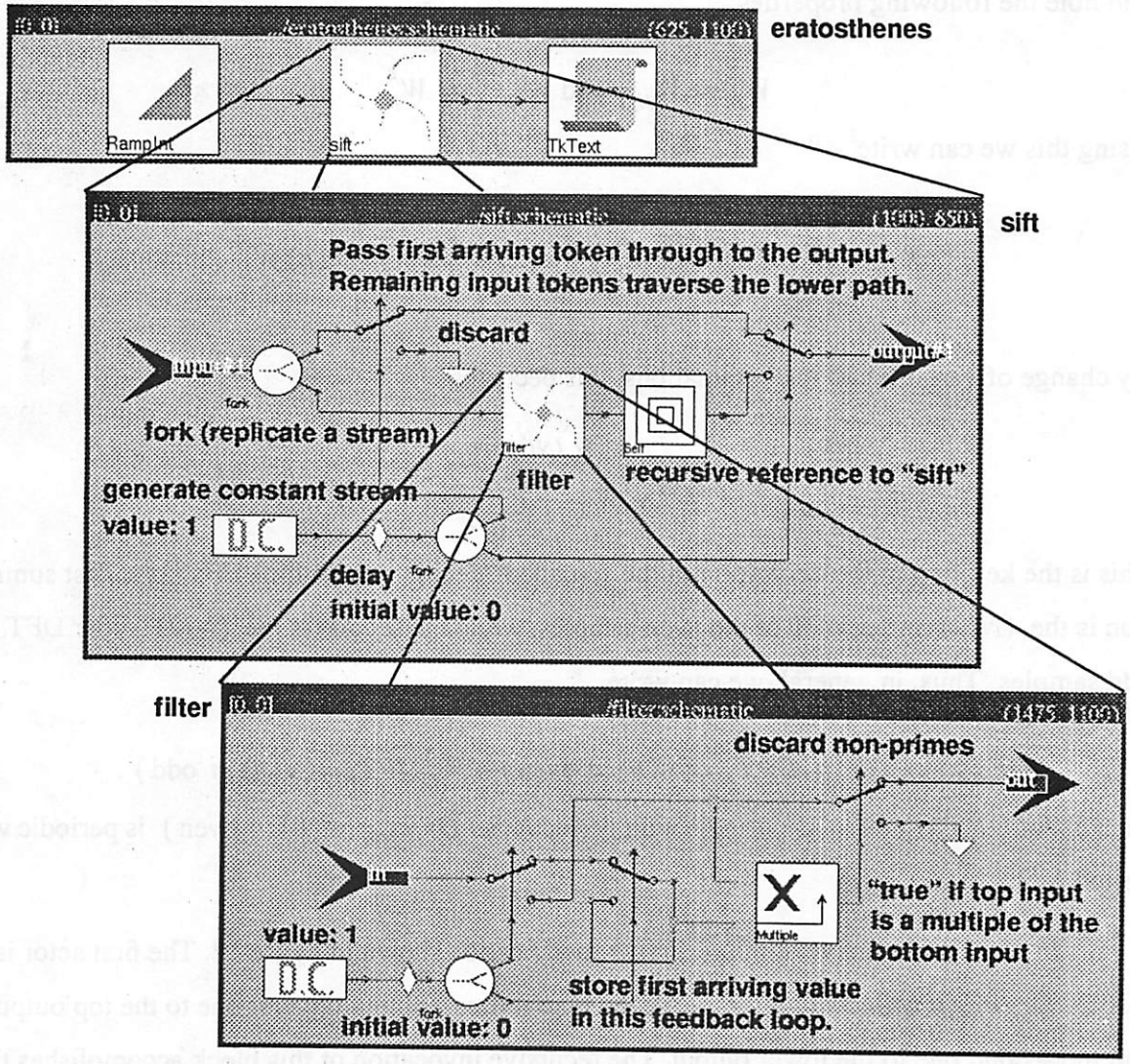


Figure 11. A recursive implementation of the sieve of Eratosthenes in the dynamic dataflow domain in Ptolemy. The top-level system (with just three actors) produces all the integers greater than 1, filters them for primes, and displays the results. Other icons are explained once each.

$$X_k = \sum_{n=0}^{N-1} x(n)\, e^{-j\left(\frac{2\pi}{N}\right)kn} \tag{23}$$

for $0 \le k < N$. To get the values for other k, simply periodically repeat the values given above with period N. Define

$$W_N = e^{-j\left(\frac{2\pi}{N}\right)} \tag{24}$$

and note the following properties:

$$W_N^2 = W_{N/2} \text{ and } W_N^{N+k} = W_N^k. \tag{25}$$

Using this we can write

$$X_k = \sum_{n=0}^{N-1} x(n)\, W_N^{kn} = \sum_{\substack{n=0 \\ n \text{ even}}}^{N-2} x(n)\, W_N^{kn} + \sum_{\substack{n=1 \\ n \text{ odd}}}^{N-1} x(n)\, W_N^{kn} \tag{26}$$

By change of variables on the summations, this becomes

$$X_k = \sum_{n=0}^{(N/2)-1} x(2n)\, W_{N/2}^{kn} + \left(\sum_{n=0}^{(N/2)-1} x(2n-1)\, W_{N/2}^{kn} \right) W_N^k. \tag{27}$$

This is the key step in the derivation of the so-called "decimation-in-time FFT"; the first summation is the $(N/2)$ order DFT of the even samples, while the second is the $(N/2)$ order DFT of odd samples. Thus, in general, we can write

$$DFT_N(x(n)) = DFT_{N/2}(x(n); n \text{ even}) + W_N^k DFT_{N/2}(x(n); n \text{ odd}). \tag{28}$$

Recall that $DFT_N(x(n))$ is periodic with period N, so $DFT_{N/2}(x(n); n \text{ even})$ is periodic with period $N/2$.

From this, we arrive at the recursive specification shown in figure 12. The first actor is a *distributor*, which collects two samples each time it fires, routing the first one to the top output and the second one to the lower output. The recursive invocation of this block accomplishes the decimation in time. The outputs of the distributor are connected to two *IfThenElse* blocks, represent one of two possible replacement subsystems. When the *order* parameter is larger than some

threshold, the *IfThenElse* block replaces itself with a recursive reference to the galaxy within which it sits. When it gets below some threshold, then the *IfThenElse* block replaces itself with some direct implementation of a small order FFT. The *IfThenElse* block is another example of a higher-order function, and will be discussed in more detail below. The *repeat* block takes into account the periodicity of the DFTs of order $N/2$ without duplicating the computation. The *expgen* block at the bottom simply generates the W_N^k sequence. The sequence might be precomputed, or computed on the fly.

A more traditional visual representation of an FFT is shown in figure 13. This representation is extremely inconvenient for programming, however, since it cannot represent FFTs of the size typically used (128 to 1024 points). Moreover, any such visual representation has the order of the FFT and the granularity of the specification hard-wired into the specification. It is better to have both parameterized, as in figure 12. Moreover, I would argue that the visual representation in figure 12 is more intuitive, since it is a more direct representation of the underlying idea.

An interesting generalization of the conditional used in the recursion in figure 12 would use templates on the parameter values to select from among the possible implementations for the node. This would make the recursion stylistically identical to that found in functional languages

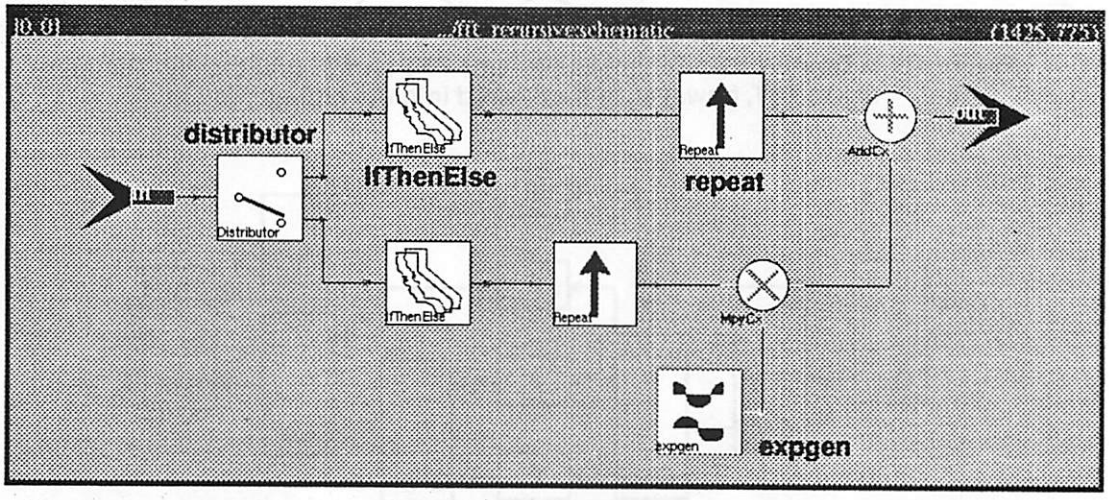


Figure 12. A recursive specification of an FFT implemented in the SDF domain in Ptolemy. The recursion is unfolded during the setup phase of the execution, so that the graph can be completely scheduled at compile time.

like Haskell, albeit with a visual syntax. This can be illustrated with another practical example of an application of recursion.

Consider the system shown in figure 14. It shows a multirate signal processing application: an analysis/synthesis filter bank with harmonically spaced subbands. The stream coming in at the left is split by matching highpass and lowpass filters (labeled "QMF"). These are decimating polyphase FIR filters, so for every two tokens consumed on the input, one token is produced on each of two outputs. The left-most QMF only is labeled with the number of tokens consumed and produced, but the others behave the same way. The output of the lowpass side is further split by a second QMF, and the lowpass output of that by a third QMF. The boxes labeled "F" represent some function performed on the decimated stream (such as quantization). The QMF boxes to the right of these reconstruct the signal using matching polyphase interpolating FIR filters.

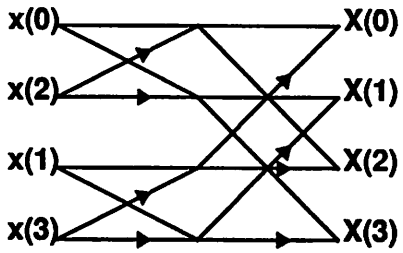


Figure 13. A fourth-order decimation-in-time FFT shown graphically. The order of the FFT, however, is hard-wired into the representation.

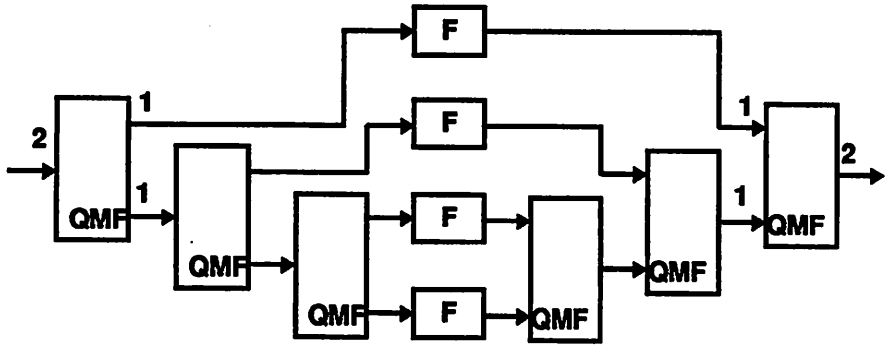


Figure 14. An analysis/synthesis filter bank under the SDF model. The depth of the filter bank, however, is hard-wired into the representation.

There are four distinct sample rates in figure 14 with a ratio of 8 between the largest and the smallest. This type of application typically needs to be implemented in real time at low cost, so compile-time scheduling is essential.

The graphical representation in figure 14 is useful for developing intuition, and exposes exploitable parallelism, but it is not so useful for programming. The depth of the filter bank is hard-wired into the visual representation, so it cannot be conveniently made into a parameter of a filter-bank module. The representation in figure 15 is better. A hierarchical node called "FB", for "filterbank" is defined, and given a parameter D for "*depth*". For $D > 0$ the definition of the block is at the left. It contains a self-reference, with the parameter of the inside reference changed to D - 1. When D=0, the definition at the right is used. The system at the top, consisting of just one block, labeled "FB(D = 3)", is exactly equivalent to the representation in figure 14, except that the visual representation does not now depend on the depth. The visual recursion in figure 15 can be unfolded completely at compile time, exposing all exploitable parallelism, and incurring no unnecessary run-time overhead

3.6 Higher-Order Functions

In dataflow process networks, all arcs connecting actors represent streams. The icons represent both actors and the processes made up of repeated firings of the actor. Functional languages often represent such processes using higher order functions. For example, in Haskell,

$$\text{map}\, f\, xs$$

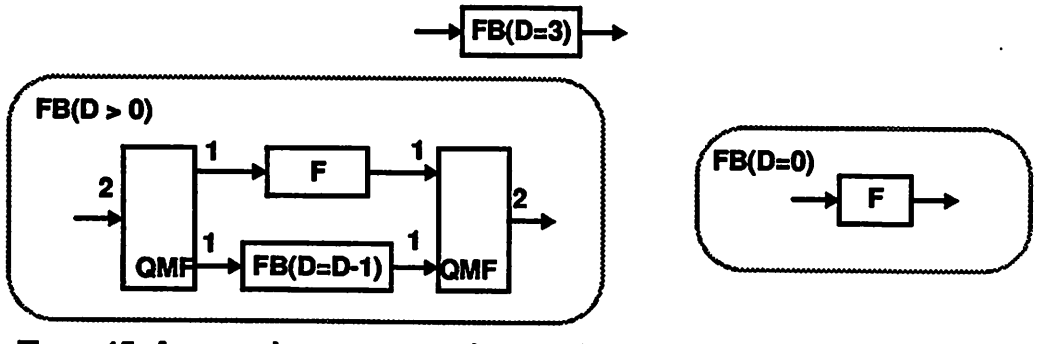


Figure 15. A recursive representation of the filter bank application. This representation uses template matching.

applies the function *f* to the list *xs*. Every single-input process in a dataflow process network constitutes an invocation of such a higher order function, applied to a stream rather than a list. In a visual syntax, the function itself is specified simply by the choice of icon. Moreover, Haskell has the variant

zipWith *f xs ys*

where the function *f* has arity two. This corresponds simply to a dataflow process with two inputs. Similarly, the Haskell function

scanl *f a xs*

takes a scalar *a* and a list *xs*. The function *f* is applied first to *a* and the head of *xs*. The function is then applied to the first returned value and the second element of *xs*. A corresponding visual syntax for a dataflow process network is given in figure 16.

Recall our proposed syntactic sugar for representing feedback loops such as that in figure 16 using actors with state. Typically the initial value of the state (*a*) will be a parameter of the node. In fact, dataflow processes with state cover many of the commonly used higher-order functions in Haskell.

The most basic use of icons in our visual syntax may therefore be viewed as implementation a small set of built-in higher-order functions. More elaborate higher-order functions will be more immediately recognizable as such, and will prove extremely useful. Pioneering work in the use of higher-order functions in visual languages was done by Hills [36], Najork and Golin [59], and Reekie [68]. We will draw on this work here.

I created an actor in Ptolemy called *Map* that generalizes the Haskell *map*. It has a parameter that specifies another actor (primitive or hierarchical) by name. That actor defines one or

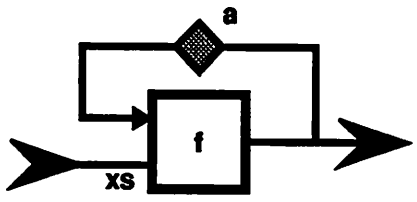


Figure 16. Visual syntax for the dataflow process network equivalent of the Haskell "scanl f a xs" higher-order function.

more processes that are applied to any number of input streams. Its icon is shown in figure 17a. A simple example of its use is shown in figure 17c. In that figure, three *Ramp* actors generate three streams consisting of increasing integers. The *Map* actor simply applies any named actor to those three streams. If the named actor has a single input, then three instances will be created. The parameters of the three instances can be set independently via the parameters of the *Map* actor.

My implementation of *Map* is simple but effective. It simply creates one or more instances of a the specified actor (which may itself be a hierarchical node) and splices those instance into its own position in the graph. Thus, we call the specified actor the *replacement actor*, since it takes the place of the *Map* actor. The *Map* actor then self-destructs. This is done in the setup phase of execution so that no overhead is incurred for the higher order function during the run phase of execution, which for signal processing applications is the most critical.

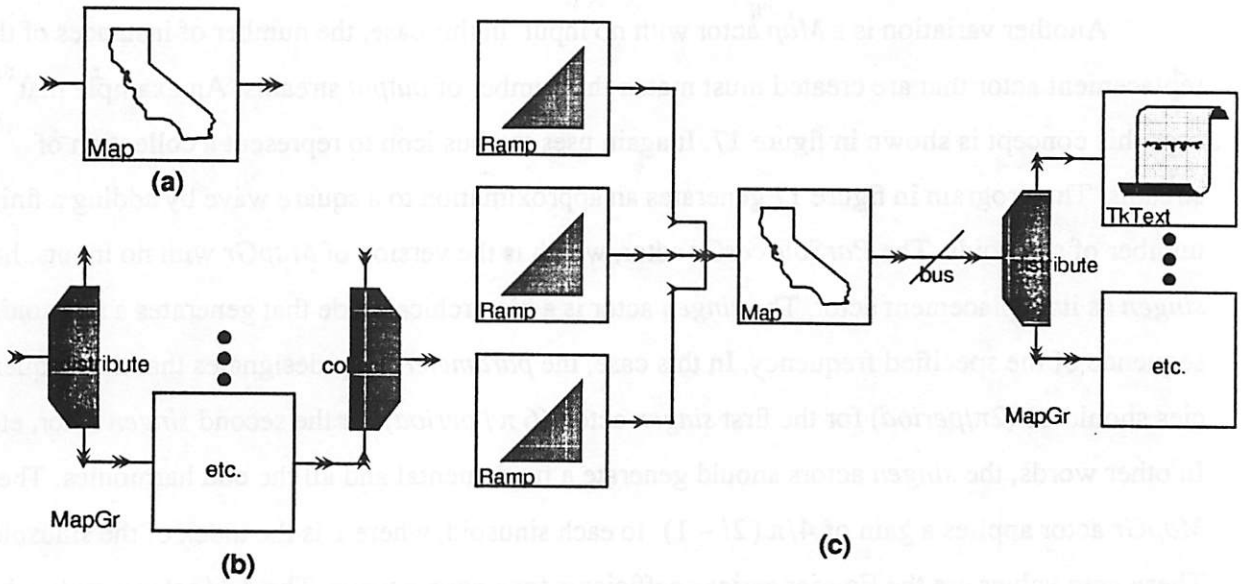


Figure 17. (a) An icon for a simple higher-order function in Ptolemy, Map, which applies a named actor to its input streams. (b) A variant of Map where the actor to apply to the input streams is specified graphically instead of textually. (c) A simple use of the two types of Map actors where three ramps (linearly increasing sequences) have three instances of a named actor applied to them and the three resulting streams are displayed using TkText.

In the visual programming languages ESTL [59] and DataVis [36], higher-order functions use a "function slots" concept, visually representing the replacement function as a box inside the icon for the higher-order function. I have implemented in Ptolemy a conceptually similar visual representation, using the icon shown in figure 17b. The replacement actor is graphically connected to the (rather elaborate) icon, as shown in figure 17c. There, at the right, a version of the *MapGr* actor (one with inputs but no outputs) specifies that the *TkText* actor should be applied to each of the input streams. Notice also that the streams going from the *Map* to the *MapGr* are compactly represented using the Ptolemy *bus* icon, a slash through the connecting wire. The bus icon has a single parameter, the width of the bus.

A number of additional variations are possible. First, the replacement actor may have arity larger than one, in which case the input streams are grouped in appropriately sized groups to provide the arguments for each instance of the specified actor. For example, if the replacement actor has arity two, and there are 12 input streams, then six instances of the actor will be created. The first instance will process the first two streams, the second the next two streams, etc.

Another variation is a *Map* actor with no input. In this case, the number of instances of the replacement actor that are created must match the number of *output* streams. An example that uses this concept is shown in figure 17. It again uses the bus icon to represent a collection of streams. The program in figure 17 generates an approximation to a square wave by adding a finite number of sinusoids. The *ParSourcesGr* actor, which is the version of *MapGr* with no inputs, has *singen* as its replacement actor. The *singen* actor is a hierarchical node that generates a sinusoidal sequence of the specified frequency. In this case, the *parameter_map* designates that the frequencies should be (2π/*period*) for the first *singen* actor, ($6\ \pi$ / *period*) for the second *singen* actor, etc. In other words, the *singen* actors should generate a fundamental and all the odd harmonics. The *MapGr* actor applies a gain of $4/\pi\,(2i - 1)$ to each sinusoid, where i is the index of the sinusoid. These gain values are the Fourier series coefficients for a square wave. The *Add* actor simply adds all its inputs. The *XMgraph* actor plots the signal, as shown in figure 19.

Since the *Map* actor always creates at least one instance of the replacement actor, it cannot be used directly for recursion. Such a recursion would never terminate. A variant of the *Map* actor

can be defined that instantiates the replacement actor(s) only at run time. This is (essentially) what we used in figure 11 to implement recursion. Using dynamic dataflow, the *dynamic Map* actor fires conditionally. When it fires, it creates an instance of its replacement actor (which may be a hierarchical node recursively referenced), and self-destructs.

The dynamic *Map* was the first higher-order function implemented in Ptolemy (it was implemented under a different name by Soonhoi Ha). Its runtime operation is quite expensive, however, requiring dynamic creation of a dataflow graph. So there is still considerable motivation

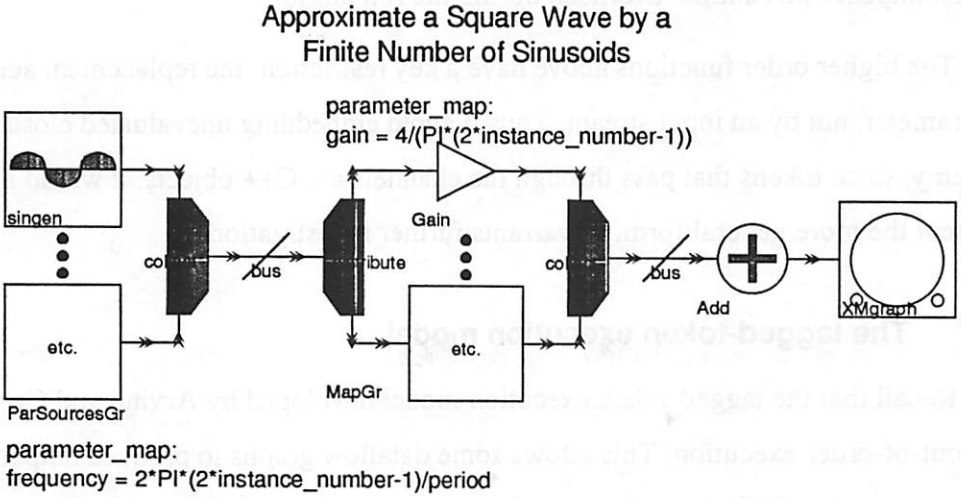


Figure 18. The Map actor in Ptolemy can be used with no inputs, in which case, the number of instances of the replacement actor that are created must match the number of outputs. The slash through a wire indicates a bus, which represents a collection of streams.

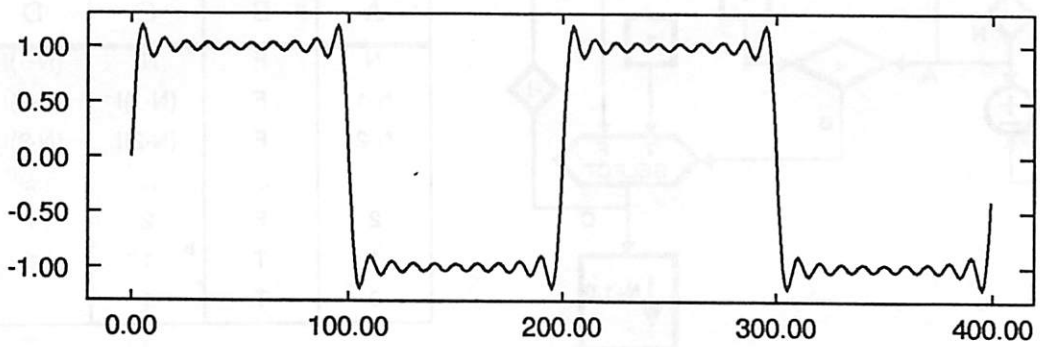


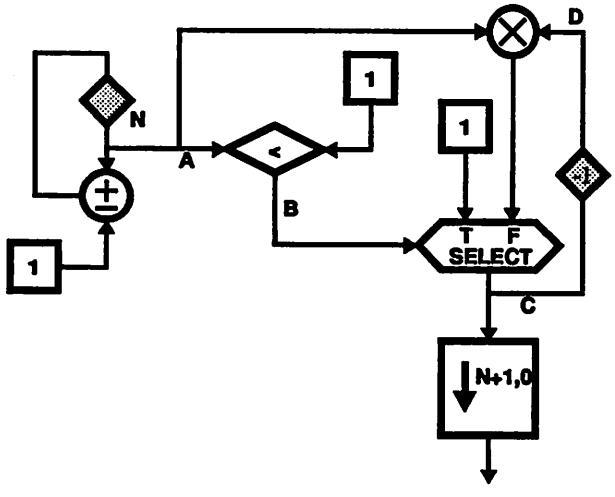
Figure 19. First 400 values computed by the program in figure 17. It is an approximation to a square wave computed by adding 10 sinusoids. Thus, the bus widths in figure 17 are 10.

for recursion that can be statically unrolled, as done in figure 12. In fact, that system is implemented using another higher-order function, *IfThenElse*, which is derived from *Map*. The *IfThenElse* actor takes two replacement actors as parameters plus a predicate. The predicate specifies which of the two replacement actors should be used. That actor is expanded into a graph instance and spliced into the position of the *IfThenElse* actor. The *IfThenElse* actor, like the *Map* actor, then self-destructs. Since the unused replacement actor argument is not evaluated, the semantics are non-strict, and the *IfThenElse* actor can be used to implement recursion. The recursion is completely evaluated during the setup phase of execution (or at compile time), so the recursion imposes no runtime overhead during the run phase.

The higher order functions above have a key restriction: the replacement actor is specified by a parameter, not by an input stream. Thus, I avoid embedding unevaluated closures in streams. In Ptolemy, since tokens that pass through the channels are C++ objects, it would not be hard to implement the more general form. It warrants further investigation.

3.7 The tagged-token execution model

Recall that the tagged-token execution model developed by Arvind and Gostelow [5][6] allows out-of-order execution. This allows some dataflow graphs to produce output that would deadlock under the FIFO channel model. An example is shown in figure 20. This graph computes



A	B	C	D
N	F	N!	(N-1)!
N-1	F	(N-1)!	(N-2)!
N-2	F	(N-2)!	(N-3)!
...	...	...	...
2	F	2	1
1	T	1	1
0	T	1	

Figure 20. This factorial program deadlocks without out-of-order execution, as provided for example by the tagged token model.

N! if out-of-order execution is allowed, but deadlocks without producing an output under the FIFO model. The sequence of values on the labeled arcs is given in the table in the figure.

The loop at the left counts down from N to 0, since the delay is initialized to N and the value circulating in the loop is decremented by 1 each time around. The test (diamond shape) compares the value at A to 1. When $A<1$, it outputs a *True*. Until that time, the *select* is not enabled, because there are no tokens on the *False* input. But notice that at that time, the queue at the control input (B) of the select has N *False* tokens followed by one *True* token. The *False* tokens still cannot be consumed. If out-of-order execution is not allowed, then the *select* will never be able to fire. However, since the *select* has no state, there is no reason to prohibit out-of-order execution.

Out-of-order execution requires bookkeeping like that provided by the tagged-token model. The consumption of the *True* token is by the (N+1)-th firing (logically) of the *select*. Thus, the 1 produced at its output is (logically) the (N+1)-th output produced by the *select*. Hence, at C, we show the 1 output as the *last* entry in the table, even though it is the first one produced temporally. The logical ordering must be preserved.

Recall that a delay is an initial token on a channel. The delay at the left is an ordinary delay, where the initial token is initialized to value N. The delay on the right, however, is something new, a *negative delay*. Instead of an initial token, this delay discards the first token that enters the channel. It can be implemented in a variety of ways, one of which is shown in figure 21. The effect of the negative delay is shown in column D; the first token (logically, not temporally) produced by the *select* is discarded by the negative delay. Thus, the 1 produced by the (N+1)-th firing (logically) of the select must be consumed by the N-th firing of the multiply at the upper

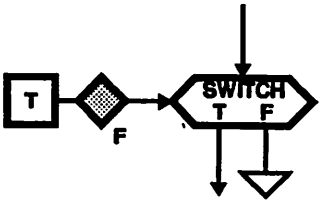


Figure 21. One way to implement a negative delay, which discards the first token that arrives on the input stream.

right. The other input of the multiply has a value "1" as its N-th input (A), so the N-th output (logically) or first output (temporally) of the multiply is $1 \times 1 = 1$. This makes available the N-th token (logically) of the *select False* input, which can now be consumed by the N-th firing (logically) of the *select*. The "1" produced here will be multiplied by 2, enabling the (N-2)-th firing of the *select*. We continue until the first firing (logically) of the *select* produces $N!$. At this point, there are $N+1$ tokens at the *downsampler* input (the icon at the bottom with the downward arrow), enabling it. It consumes these tokens and outputs the first one (logically). Thus the output of the downsampler is $N!$.

Note that although this might appear to be an unduly complicated way to compute a factorial, it nonetheless demonstrates that enabling out-of-order execution does increase the expressiveness in the language. Of course, this has limited value if its only use is to represent obscure and unnecessarily complicated algorithms.

3.8 Data types and polymorphism

A key observation about our dataflow process networks so far is that the only datatype represented visually is the stream. The tokens on a stream can have arbitrary type, so this approach is more flexible than it sounds like at first. For instance, we can embed arrays into streams either directly by sequencing the elements of the array, or by encapsulating each array into a single token, or by generalizing to multidimensional streams [49][75]. In Ptolemy, tokens can contain arbitrary C++ objects, so the actors can operate on these tokens in rather sophisticated ways, making effective use of data abstraction.

Ptolemy networks are strongly typed. Each actor port (input or output) has a type, and type consistency is statically checked. Polymorphism, in which a single actor can operate on any of a variety of datatypes, is supported in a natural way.

Hudak distinguishes two types of polymorphism, *parametric* and *ad-hoc* (or *overloading*) [38]. In the former, a function behaves the same way regardless of the data type of its arguments. In the latter, the behavior can be different, depending on the type. Although in principle both are supported in Ptolemy, we have made more use of parametric polymorphism in the visual pro-

gramming syntax. The way that parametric polymorphism is handled is that actors declare their inputs or outputs to be of type "anytype". The actors then operate on the tokens via abstracted type handles.

Polymorphic blocks in Ptolemy include all those that perform control functions on streams, like the *commutator* and *distributor* in figure 12. The *Map* actor is also polymorphic, although in a somewhat more complicated way. Its inputs and outputs are declared to be "anytype", and type resolution is redone for actors connected to it after the substitution of the replacement actor(s).

3.9 Parallelism

For functional languages, the dominant view appears to be that parallelism must be explicitly defined by the programmer by annotating the program with the processor allocation [38]. Moreover, as indicated by Harrison [33], the ubiquity of recursion in functional programs sequentializes what would otherwise be parallel algorithms. Harrison proposes using higher-order functions to express parallel algorithms in a functional language, in place of recursion. The parallel implementation is accomplished by mechanized program transformations from the higher-order function description. This is called "transformational parallel programming," and has also been explored by Reekie and Potter [70] in the context of process networks. The transformations could also be interactive, supported by "meta-programming". One transformation methodology is the unfold/fold method of Burstall and Darlington [17], which is based on partial (symbolic) evaluation and substitution of equal expressions.

In the dataflow community, by contrast, parallelism has always been implicit. This is, in part, due to the scarce use of recursion. A dataflow graph typically reveals a great deal of parallelism that can be exploited either by run-time hardware [3] or, if the firing sequence is sufficiently predictable, a compiler [30][65][73][74].

Dataflow process networks can combine the best of these. Parallelism can be implicit, and higher-order functions can be used to simplify the syntax of the graphical specification. The phased execution, in which the static higher-order functions are evaluated during a setup phase, is

analogous to the fold/unfold method of Burstall and Darlington [17], but there is no need for a specialized transformation tool that "understands" the semantics of the higher-order functions. Thus, parallelism is exploited equally well with user-defined higher-order functions as with those that are built into the language.

Moreover, in a surprising twist, the use of static higher-order functions enables the use of recursion *without compromising parallelism*. As long as the recursion can be evaluated during the setup phase, it does not sequentialize the program. Thus, we regain much of the elegance that the use of recursion lends to functional languages. An example (a recursive specification of an FFT) is given above in figure 12. In situations where the recursion cannot be evaluated during the setup phase, as in the sieve of Eratosthenes in figure 11, the algorithm is inherently sequential.

4.0 Conclusions

Signal processing software environments are domain-specific. Some of the techniques they use, including (and maybe especially) their visual syntax has only been proven in this domain-specific context. Nonetheless, they have (or can have) the best features of the best modern languages, including natural and efficient recursion, higher-order functions, data abstraction, and polymorphism.

This paper presents a theory of design that has been (at least partially) put into practice by the signal processing community. In the words of Milner [53], such a theory "does not stand or fall by experiment in the conventional scientific sense." It is the "pertinence" of a theory that is judged by experiment rather than its "truth".

5.0 Acknowledgments

I would like to thank the entire Ptolemy team, but especially Joe Buck, Soonhoi Ha, Alan Kamas, and Dave Messerschmitt, for conceiving and building a magnificent infrastructure for the kinds of experiments described here. I would also like to gratefully acknowledge helpful com-

ments on early drafts of the paper from Shuvra Bhattacharyya, Tom Parks, and Juergen Teich. The inspiration for this paper came originally from Jack Dennis, who pointed out the need to relate the work with dataflow in signal processing with the broader computer science community.

6.0 References

[1] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, MA, 1985.

[2] W. B. Ackerman, "Data Flow Languages," *Computer*, Vol. 15, No. 2, February 1982.

[3] Arvind, L. Bic, T. Ungerer, "Evolution of Data-Flow Computers," in *Advanced Topics in Data-Flow Computing*, ed. J.-L. Gaudiot and L. Bic, Prentice-Hall, 1991.

[4] Arvind and J. D. Brock, "Resource Managers in Functional Programming," *J. of Parallel and Distributed Computing*, Vol. 1, No. 5-21, 1984

[5] Arvind and K. P. Gostelow, "Some Relationships between Asynchronous Interpreters of a Dataflow Language," In *Formal Description of Programming Languages*, IFIP Working Group 2.2, 1977.

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

[7] E. A. Ashcroft and R. Jagannathan, "Operator Nets," in *Proc. IFIP TC-10 Working Conf. on Fifth-Generation Computer Architectures*, North-Holland, The Netherlands, 1985.

[8] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1270-1282.

[9] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Tr. on Automatic Control*, Vol. 35, No. 5, pp. 525-546, May 1990.

[10] S. Bhattacharyya and E. A. Lee, "Memory Management for Synchronous Dataflow Programs," to appear in *IEEE Tr. on Signal Processing*, May 1994.

[11] S. Bhattacharyya and E. A. Lee, "Looped Schedules for Dataflow Descriptions of Multirate Signal Processing Algorithms," **to appear** in *Formal Methods in System Design*, (updated from UCB/ERL Technical Report, May 21, 1993).

[12] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Tech. Report UCB/ERL 93/69, Ph. D. Dissertation, Dept. of EECS, University of California, Berkeley, CA 94720, 1993.

[13] J. Buck and E. A. Lee, "The Token Flow Model," presented at *Data Flow Workshop*, Hamilton Island, Australia, May, 1992. Also in *Advanced Topics in Dataflow Computing and Multithreading*, ed. Lubomir Bic, Guang Gao, and Jean-Luc Gaudiot, IEEE Computer Society Press, 1994.

[14] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Multirate Signal Processing in Ptolemy", *Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing*, Toronto, Canada, April, 1991.

[15] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*, April, 1994.

[16] W. H. Burge, "Stream Processing Functions," *IBM J. of Research and Development*, Vol. 19, No. 1, January, 1975.

[17] R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," JACM, Vol. 24, No. 1, 1977.

[18] N. Carriero and D. Gelernter, "Linda in Context," *Comm. of the ACM*, Vol. 32, No. 4, pp. 444-458, April 1989.

[19] M. J. Chen, "Developing a Multidimensional Synchronous Dataflow Domain in Ptolemy", MS Report, ERL Technical Report UCB/ERL No. 94/16, University of California, Berkeley, CA 94720, May 6, 1994.

[20] A. Church, *The Calculi of Lambda-Conversion*, Princeton University Press, Princeton, NJ, 1941.

[21] F. Commoner and A. W. Holt, "Marked Directed Graphs," *Journal of Computer and System Sciences*, Vol. 5, pp. 511-523, 1971.

[22] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. Van Meerbergen, S. Note, J. Huisken, "Architecture-driven synthesis techniques for mapping digital signal processing algorithms into silicon," in special issue on computer-aided design of *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 319-335, February, 1990.

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

[24] J. B. Dennis, "Data Flow Supercomputers," *IEEE Computer*, Vol 13, No. 11, November, 1980.

[25] J. B. Dennis, "Stream Data Types for Signal Processing," unpublished memorandum, September 28, 1992.

[26] D. Desmet and D. Genin, "ASSYNT: Efficient Assembly Code Generation for DSP's starting from a Data Flow-graph," *Trans. of ICASSP '93*, Minneapolis, April, 1993.

[27] E. W. Dijkstra, A Discipline of Programming, Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[28] J. Franco, D. P. Friedman, and S. D. Johnson, "Multi-Way Streams in Scheme," *Comput. Lang.*, Vol. 15, No. 2, pp. 109-125, 1990.

[29] D. K. Gifford and J. M. Lucassen, "Integrating Functional and Imperative Programming," in *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, pp. 28-38, 1986.

[30] S. Ha, "Compile-Time Scheduling of Dataflow Program Graphs with Dynamic Constructs," Ph.D. Dissertation, EECS Dept., University of California, Berkeley, CA 94720, April 1992.

[31] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1305-1319.

[32] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Dordrecht, 1993.

[33] P. G. Harrison, "A Higher-Order Approach to Parallel Algorithms," *The Computer Journal*, Vol. 35, No. 6, 1992.

[34] J. Hicks, D. Chiou, B. S. Ang, and Arvind, "Performance Studies of Id on the Monsoon Dataflow System," *J. of Parallel and Distributed Computing*, Vol. 18, No. 3, pp. 273-300, July, 1993.

[35] P. Hilfinger, "A High-Level Language and Silicon Compiler for Digital Signal Processing", Proceedings of the *Custom Integrated Circuits Conference*, IEEE Computer Society Press, Los Alamitos, CA 1985, pp 213-216.

[36] D. D. Hills, "Visual Languages and Computing Survey: Data Flow Visual Programming Languages," *J. of Visual Languages and Computing*, Vol. 3, p. 69-101.

[37] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[38] P. Hudak, "Introduction to Haskell and functional programming", *ACM Computing Surveys*, Sept. '89.

[39] J. Hughes, "Compile-time Analysis of Functional Programs," in Turner, ed., *Research Topics in Functional Programming*, Addison-Wesley, 1990.

[40] T. Ida and J. Tanaka, "Functional Programming with Streams," *Information Processing '83*, Elsevier Science pubs.(North-Holland), 1993.

[41] R. Jagannathan, "Parallel Execution of GLU Programs," presented at *2nd International Workshop on Dataflow Computing*, Hamilton Island, Queensland, Australia, May 1992.

[42] R. Jaganathan and E. A. Ashcroft, "Eazyflow: A Hybrid Model for Parallel Processing," In *Proc. Int. Conf. on Parallel Processing*, pp 514-523, IEEE, August, 1984.

[43] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[44] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[45] D. J. Kaplan, *et al.*, "Processing Graph Method Specification Version 1.0," the Naval Research Laboratory, Washington D.C., December 11, 1987.

[46] R. M. Karp, R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal*, Vol. 14, pp. 1390-1411, November, 1966.

[47] P. J. Landin, "A Correspondence Between Algol 60 and Church's Lambda Notation," Communications of the ACM, Vol. 8, 1965.

[48] E. A. Lee, "Consistency in Dataflow Graphs", *IEEE Transactions on Parallel and Distributed Systems"*, Vol. 2, No. 2, April 1991.

[49] E. A. Lee, "Representing and Exploiting Data Parallelism Using Multidimensional Dataflow Diagrams," *Proc. of ICASSP* '93, Minneapolis, MN, April, 1993.

[50] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing" *IEEE Transactions on Computers*, January, 1987.

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

[52] P. Le Guernic, T. Gauthier, M. Le Borgne, C. Le Maire, "Programming Real-Time Applications with SIGNAL," *Proceedings of the IEEE*, Vol. 79, No. 9, September 1991.

[53] R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[54] D. McAllester, P. Panagaden, V. Shanbhogue, "Nonexpressibility of Fairness and Signaling," to appear in *JCSS*, 1993.

[55] J. McCarthy, "Recursive Functions of Symbolic Expressions and the computation by machine, Part I", *Comm. of the ACM*, V. 3, No. 4 (April 1960).

[56] J. McCarthy, "A Basis for a Mathematical Theory of Computation," in *Computer Programming and Formal Systems*, North-Holland, pp. 33-70, 1978.

[57] J. McGraw, "Sisal: Streams and Iteration in a Single Assignment Language", *Language Reference Manual*, Lawrence Livermore National Laboratory, Livermore, CA 94550.

[58] R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[59] M. A. Najork, E. Golin, "Enhancing Show-and-Tell with a Polymorphic Type System and Higher-Order Functions," in IEEE Workshop on Visual Languages, Skokie, Illinois, October 4-6, 1990, pp. 215-220.

[60] T. J. Olson, N. G. Klop, M. R. Hyett, S. M. Carnell, "MAVIS: a visual environment for active computer vision," *Proceedings 1992 IEEE Workshop on Visual Languages*, Seattle, WA, USA, 15-18 Sept. 1992, IEEE Comput. Soc. Press, 1992, p. 170-6.

[61] J. S. Onanian, "A Signal Processing Language for Coarse Grain Dataflow Multiprocessors," MIT/LCS/TR-449, 545 Technology Sq., Cambridge, MA 02139, June 14, 1989.

[62] P. Panagaden and V. Shanbhogue, "The Expressive Power of Indeterminate Dataflow Primitives," *Information and Computation*, Vol. 98, No. 1, May 1992.

[63] J. Pino, S. Ha, E. Lee, J. Buck, "Software Synthesis for DSP Using Ptolemy", invited paper in the *Journal on VLSI Signal Processing*, special issue on "Synthesis for DSP", to appear, 1994.

[64] D. G. Powell, E. A. Lee, W. C. Newman, "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," *Proceedings of ICASSP*, San Francisco, March, 1992.

[65] H. Printz, "Automatic Mapping of Large Signal Processing Systems to a Parallel Machine," Memorandum CMU-CS-91-101, School of Computer Science, Carnegie Mellon University, Ph.D. Thesis, May 15, 1991.

[66] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast Prototyping of Datapath-Intensive Architectures," *IEEE Design and Test of Computers*, pp. 40-51, June 1991.

[67] J. Rasure and C. S. Williams, "An Integrated Visual Language and Software Development Environment", *Journal of Visual Languages and Computing*, Vol 2, pp 217-246, 1991.

[68] H. J. Reekie, "Toward Effective Programming for Parallel Digital Signal Processing," Research Report 92.1, University of Technology, Sydney, PO Box 123, Broadway NSW 2007, May 1992.

[69] H. J. Reekie, "Integrating Block-Diagram and Textual Programming for Parallel DSP," *Proc. 3d Int. Symp. on Signal Processing and its Applications*, Queensland, Australia, August 1992.

[70] H. J. Reekie and J. Potter, "Transforming Process Networks," presented at the Massey Functional Programming Workshop, Massey University, Parmerston North, New Zealand, August 1992.

[71] H. J. Reekie and M. Meyer, "The Host-Engine Software Architecture for Parallel Digital Signal Processing," *Proc. of the Australian Workshop on Parallel and Real-Time Systems*, Melbourne, Australia, North-Holland, July, 1994.

[72] K. S. Shanmugan, G. J. Minden, E. Komp, T. C. Manning, and E. R. Wiswell, "Block-Oriented System Simulator (BOSS)," Telecommunications Laboratory, University of Kansas, Internal Memorandum, 1987.

[73] G. C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 2, February, 1993.

[74] G. C. Sih and E. A. Lee, "Declustering: A New Multiprocessor Scheduling Technique," *IEEE Trans. on Parallel and Distributed Systems*, June 1993.

[75] D. B. Skillcorn, "Stream Languages and Data-Flow," in *Advanced Topics in Data-Flow Computing*, ed. J.-L. Gaudiot and L. Bic, Prentice-Hall, 1991.

[76] P. A. Suhler, J. Biswas, K. M. Korner, J. C. Browne, "TDFL: A Task-Level Dataflow Language", *J. on Parallel and Distributed Systems*, 9(2), June 1990.

[77] W. W. Wadge and E. A. Ashcroft, *Lucid, the dataflow programming language*, London Academic Press, 1985.

[78] A. L. Wendelborn, H. Garsden, "Exploring the Stream Data Type in SISAL and other Languages," to appear in *Advanced Topics in Dataflow Computing and Multithreading*, ed. Lubomir Bic, Guang Gao, and Jean-Luc Gaudiot, IEEE Computer Society Press, 1994.