[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

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

[88] H. J. Reekie and J. M. Potter, “Generating Efficient Loop Code for Programmable DSPs,” *Proc. of ICASSP `94*, Adelaide, Australia, April 1994.

[89] 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.

[90] 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.

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

[92] 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.

[93] V. Srini, “An Architectural Comparison of Dataflow Systems,” *Computer,* 19(3), March 1986.

[94] 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.

[95] J. Vuillemin, *Proof Techniques for Recursive Programs*, Ph. D. Thesis, Computer Science Department, Stanford University, USA, 1973.

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

[97] 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.

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

[63] R. Lauwereins, P. Wauters, M. Adé, J. A. Peperstraete, "Geometric Parallelism and Cyclo-Static Dataflow in GRAPE-II", *Proc. 5th Int. Workshop on Rapid System Prototyping*, Grenoble, France, June, 1994.

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

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

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

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

[68] 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.

[69] O. Maffeïs and P. Le Guernic, "From Signal to Fine-Grain Parallel Implementations," in *Int. Conference on Parallel Architectures and Compilation Techniques*, IFIP A-50, North-Holland, pp. 237-246, August 1994.

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

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

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

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

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

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

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

[77] 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.

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

[79] J. L. Pino, T. M. Parks, E. A. Lee, "Mapping Multiple Independent Synchronous Dataflow Graphs onto Heterogeneous Multiprocessors", *Proceedings of IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1994.

[80] J. L. Pino, S. Ha, E. A. Lee, J. T. Buck, "Software Synthesis for DSP Using Ptolemy", *Journal of VLSI Signal Processing*, Vol. 9, No. 1, pp 7-21, January 1995.

[81] 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.

[40] J. B. Dennis, “Stream Data Types for Signal Processing,” unpublished memorandum, September 28, 1992.

[41] 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.

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

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

[44] 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.

[45] 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.

[46] 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.

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

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

[49] 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.

[50] 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.

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

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

[53] P. Hudak, “Introduction to Haskell and functional programming”, *ACM Computing Surveys*, Sept. ‘89.

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

[55] T. Ida and J. Tanaka, “Functional Programming with Streams,” *Information Processing ‘83*, Elsevier Science pubs.(North-Holland), 1993.

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

[57] 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.

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

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

[60] D. J. Kaplan, *et al.*, “Processing Graph Method Specification Version 1.0,” Unpublished Memorandum, The Naval Research Laboratory, Washington D.C., December 11, 1987.

[61] 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.

[19] F. Boussinot, “Reseaux de Processus Avec Melange Equitable: Une Approche du Temps Reel,” Thèse d’Etat (Ph.D. thesis, in French), Université P. et M. Curie, CNRS, and Université Paris 7, 2, Place Jussieu, 75221 Paris Cedex 05, FRANCE, June, 1981.

[20] F. Boussinot, “Reseaux de Processus Reactifs,” Rapport de Recherche No. 12/91, INRIA, Sophia-Antipolis, France, November 1991 (in French).

[21] J. D. Brock and W. B. Ackerman, “Scenarios, a Model of Non-Determinate Computation,”, *Proc. Conf. on Formal Definition of Programming Concepts*, LNCS 107, pp. 252-259, Springer-Verlag, Berlin, 1981.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

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

[29] P. Caspi, “Clocks in Dataflow Languages,” *Theoretical Computer Science*, Vol. 94, No. 1, March 1992.

[30] P. Caspi, “Lucid Synchrone,” Proc. OPOPAC, HERMES (Paris) pp. 79-93, 1993.

[31] 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.

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

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

[34] C. Consel and O. Danvy, “Tutorial Notes on Partial Evaluation,” *20th ACM Symp. on Principles of Programming Languages*, pp. 493-501, January, 1993.

[35] A. L. Davis, “Data Driven Nets: A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems,” Technical Report, Department of Computer Science, University of Utah, Salt Lake City, Utah, July 1978.

[36] A. L. Davis and R. M. Keller, “Data Flow Program Graphs”, *Computer*, **15(2)**, February, 1982.

[37] 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,” *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 319-335, February, 1990.

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

[39] J. B. Dennis, “Data Flow Supercomputers,” *IEEE Computer,* Vol 13, No. 11, November, 1980.

6.0 References

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

[2] S. Abramsky, “Reasoning About Concurrent Systems,” in F. B. Chambers, D. A. Duce, and G. P. Jones, editors, *Distributed Computing*, Academic Press, London, 1984.

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

[4] K. R. Apt and G. D. Plotkin, “Countable Nondeterminism and Random Assignment,” *J. of the ACM*, Vol. 33, No. 4, pp. 724-767, 1986.

[5] 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.

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

[7] 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.

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

[9] Arvind, R. S. Nikhil, and K. K. Pingali, “I-structures: Data Structures for Parallel Computing,” *ACM Transactions on Programming Languages & Systems*, vol. 11, no. 4, pp. 598-633, Oct, 1989.

[10] 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.

[11] 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.

[12] 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.

[13] A. Benveniste, P. Caspi, P. Le Guernic, N. Halbwachs, “Data-flow Synchronous Languages,” in J. W. de Bakker W.-P. de Roever, and G. Rozenberg, eds., *A Decade of Concurrency — Reflections and Perspectives*, Lecture Notes in Computer Science no. 803, Springer-Verlag, Berlin, 1994.

[14] G. Berry, “Bottom-Up Computation of Recursive Programs,” *Revue Française d’Automatique, Informatique et Recherche Opérationnelle*, vol. 10, no. 3, pp. 47-82, March, 1976.

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

[16] 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).

[17] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Static Scheduling of Multi-rate and Cyclo-Static DSP Applications”, *Proc. 1994 Workshop on VLSI Signal Processing*, IEEE Press, 1994.

[18] A. Bloss and P. Hudak, “Path Semantics,” in *Proc. of the Third Workshop on the Mathematical Foundations of Programming Language Semantics,* Lecture Notes in Computer Science No. 298, pp. 479-489, Springer-Verlag 1987.

guages. An example (a recursive specification of an FFT) is given above in figure 13. In situations where the recursion cannot be evaluated during the setup phase, as in the sieve of Eratosthenes in figure 12, it is much more difficult to exploit the parallelism at compile time.

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 [74], 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

We 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. We would also like to gratefully acknowledge helpful comments and suggestions from Albert Benveniste, Gerard Berry, Shuvra Bhattacharyya, John Reekie, Vason Srini, Juergen Teich, and the anonymous reviewers. 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.

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 [53]. Moreover, as indicated by Harrison [48], 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 [87] 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 [27], 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 [5] or, if the firing sequence is sufficiently predictable, a compiler [45][82][90][91].

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 [27], 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 statically evaluated higher-order functions enables the use of recursion *without compromising parallelism*. The recursion is evaluated during the setup phase, before the parallelizing scheduler is invoked. Thus, the scheduler sees only the fully expanded graph, not the recursion. It can fully exploit at compile time the parallelism in this graph. Thus, we regain much of the elegance that the use of recursion lends to functional lan-

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 data type 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 directly by sequencing the elements of the array, or by encapsulating each array into a single token, or by generalizing to multidimensional streams [65][92]. 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 data types, is supported in a natural way.

Hudak distinguishes two types of polymorphism, *parametric* and *ad-hoc* (or *overloading*) [53]. 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 programming 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 *distributor* in figure 13. The *Map* actor is also polymorphic, although in a somewhat more complicated way.

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 26. 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 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),

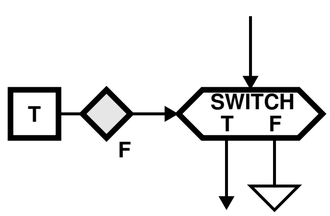


Figure 26. One way to implement a negative delay, which discards the first token that arrives on the input stream.

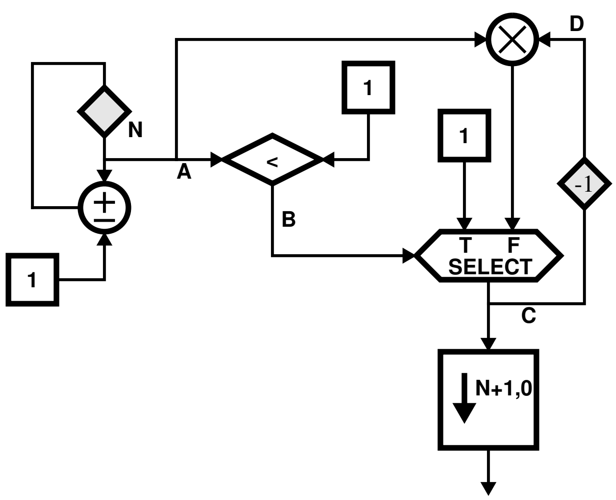
semantics are non-strict, and the *IfThenElse* actor can be used to implement recursion. The recursion is completely evaluated during the setup phase of execution (or at compile time), so the recursion imposes no run-time overhead during the run phase. This is analogous to the unrolling style of partial evaluation [34], and could be called *manifest recursion*.

The higher order functions above have a key restriction: the replacement actor is specified by a parameter, not by an input stream. Thus, we 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 [7][8] 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 25. This graph computes *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



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

provide the arguments for each instance of the specified actor. For example, if the replacement actor has two inputs, 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.

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 12 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 run-time operation is quite expensive, however, requiring dynamic creation of a dataflow graph. So there is still considerable motivation for recursion that can be statically unrolled, as done in figure 13. 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

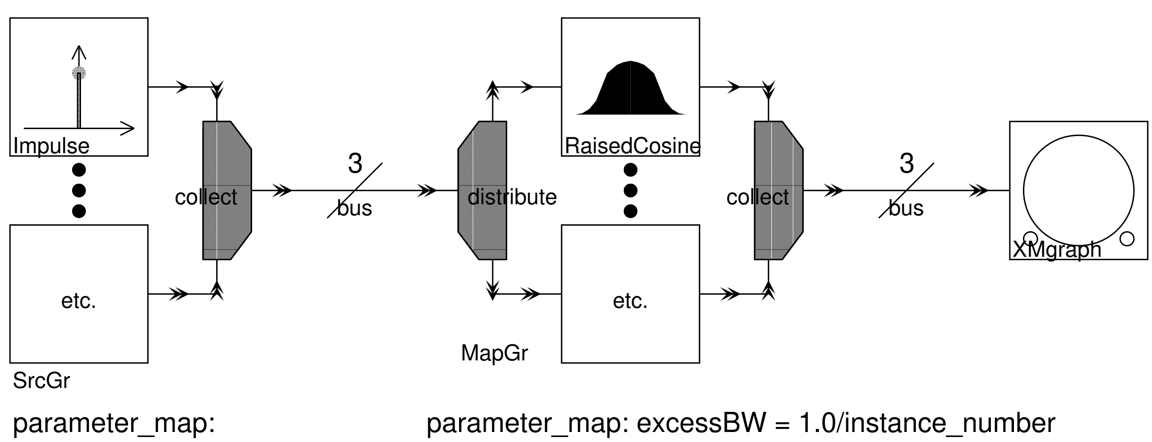


Figure 24. A program equivalent to that in figure 22 except that the replacement actors for the two higher-order actors are specified visually rather than textually.

be more easily modified to include more or fewer instances of the *RaisedCosine* actor. It is only necessary to modify the parameters of the bus icons, not the visual representation.

The left-most actor in figure 22 is a variant of the *Map* actor called *Src*. It has no inputs. In this case, the number of instances of the replacement actor that are created must match the number of *output* streams.

In the visual programming languages ESTL [75] and DataVis [51], higher-order functions use a "function slots" concept, visually representing the replacement function as a box inside the icon for the higher-order function. We have implemented in Ptolemy a conceptually similar visual representation. Variants of the *Map* and *Src* actors, called *MapGr* and *SrcGr*, have the icons shown in figure 23. It is important to realize that figure 23 contains only two icons, each representing a single actor. The complicated shape of the icon is intended to be suggestive of its function when it is found in a block diagram. The *MapGr* and *SrcGr* actors work just like the *Map* and *Src* actors, except that the programmer specifies the replacement block visually rather than textually. For example, the system in figure 22 can be specified as shown in figure 24. Notice that replacement actors *Impulse* and *RaisedCosine* each have one instance shown visually. The *MapGr* and *SrcGr* actors have only a single parameter, called *parameter_map*. The other parameters of *Map* and *Src* are now represented visually (the replacement block and input/output mapping).

Note that the same effect could be accomplished by tricks in the graphical user interface, as done for instance in GRAPE II [63]. However, this then requires modifying the GUI to support new capabilities.

A number of additional variations are possible. First, the replacement actor may have more than one input, in which case the input streams are grouped in appropriately sized groups to

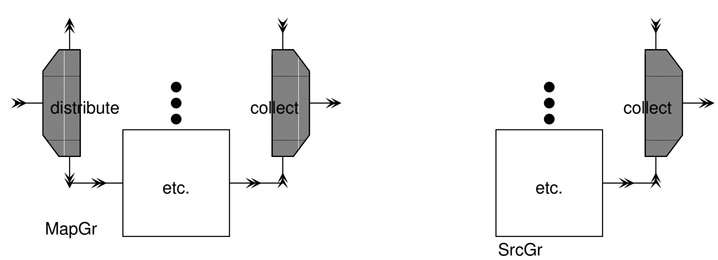


Figure 23. Icons for the MapGr and SrcGr higher-order functions in Ptolemy.

The program in figure 19 is equivalent to that in figure 21. Indeed, after the setup phase of execution, the topology of the process network will be exactly as in figure 21. The *Map* actor itself will not appear in the topology.

In both figures 19 and 21, the number of instances of the *RaisedCosine* actor is specified graphically. In figure 19, it is specified by implication, through the number of instances of the *Impulse* actor. In figure 21 it is specified directly. Neither of these really takes advantage of higher-order functions. The program in figure 22 is equivalent to both figures 19 and 21, but can

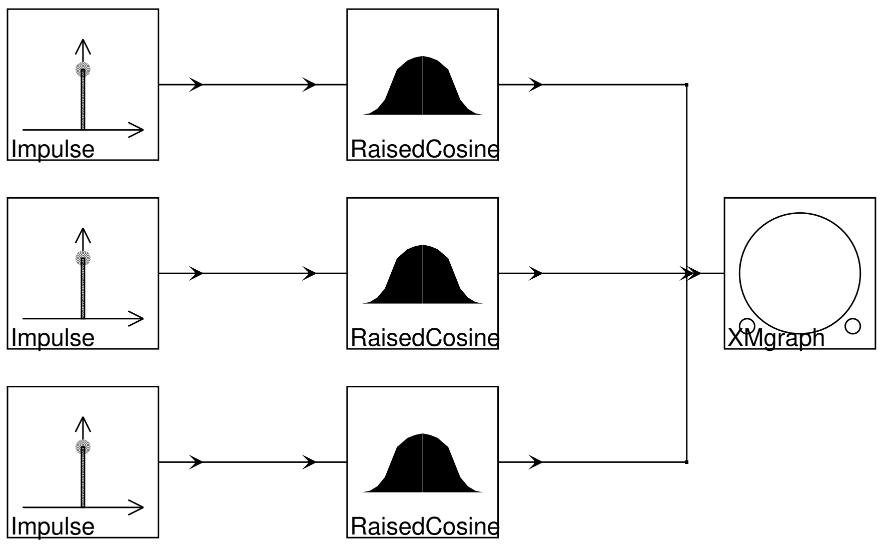


Figure 21. A program equivalent to that in figure 19, but without higher-order functions.

blockname: Impulse
where_defined:
parameter_map:
output_map: output

blockname: RaisedCosine
where_defined:
parameter_map: excessBW = 1.0/instance_number
input_map: signalIn
output_map: signalOut

Figure 22. A program equivalent to that in figures 19 and 21, except that the number of instances of the ***RaisedCosine*** **and** ***Impulse*** **actors can be specified by a parameter.**

The diagonal slash through the last connection on the right in figure 19 is a *Bus*. Its single parameter specifies the number of logical connections that the single visual connection represents. Here, the bus width is three. This must be so because there are three inputs to the *Map* actor, so three instances of the *RaisedCosine* actor will be created. The three outputs from these three instances need somewhere to go. The result of running this system is shown in figure 20.

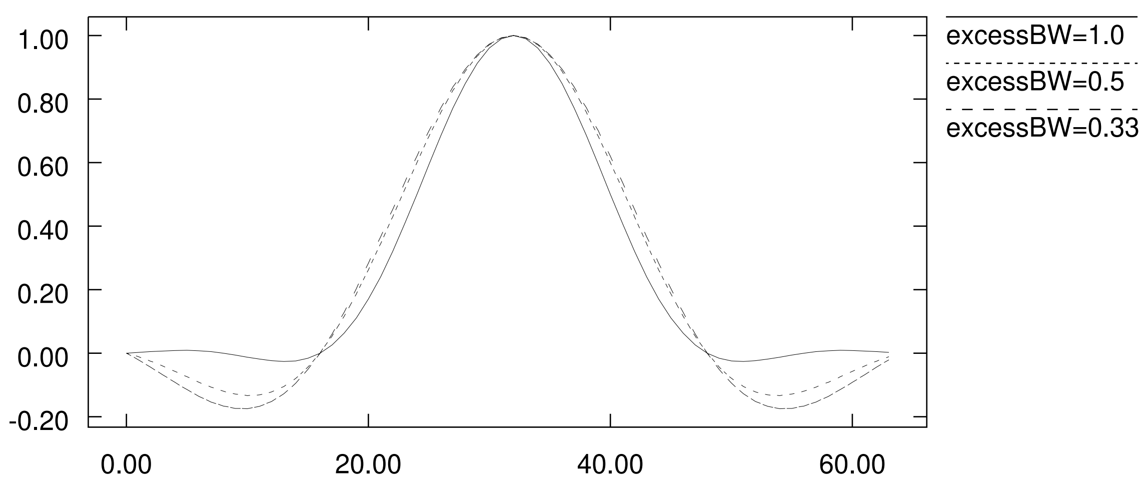


Figure 20. The plot that results from running the program in figure 19.

Our implementation of *Map* is simple but effective. It 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. This replacement can be viewed as a form of partial evaluation of the program [34].

Consider the example shown in figure 19. The replacement actor is specified to be *RaisedCosine*, a built-in actor in the signal processing environment in Ptolemy. Since this is built-in, there is no need to specify where it is defined, so the *where_defined* parameter is blank. The *RaisedCosine* actor has a single input named *signalIn* and a single output named *signalOut*, so these names are given as the values of the *input_map* and *output_map* parameters. The *parameter_map* parameter specifies the values of the *excessBW* parameter for each instance of the replacement block to be created. This parameter specifies the excess bandwidth of the raised cosine pulse generated by this actor. The value of the *excessBW* parameter will be 1.0 for the first instance of the *RaisedCosine* actor, 0.5 for the second, and 0.33 for the third.

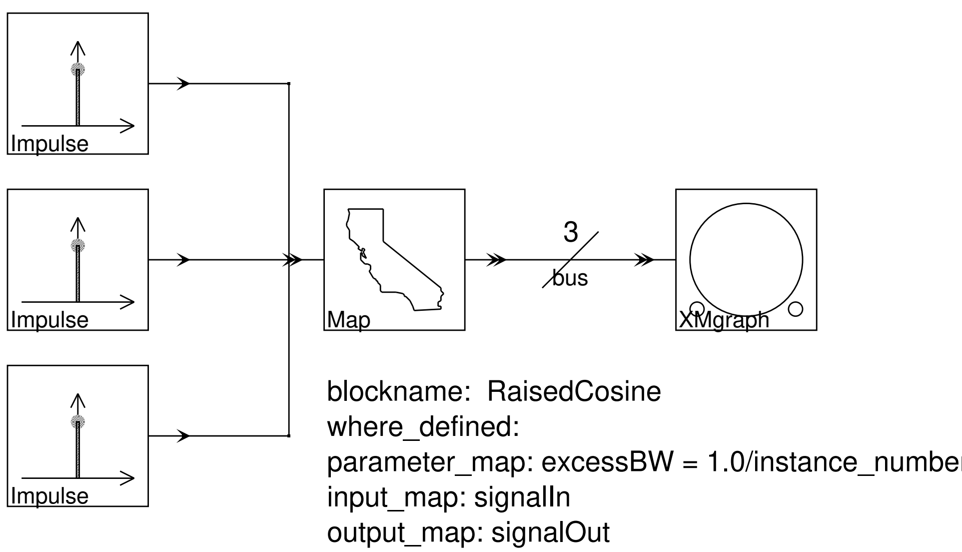


Figure 19. An example of the use of the *Map* actor to plot three different raised cosine pulses.

Recall our proposed syntactic sugar for representing feedback loops such as that in figure 17 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 implementing 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 [51], Najork and Golin [75], and Reekie [85]. We will draw on this work here.

We created an actor in Ptolemy called *Map* that generalizes the Haskell *map*. Its icon is shown in figure 18. It has the following parameters:

blockname	The name of the replacement actor.
where_defined	The location of the definition of the actor.
parameter_map	How to set the parameters of the replacement actor.
input_map	How to connect the inputs.
output_map	How to connect the outputs.

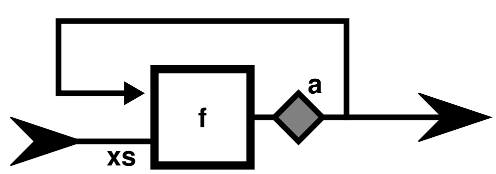


Figure 17. Visual syntax for the dataflow process network equivalent of the Haskell "scanl f a xs" higher-order function.

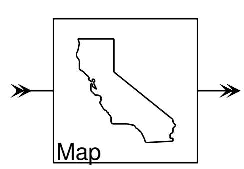


Figure 18. Icon for the Map higher-order function in Ptolemy.

often represent such processes using higher order functions. For example, in Haskell,

map *f xs*

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* takes two arguments. 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 17.

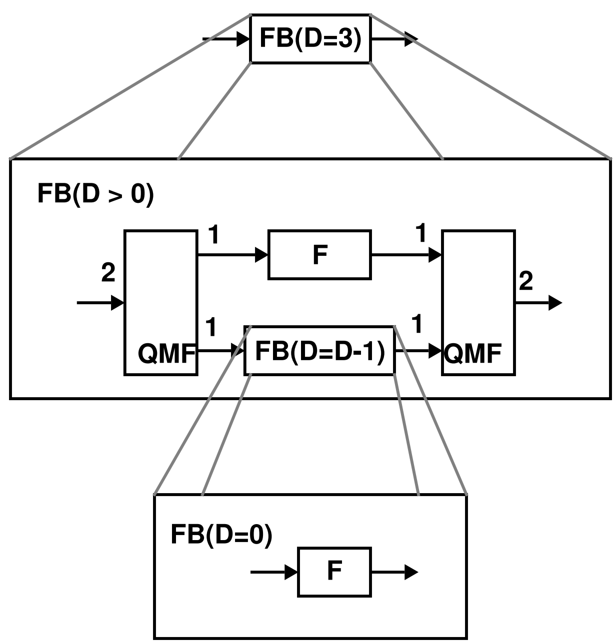


Figure 16. A recursive representation of the filter bank application. This representation uses template matching.

stream (such as quantization). The QMF boxes to the right of these reconstruct the signal using matching polyphase interpolating FIR filters.

There are four distinct sample rates in figure 15 with a ratio of 8:1 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 15 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 16 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 center. It contains a self-reference, with the parameter of the inside reference changed to D - 1. When D=0, the definition at the bottom is used. The system at the top, consisting of just one block, labeled "FB(D = 3)", is exactly equivalent to the representation in figure 15, except that the visual representation does not now depend on the depth. The visual recursion in figure 16 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

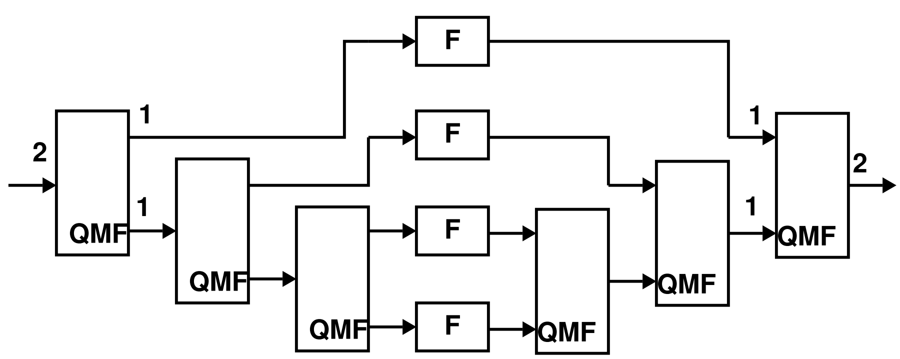


Figure 15. An analysis/synthesis filter bank under the SDF model. The depth of the filter bank, however, is hard-wired into the representation.

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 14. 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 13. Moreover, we argue that the visual representation in figure 13 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 13 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 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 15. 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” for “quadrature mirror filter”). These are decimating polyphase finite impulse response (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

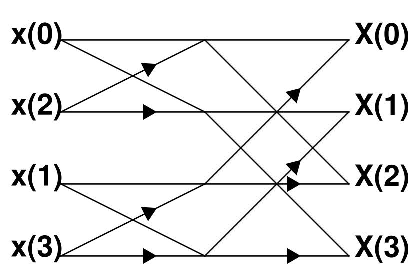


Figure 14. A fourth-order decimation-in-time FFT shown graphically. The order of the FFT, however, is hard-wired into the representation.

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

$$DFT_N(x(n)) = DFT_{N/2}(x(2n)) + W_N^k DFT_{N/2}(x(2n+1)) . \tag{34}$$

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

From this, we arrive at the recursive specification shown in figure 13. 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, representing 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, implementing an FFT of half the order. When the *order* parameter 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

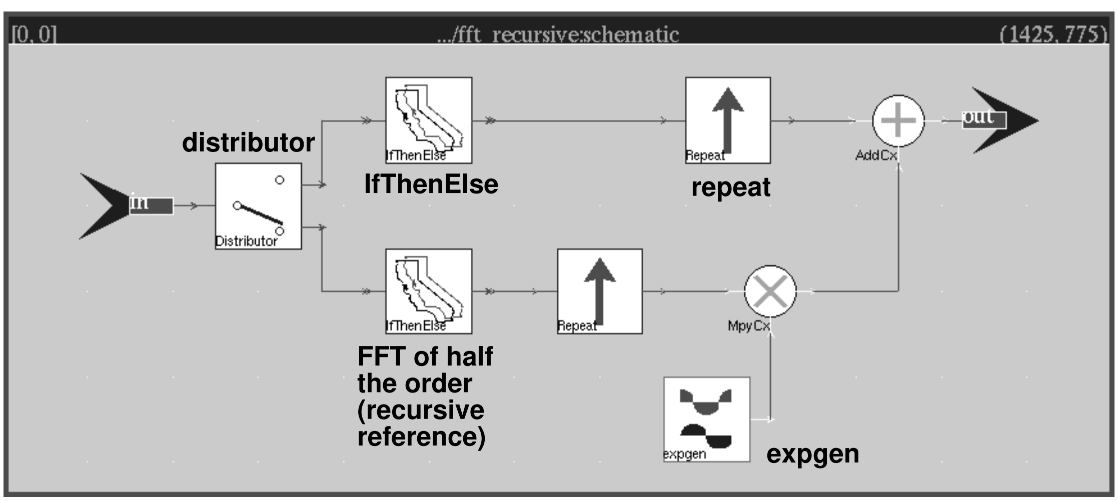


Figure 13. A recursive specification of an FFT implemented in the SDF domain in Ptolemy. The recursion is unfolded during the setup phase of the execution, so that the graph can be completely scheduled at compile time.

Queen [59] and in TLDF [94]. Mutability, however, considerably complicates compile-time analysis of the graph. The compile-time scheduling methods in [22] and [66] 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 synchronous 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

$$X_k = \sum_{n=0}^{N-1} x(n)\, e^{-j\left(\frac{2\pi}{N}\right)kn} \tag{29}$$

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

$$W_N = e^{-j\left(\frac{2\pi}{N}\right)} \tag{30}$$

and note the following properties:

$$W_N^2 = W_{N/2} \text{ and } W_N^{N+k} = W_N^k. \tag{31}$$

Using this we can write

$$X_k = \sum_{n=0}^{N-1} x(n)\, W_N^{kn} = \sum_{\substack{n=0 \\ n \text{ even}}}^{N-2} x(n)\, W_N^{kn} + \sum_{\substack{n=1 \\ n \text{ odd}}}^{N-1} x(n)\, W_N^{kn} \tag{32}$$

By change of variables on the summations, this becomes

$$X_k = \sum_{n=0}^{(N/2)-1} x(2n)\, W_{N/2}^{kn} + \left(\sum_{n=0}^{(N/2)-1} x(2n+1)\, W_{N/2}^{kn}\right) W_N^k. \tag{33}$$

direct expression of recursion is not yet supported by the Ptolemy graphical interface, although it is supported in the underlying kernel. Ptolemy implements this in a simple, and rather expensive way; it dynamically expands the graph when the recursive block is invoked. More efficient implementations are easy to image, however.

Note that recursion in figure 12 expresses a "mutable graph", in that the structure of the graph changes as the program executes. Such dynamics are also permitted by Kahn and Mac-

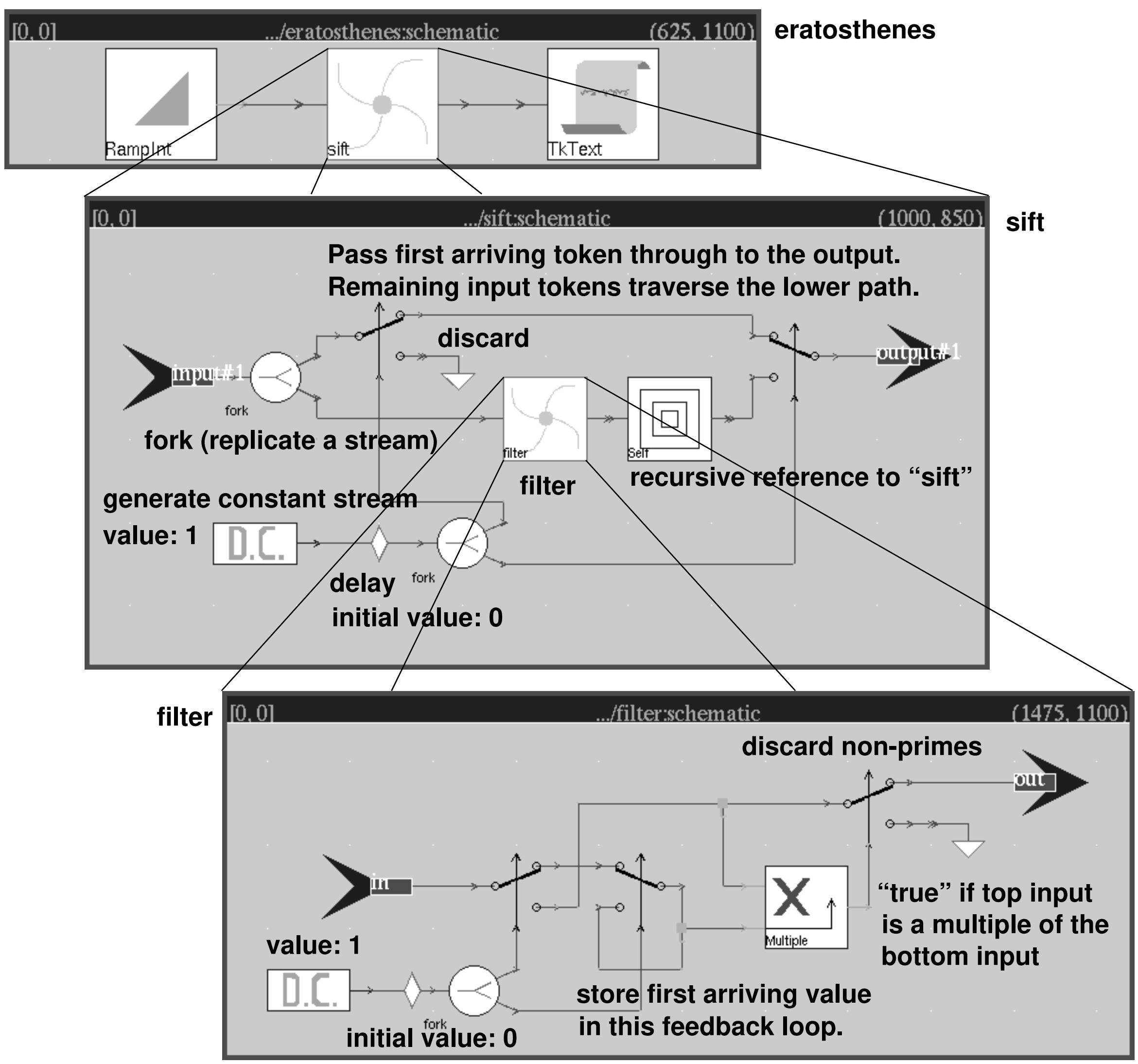


Figure 12. A recursive implementation of the sieve of Eratosthenes in the dynamic dataflow domain in Ptolemy. The top-level system (with just three actors) produces all the integers greater than 1, filters them for primes, and displays the results. Other icons are explained once each.

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, particularly if in-line code is generated. It also precludes incremental compilation of hierarchical nodes. But it appears to be necessary to support graphs like that in figure 11. At least one more sophisticated scheduler [16] 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.

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 7a and 7b. These feedback loops specify recurrence relations, but are not self-referential in the usual sense of recursion. Ida and Tanaka [55] and Abramsky [2] have also noted the advantages of this representation. 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 [59], among others. 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 by creating 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 [53].

A recursive implementation in the dynamic dataflow domain of Ptolemy is shown in figure 12. 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

The next natural question is whether hierarchical nodes should be strict. In particular, for those hierarchical nodes for which there exists a well-defined firing, should that firing be strict? The example shown in figure 11 suggests a definitive "no" for the answer. A hierarchical node A is composed of subprocesses B and C as shown in the figure. A firing of the expanded definition in figure 11b might consist of a firing of B followed by C. However, when connected as shown in figure 11a, the network deadlocks, quite unnecessarily, if we insist that the hierarchical node have both inputs available before firing.

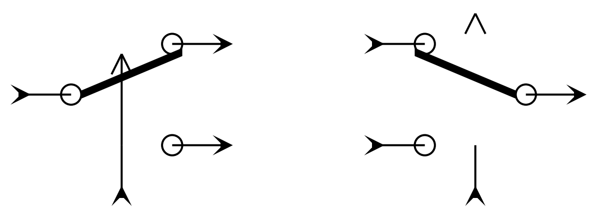


Figure 10. Switch and Select actors in the dynamic dataflow domains of Ptolemy. These are determinate actors that merge or split streams under the control of a Boolean stream.

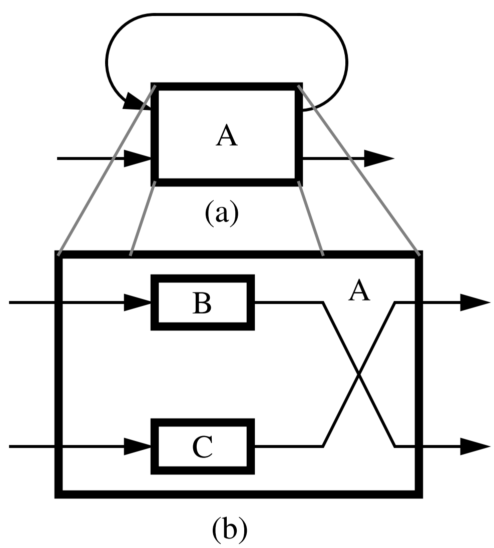


Figure 11. A hierarchical node A in a simple subnetwork (a) and its expanded definition (b). If the actor A is strict, the subnetwork in (a) deadlocks.

Referential transparency for source actors is also preserved, as long as the parameters are considered. Thus, the transformation shown in figure 6 is now possible only if the actors or subgraphs being consolidated have identical parameters. Thus, with these syntactic devices (actors with state, the notation $F = map\ (f)$, delays, and actors with parameters as well as inputs), referential transparency is still possible. We 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 f_1 function in section 2.4.4 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. Recall the *select* actor from figure 2a.

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

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

The firing rules implied by this definition are sequential, since a token is always 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 10. 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 in functional languages has been observed before, of course [53] (at least the if-then-else must be non-strict in the consequent and the alternative).

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, optimized code that takes advantage of the fact that the third harmonic distortion is set to zero (see figure 9) 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).

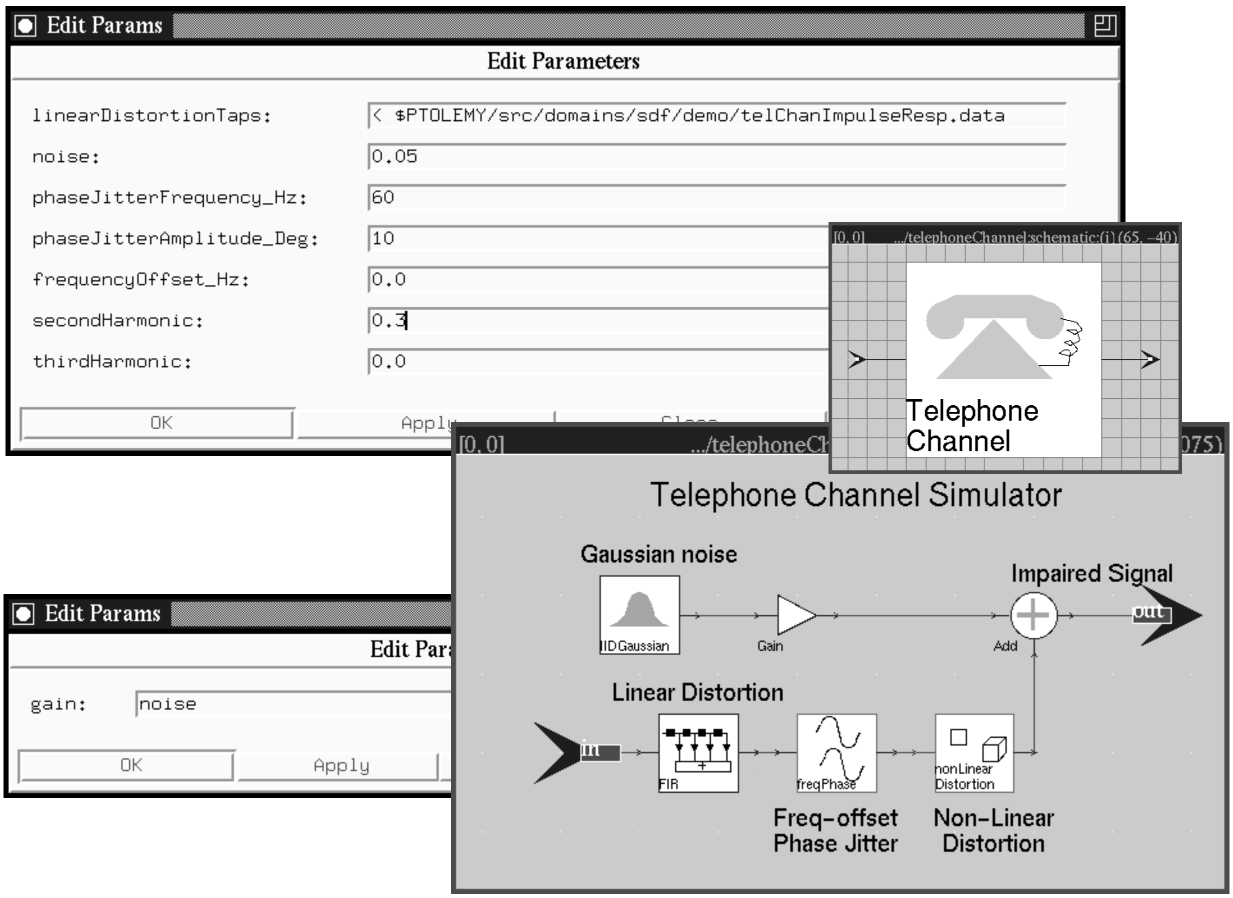


Figure 9. Top: A typical parameter screen in Ptolemy for a hierarchical node that models a telephone channel. The first parameter is given as a reference to a file. The icon for the node is shown to the right. The next level down in the hierarchy is shown in the lower right window. At the lower left, 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.

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 8 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 8 may be viewed as functions with two arguments, the multiplying constant and the input stream. But unlike functional languages, a clear syntactic distinction is made between *parameter arguments* and *stream arguments*. In functional languages, if the distinction is made at all, it is made through the type system. 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 9. This expression language has some of the trappings of 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 can make the job of a compiler or interpreter simpler, obviating the optimization step of identifying such static streams.

The basic observation is that internal state in a primitive or a hierarchical node is *syntactic sugar* (a convenient syntactic shorthand) for delays on 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.

If actors have state, the notation $F = map\ (f)$ is no longer directly valid. With a little adaptation, however, we can still use it. If we wish to model an actor with p inputs and q outputs, plus state, we can define $F: S^{p+1} \rightarrow S^{q+1}$ based on an actor function $f: S^{p+1} \rightarrow S^{q+1}$, where the extra argument carries the state from one firing to the next.

With this device, notice that the firing rules can now depend on the state. For example, in the cyclo-static dataflow model of Lauwereins, *et al*. [63], an actor can consume a cyclically varying number of tokens on an input. For instance, a dataflow process with one input and one output might consume one token on its odd-numbered firings and two tokens on even-numbered firings. In this case, a binary-valued state variable will have value zero on even-numbered firings and one on odd-numbered firings. Thus, the firing rules become

$$\mathbf{R}_1 = \{\ [0], [*]\ \} \tag{27}$$

$$\mathbf{R}_2 = \{\ [1], [*,*]\ \} \tag{28}$$

where the first argument is the state. Any cyclo-static actor can be modeled in this way. In fact, firing rules that change over the course of several firings can be modeled in the same way even if they do not vary cyclically, as long as the firing rules for the n^{th} firing can be determined during the $(n-1)^{th}$ firing.

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

where *scanl* is a higher order function with three arguments, a function, a number, and a list. It is defined as follows (taking certain liberties with Haskell syntax):

scanl f q $\perp$ = [q]

scanl f q (x:xs) = q : scanl f f (q,x) xs

These two definitions use template matching; the first is invoked if the third argument is an empty stream. The q first gives the initial value for the sum, equivalent to the value of the initial token in a delay, and later carries the running summation. The syntax (x:xs) divides a list into the first element (x) and the rest (xs). The syntax q : *expr* represents a list where q is the head and *expr* defines the rest, just as we have done above for sequences. For example,

scanl (+) 0 [1,2,3,4]

produces [0,1,3,6,10].

The program above uses recursion to carry state, via the higher-order function *scanl*. It has been observed that for efficiency this recursion must be translated into an iterative implementation [55][40][43]. For streams this is mandatory, since otherwise the depth of the recursion could become extremely large.

Delays in a hierarchical node can make a single firing of the node non-functional even if it is not in a feedback loop. Consider the example in figure 7a. The balance equations tell us that a complete cycle consists of one firing of A_1 and one of A_2. But under this policy, state will have to be preserved between firings on the arc connecting the two actors, making a firing of the hierarchical node non-functional.

Some of the problems with state 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 [89]. 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.

Side effects and state

Even when a hiearchical node has a complete cycle, a second problem arises in our attempt to define its mapping in terms of $F = map\ (f)$. Even if all actors within the node are functional, the hiearchical node may not be.

Consider the example in figure 7b. A single firing of actor B obviously defines a complete cycle. The feedback loop is used to implement a recurrence, so 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 even if f_B is. In this case, the feedback loop must be initialized with tokens in order to avoid deadlock.

The shaded diamond is called a *delay*, which is typically implemented as an initial token in the channel. It cannot be described by $F = map\ (f)$, where f is functional, but its behavior is easily defined by $F(X) = i : X$, where X is the input sequence, i is the initial token, and ":" is the concatenation operator. The initial token enables the first firing of actor B 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 any single firing of the hierarchical node non-functional.

The delay shown in figure 7b is typically implemented using the "cons" operator to initialize streams when streams are based on the recursive-cons model [62]. It is roughly equivalent to the "D" operator in the tagged-token model [8]. It is the visual equivalent of "fby" (followed-by) in Lucid [92] and the "pre" operator in Lustre [47]. In the single assignment language Silage, developed for signal processing [50], 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.

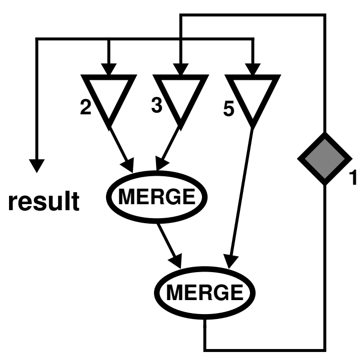
In functional languages, instead of using a recurrent construct like a delay, state is usually carried in the program using recursion. Consider, for example, the following Haskell program:

implementation that does not generate such duplications (and hence does not need to eliminate them) is given in figure 8b. It is also inconsistent. Neither of these can be implemented with bounded memory.

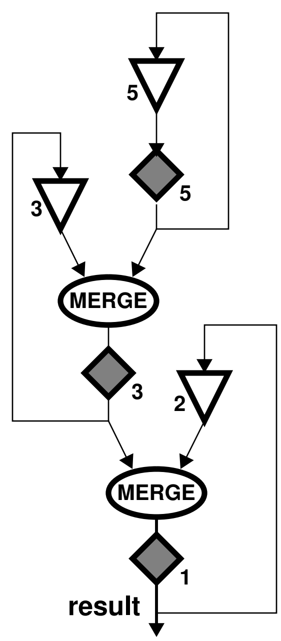
The second, more fundamental problem is that the existence of complete cycles for dynamic dataflow graphs is *undecidable* [22]. Thus, no algorithm will be able to identify a complete cycle for all graphs that have one.

A third problem is that the actors in a hierarchical node may not form a *connected* graph without considering as well the graph within which the hiearchical node sits. In this case, the balance equations for the hiearchical node alone will have more than one solution. There is no way to select among these solutions.

When a hiearchical node has a complete cycle that can be identified, then we may be able to define f to be the mapping performed by this complete cycle. In this case, $F = map\,(f)$ captures the behavior of the hiearchical node. Unfortunately, there are still difficulties.



(a)



(b)

Figure 8. 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.

of A_2 means that it consumes one token when it fires. A "reasonable" firing of the hierarchical node would therefore consist of one firing of A_1 and one of A_2. The single balance equation for this example is

$$r_{A_1} \times 1 = r_{A_2} \times 1 , \tag{23}$$

where r_{A_i} is the number of firings of A_i. This equation simply says that r_{A_i} should be such that the number of tokens produced on the arc should equal the number consumed, thus keeping it "in balance". Any "firing" of the hierarchical node that invokes the *i*-th actor r_{A_i} times (for all *i*) will therefore return the subsystem to its original state. 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.

A nonempty set of firings that returns a subsystem to its original state is called a *complete cycle* [64]. Unfortunately, three problems arise. First, some useful systems have balance equations with no solution [22][23]. Such systems are said to be *inconsistent*, or *unbalanced*, and have no complete cycle, and usually have unbounded memory requirements. A simplified (and probably not useful) example is shown in figure 7c. The balance equations for this subsystem are (one for each arc)

$$r_{C_1} \times 1 = r_{C_2} \times 1 , \tag{24}$$

$$r_{C_1} \times 1 = r_{C_3} \times 1 , \tag{25}$$

$$r_{C_2} \times 2 = r_{C_3} \times 1 . \tag{26}$$

These equations have no solution. Indeed, any set of firings of these actors will leave the subsystem in a new state.

To hint that unbalanced systems are sometimes 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 [42] and Kahn and MacQueen [59]. A dataflow implementation equivalent to the first of two by Kahn and MacQueen is shown in figure 8a. The "merge" block is an ordered merge [64]; given a nondecreasing sequence of input values on two streams, it merges them into a single stream of nondecreasing values, and removes duplicates. A more efficient

describe the behavior of a hierarchical node by $F = map\ (f\ )$, where f constitutes a single, functional firing of the hierarchical node. This is not always possible. Two problems arise: f may not be well defined, and when it is, it may not be functional. Note that no problem arises in defining the hierarchical node to be a mapping F from input sequences to output sequences. F will be functional if the actors in the hierarchial node have functional firings and sequential firing rules.

Firing subgraphs — the balance equations

Examples that have more than one actor, such as in figure 7a and 7c, raise 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 [64][66][67] 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 before and after the firing.

Consider the example in figure 7a. Following Lee and Messerschmitt [66], the "1" symbol next to the output of A_1 means that it produces one token when it fires. The "1" next to the input

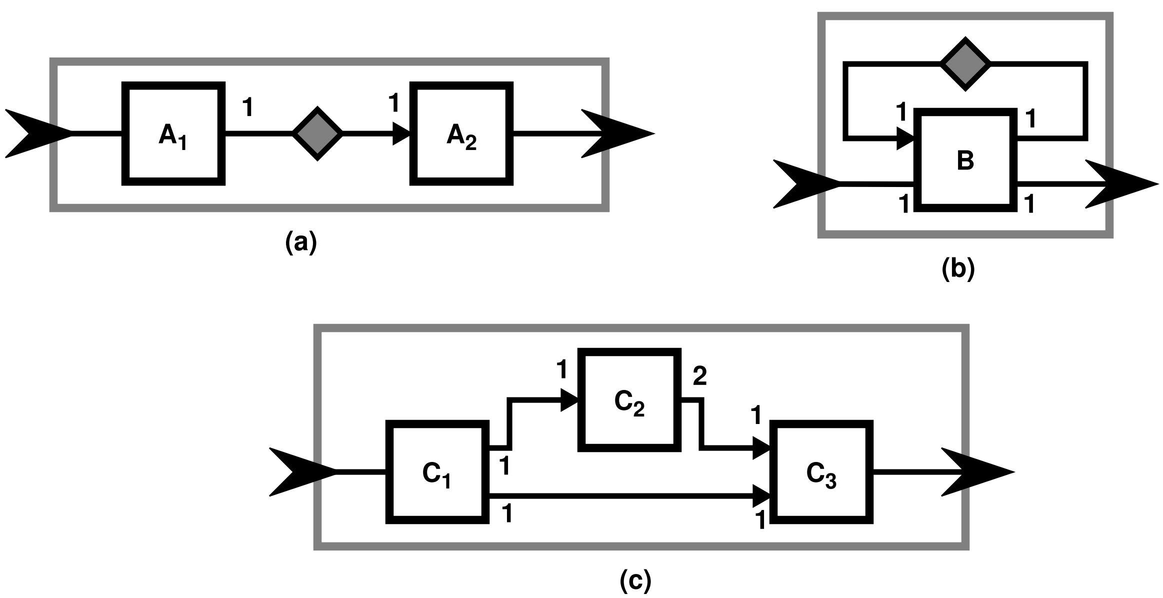


Figure 7. Hierarchical nodes in a dataflow process network may not be functional even if the primitives they contain are functional. The large arrowheads indicate input and output for the hierarchical node.

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 in section 2.4. 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 6 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 7. To reach the above ideal, we should be able to

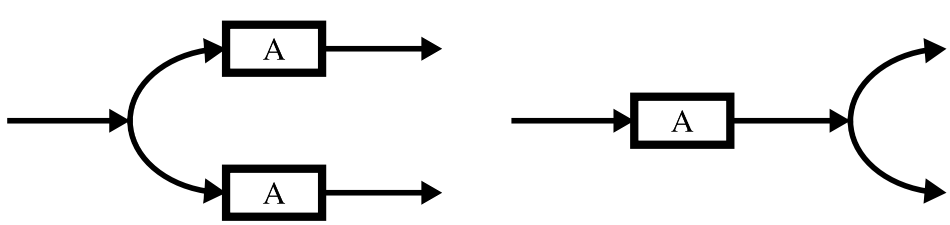


Figure 6. Referential transparency implies that these two dataflow process networks are equivalent.

- It does not have any model of computation built into the kernel, and hence can be used to experiment with different models of computation, and interactions between the models.
- Three dataflow process network “domains” have already been built in Ptolemy, precisely to carry out such experiments.
- 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 network 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) [66][67] is particularly well suited to signal processing [24], 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 run-time, 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 [22][64]. It resorts to run-time scheduling only when its analysis techniques break down. The *process network* domain (PN) uses a multi-tasking kernel to manage process suspension and resumption. It permits non-blocking reads, and hence allows nondeterminism.

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 [80].

3.2 Visual hierarchy — the analog to procedural abstraction

In keeping with the majority of signal processing programming environments, we 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

struct 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, we 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 [56]. Coordination languages with dataflow semantics are described by Suhler *et al.* [94], Gifford and Lucassen [44], Onanian [77], Printz [82], and Rasure and Williams [84]. Contrast these to the approach of Reekie [85] and the DSP Station from Mentor Graphics [41], where new actors are defined in a language with semantics identical to those of 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 is to explore the dataflow process network model fully, this flexibility is essential.

3.1 The Ptolemy system

To make the discussion concrete, we will use the Ptolemy software environment [25] to illustrate some of the trade-offs. It is well suited for several reasons:

- It has both a visual (“block diagram”) and a textual interface; the visual interface is similar in principle to many of those used in other signal processing software environments.

In addition to scheduling, efficient compilation requires that memory allocation be done statically, if possible. Despite the Kahn process network model of infinite FIFO channels, it is usually possible to construct bounded memory implementations with statically allocated memory for the channels [22]. Unfortunately, since the Boolean dataflow model is Turing complete, it is undecidable whether an arbitrary dataflow graph can be executed in bounded memory, so static memory allocation for the channels is not always possible. But for most programs, it is, so the cost of dynamically allocated memory for the channels only needs to be incurred when the static analysis techniques break down.

To address the same problems, Benveniste, et al., argue in [13] for the so-called synchronous approach to dataflow, where clocks are associated with tokens carried by the channels. A major part of the motivation is to guarantee bounded memory. There are other compelling advantages to this approach as well. The clocks impose a *total order* on tokens in the system, compared to the *partial order* specified in a process network. This makes it easy to implement, for example, a *determinate* merge operation. Viewed another way, actors can test their inputs for *absence* of data, something that would cause nondeterminism in process networks. However, the synchronous approach alone does not make the critical questions decidable. So further restrictions on a language are required if all programs are to be "executable" [13]. Moreover, one could argue that the total ordering in a synchronous specification is in fact an overspecification, reducing the implementation options. However, this can be at least partially ameliorated by *desynchronizing* the implementation, as explored by Mafeïs and Le Guernic [69].

2.5.5 The tagged-token model

An execution model developed by Arvind and Gostelow [7][8] 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 con-

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* [61][66][67]. 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 several domains in Ptolemy.

Balance equation methods have recently been extended to cover most dynamic dataflow graphs [22][64] and have been implement in the Boolean dataflow and CGC (code generation in C) domains in Ptolemy. However, Buck has shown that the addition of only the *select* actor of figure 2 and a *switch* actor (which routes input data tokens to one of two outputs under the control of a Boolean input) to the synchronous dataflow model is sufficient to make it Turing complete [22]. This means that one can implement a universal Turing machine using this programming model. It also means that many critical questions become undecidable. For this reason, Buck's methods cannot statically schedule all dynamic dataflow graphs. For Turing complete dataflow models, it is still necessary for some programs to have some responsibilities deferred to a run-time scheduler.

2.5.4 Compilation of dataflow graphs

The static schedules that emerge from Buck's Boolean dataflow scheduler are finite sequential representations of an infinite execution of a dataflow graph. Given such a schedule, the dataflow graph can be translated into a lean sequential program, a process we normally call compilation. (Parallel implementations are briefly discussed below in section 3.9).

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 than in the general case, 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 [59] and explicitly described by Franco, *et al*. [43]. 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 [5][93]. This association may be unfortunate, since the performance of such architectures has yet to prove competitive [49]. 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 [77], 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

2.4.7 Relationship to Kahn Process Networks

Dataflow process networks with sequential firing rules and functional actors are a special case of Kahn process networks. They construct a process F as a sequence *map* (f) of atomic actor invocations f. Instead of suspending a process on a blocking read or non-blocking write, processes 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.

There is still the cost of scheduling. However, for most programs, this cost can be entirely shifted to the compiler [66] [22]. While it is impossible to always shift all costs to the compiler [22], 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 over specify 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 [59]. 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

The question naturally arises whether there are non-sequential functions that are continuous (and thus guarantee determinacy). In fact, a rather trivial example of such a function is the identity function with two inputs,

$$f(X_1, X_2) = \{X_1, X_2\} \quad . \tag{21}$$

It is easy to see that it is not sequential (extending either input extends the output). It is also straightforward to prove that it is continuous. In order to define *F = map (f)*, we need a set of firing rules. A reasonable set of firing rules for the identity function is $\mathbf{R}_1 = \{ [*], \perp \}$ and $\mathbf{R}_2 = \{ \perp, [*] \}$. Even though these are the same firing rules used earlier for the nondeterminate merge, the identity function is clearly determinate. In this case *f* is continuous and *F = map (f)* is also continuous.

The question naturally arises whether the above theorem extends to continuous functions. That is, given that *f* is continuous, can we conclude that *F = map (f)* is continuous? The answer is no, as demonstrated by the following counterexample. Let *Y* be some non-empty finite sequence. Define

$$f(X_1, X_2) = \{ Y : X_1, Y : X_2 \}. \tag{22}$$

The colon ":" again means concatenation of two sequences. This function is similar to the identity function, with the simple difference that it prepends a prefix to each of two input sequences. It is easy to show that this is continuous. However, *F = map* (*f*) is not continuous if we use the firing rules we defined for the identity function. In fact, it is not monotonic, nor even functional. That is, for any input sequences X_1 and X_2, there is more than one possible output. This is because the function *f* produces a copy of the prefix on *both* outputs when it fires. On the output streams there can be any number of copies of the sequence *Y* inserted between tokens from the corresponding input stream.

Berry [14] has defined a class of functions called *stable functions* that may not be sequential but are always continuous. This class is not as broad as the class of continuous functions, but in certain circumstances, is easier to work with. But this is beyond the scope of this paper.

2.4.6 Sequential Processes

Vuillemin [95] has given a mathematical definition of sequential functions that is entirely consistent with the notion given here of sequential firing rules. Both our actor functions and the processes made from them are sequential in his sense. The definition and its relationship to continuity and monotonicity is summarized in figure 5.

A process $F : S^p \to S^q$ is *sequential* if it is continuous and if for any $\mathbf{X} = \{ X_1, X_2, \ldots , X_p \}$, there exists an i, $1 \le i \le p$, such that for any $\mathbf{X'}$ where $\mathbf{X} \sqsubseteq \mathbf{X'}$ and $X_i = X'_i$, $F(\mathbf{X}) = F(\mathbf{X'})$. This is intuitively easy to understand in the context of process networks if one considers $\mathbf{X'}$ to be simply a more evolved state of the input streams than $\mathbf{X}$. In other words, $\mathbf{X'}$ extends the streams in $\mathbf{X}$, except the one stream X_i, which is not extended. The process is sequential because it needs for the stream X_i to be extended before it can extend any output stream. Moreover, for any $\mathbf{X}$, there is an i such that the process needs X_i to be extended before it can extend the output. Notice that this definition of sequentiality can be applied just as easily to an actor function f as to a process $F = map(f)$. Given this, the following theorem is obvious.

Theorem If an actor function f has sequential firing rules, then the process $F = map(f)$ is sequential.

sequential ⇒ continuous ⇒ monotonic	
monotonic	$\mathbf{X} \sqsubseteq \mathbf{Y} \Rightarrow F(\mathbf{X}) \sqsubseteq F(\mathbf{Y})$.
continuous	$\forall\, \chi = \{ \mathbf{X}_0, \mathbf{X}_1, \ldots \}$ such that $\mathbf{X}_0 \sqsubseteq \mathbf{X}_1 \sqsubseteq \ldots$, $F(\sqcup \chi) = \sqcup F(\chi)$.
sequential	$\forall\, \mathbf{X} = \{ X_1, X_2, \ldots , X_p \}, \exists\, i, 1 \le i \le p$, such that $\forall\, \mathbf{Y} \sqsupseteq \mathbf{X}$ where $Y_i = X_i$, $F(\mathbf{Y}) = F(\mathbf{X})$, when F is continuous.

Figure 5. Summary of function class definitions and their relationships for the function $F : S^p \to S^q$.

lems 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 $F_4 = map\ (f_4)$ process. 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 we have been using firing rules. Consider the following Haskell example (with slightly simplified syntax):

fac 0 $\quad = 1$

fac n $\quad = n \times \mathrm{fac}(n\text{-}1)$

This defines a factorial function. If the argument is 0, the result is 1. If the argument is n, the result is $n \times \mathrm{fac}(n\text{-}1)$. 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 $F(X)$, where $F = map\ (fac\ )$ and X is a stream. Each firing of the actor can trigger the creation of new streams, so this process network is not static. We will consider more interesting recursive examples than this in considerable detail below, so we defer further discussion.

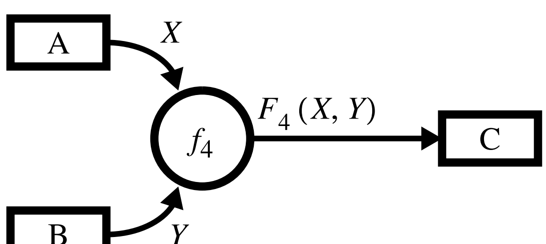


Figure 4. A variant of McCarthy's ambiguous function embedded in a dataflow process network.

the dataflow process made with actor function f_4. It is easy to show that the process is not monotonic. In fact, it is not even a function, since for some inputs, it can take on more than one possible output value. Consider $F_4(X_1, Y_1)$ and $F_4(X_2, Y_2)$ where

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

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

$$F_4(X_1, Y_1) \not\sqsubseteq F_4(X_2, Y_2) \ . \tag{18}$$

We get $F_4(X_1, Y_1) = [1]$, while $F_4(X_2, Y_2)$ 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 (and nonfunctional). 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 $F_4(X_1, Y_1) = [1]$ and $F_4(X_2, Y_2) = [2, 1]$. 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 we have made some implicit assumptions about synchronization between the input streams. To see this, consider the prefix ordered sequences

$$X_1 = [1], X_2 = [1], X_3 = [1, 1], \text{and } Y_1 = \perp, Y_2 = [2], Y_3 = [2] \ . \tag{19}$$

It would seem reasonable to argue that these are in fact exactly the same sequences as in (17). We are just looking at the value of the sequences more often. However, under the same implicit synchronization assumptions, the output is different:

$$F_4(X_1, Y_1) = [1], F_4(X_2, Y_2) = [2], F_4(X_3, Y_3) = [2, 1] \ . \tag{20}$$

These outputs are not prefix ordered, as they would be for a monotonic 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 simple; they are always enabled. They produce at their outputs the streams X and Y. The prob-

$f_1(x, y) = x$ or y chosen randomly

These three declarations define the output of the f_1 function under three firing rules: $\mathbf{R}_1 = \{ [*], \perp \}$, $\mathbf{R}_2 = \{ \perp, [*] \}$ and $\mathbf{R}_3 = \{ [*], [*] \}$. A dataflow process could be constructed by repeatedly firing this function on stream inputs.

McCarthy points out that the expression $f_1(1, 2) + f_1(1, 2)$ could take on the value 3, and uses this to argue that nondeterminism implies a loss of referential transparency[1]. However, 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:

$f_2(x, y) = x$ or y chosen randomly

is sufficient for $f_2(1, 2) + f_2(1, 2)$ 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:

$f_3(x, y, s) = x$ or y chosen by generating a random number from seed s.

Suddenly, we regain referential transparency and determinacy. It would not be possible for $f_3(1, 2, 3) + f_3(1, 2, 3)$ to equal 3, for example. Without giving the seed as an argument, f_3 is not functional.

Consider the simplified definition:

$f_4(x, \perp) = x$

$f_4(\perp, y) = y$

$f_4(x, y) = y$

This definition has no random numbers in it, but in a dataflow process network, it is still possible for $f_4(1, 2) + f_4(1, 2)$ 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 an attempt to construct a dataflow process from the function f_4 yields a process that is not monotonic, and hence is not continuous. Let $F_4 = map(f_4)$ represent

1. A basic notion used in the lambda calculus [32], referential transparency means that any two identical expressions have identical values. If $f_1(1,2)+f_1(1,2)=3$, then clearly the two instances of $f_1(1,2)$ cannot have taken on the same value.

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 [26] and Reekie [85]. 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 [71]. We define $F = map\ (f\,)$, where $f : S^p \to S^q$ is a function, to return a function $F : S^p \to S^q$ that applies f to each element of a stream when one of a set of firing rules is enabled. More precisely, $F = map\ (f\,)$, where

$$F(\mathbf{R}{:}\mathbf{X}) = f(\mathbf{R}) : F(\mathbf{X}) \tag{16}$$

and $\mathbf{R}$ is any firing rule of f. The colon ":" indicates concatenation of sequences. That is, if $\mathbf{X}$ and $\mathbf{Y}$ are each in S^p, then $\mathbf{X}{:}\mathbf{Y}$ is a new set of sequences formed by appending each sequence in $\mathbf{Y}$ to the end of the corresponding sequence in $\mathbf{X}$. Following the notation in Haskell, (16) defines the sequences returned by F when the input sequences have $\mathbf{R}$ as a prefix.

Notice that definition (16) is recursive. The recursion terminates when the argument to F no longer has any firing rule as a prefix.

The function f will typically require only some finite number of tokens on each input, while the function returned by $map\ (f\,)$ can take infinite stream arguments. Thus, $F = map\ (f\,)$ is a dataflow process, where each firing consists of one application of the dataflow actor function 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 [72] and used by Hudak [53]:

$$f_1(x, \bot) = x$$

$$f_1(\bot, y) = y$$

figure 3b, a blocking read of the bottom input will never unblock. In both cases, the behavior is 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 does not fail[1]:

1. Find an input j such that $[*] \sqsubseteq R_{i,j}$ 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 $R_{i,j}$ for any $i = 1, ..., N$. If $R_{i,j} = [*, ...]$, then the firing rule R_i should appear in all such subsets.
3. Remove the first element of $R_{i,j}$ 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

1. $j = 3$.
2. The firing rules divide into two sets, { $\mathbf{R}_1$ } and { $\mathbf{R}_2$ }, each with only one rule.
3. The new firing rules become $\mathbf{R}_1 = \{ [*], \perp, \perp \}$ in the first subset and $\mathbf{R}_2 = \{ \perp, [*], \perp \}$ in the second subset.

1. In (8), we imply that the number of firing rules is finite. John Reekie has pointed out in a personal communication that if we relax this constraint, then for some sequential firing rules corresponding to determinate actors, this procedure will not fail, but will also never terminate. Thus, as a practical matter, we may need the additional restriction that the procedure terminate. His example is an actor with two inputs, one of which is an integer specifying the number of tokens to consume from the other. The firing rules take the form { {[0], ⊥ }, {[1], [*]}, {[2], [*, *]}, ... }.

$$\mathbf{R}_1 = \{ [*], \bot, [T] \} \tag{12}$$

$$\mathbf{R}_2 = \{ \bot, [*], [F] \} \tag{13}$$

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, also has two firing rules

$$\mathbf{R}_1 = \{ [*], \bot \} \tag{14}$$

$$\mathbf{R}_2 = \{ \bot, [*] \}. \tag{15}$$

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

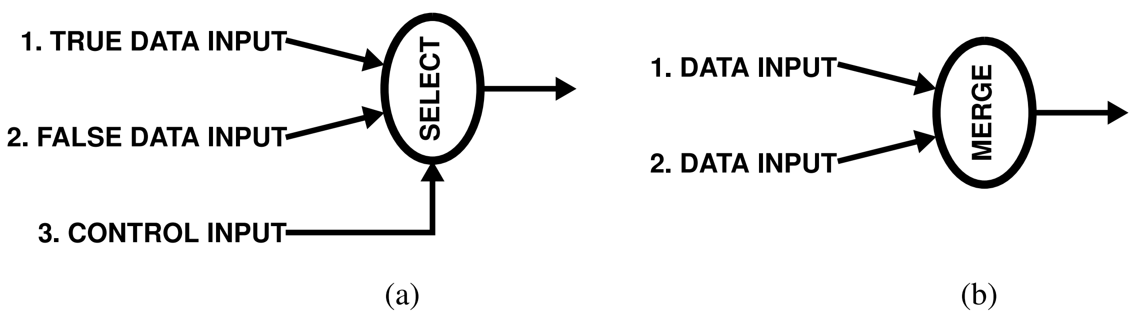


Figure 2. The select and nondeterminate merge actors each combine two data streams into one, but the select actor uses a Boolean control signal to determine how to accomplish the merge.

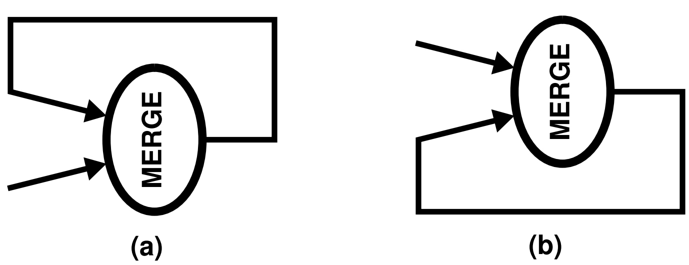


Figure 3. Illustration that the firing rules of the nondeterminate merge are not sequential. A blocking read of either input will cause one of these two networks to deadlock inappropriately.

A pattern $R_{i,j}$ is a (finite) sequence. For firing rule i to be satisfied, each pattern $R_{i,j}$ must form a prefix of the sequence of unconsumed tokens at input j . An actor with $p = 0$ input streams is always enabled.

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

To accommodate the usual dataflow firing rules, we need a generalization of the prefix ordering algebra. The 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 $[*] \sqsubseteq X$ is *not* saying that any one-token sequence is a prefix of X. All it says is that X has at least one token.

Let A_j, for $j = 1, ..., p$, denote the sequence of available unconsumed tokens on the j^{th} input. Then the firing rule $\mathbf{R}_i$ is enabled if

$$R_{i,j} \sqsubseteq A_j \text{, for all } j = 1, ..., p \ . \tag{10}$$

We can write condition (10) using the partial order on sets of sequences

$$\mathbf{R}_i \sqsubseteq \mathbf{A} \tag{11}$$

where $\mathbf{A} = \{A_1, A_2, ..., A_p\}$.

For many actors, the firing rules are very simple. Consider an adder with two inputs. It has only one firing rule, $\mathbf{R}_1$ = { [*], [*] }, meaning that each of the two inputs must have at least one token. More generally, synchronous dataflow actors [66][67], always have a single firing rule, and each pattern in the firing rule is of the form [*, *, ... , *], with some fixed number of wildcards. 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, which has the firing rules { $\mathbf{R}_1$, $\mathbf{R}_2$ }, where

1. An SDF actor also produces a fixed number of tokens when it fires, but this is not captured in the firing rules. An interesting variant, called *cyclo-static dataflow* [17], permits the number of tokens produced and consumed to vary cyclically. Modeling this with firing rules requires a straightforward generalization. We will give this generalization below in section 3.2.2.

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 call a *dataflow process*. A network of such processes is called a *dataflow process network*.

More specialized dataflow models, such as Dennis' static dataflow [39] or synchronous dataflow [66][67] can be described in terms of dataflow processes. The models used by most signal processing environments mentioned above can also be described in terms of dataflow processes. The tagged token model of Arvind and Gostelow [7][8] is related, but not identical, as we will show. Signal [12] and Lustre [46], which are called "synchronous dataflow languages," do not form dataflow processes at all because they lack the FIFO queues of the communication channels. They can, however, be implemented using dataflow process networks, with certain benefits to parallel implementation [69].

A sufficient condition for a dataflow process to be continuous, as defined in equation (1), is that each actor firing be *functional*, and that the set of firing rules be *sequential*. Here, "functional" means that an actor firing lacks side effects and that the output tokens are purely a function of the input tokens consumed in that firing. This condition is stronger than the Kahn condition that a *process* be functional, meaning that the output *sequences* are a function of the input *sequences*[58]. With Kahn's condition, actors can have and manipulate state. We later relax this constraint so that actors can have and manipulate state as well. "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 $p \geq 1$ input streams can have N firing rules

$$\mathcal{R} = \{ \mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_N \}. \tag{8}$$

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

$$\mathbf{R}_i = \{ R_{i,1}, R_{i,2}, \ldots, R_{i,p} \} \ . \tag{9}$$

In [85][86][87], Reekie *et al.* consider the problem of supporting streams in the functional programming language Haskell [53]. 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 classify 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 we 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 [87]. They propose eager evaluation whenever strictness analysis [54] 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 [57]. 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" [66][67]), 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, we can establish a clear relationship between dataflow, functional languages, and Kahn process networks.

Streams can be generalized to higher dimensionality, as done in Lucid [92] and Ptolemy [65][31]. 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

the program executes in bounded memory [30]. A different solution to the same problem is given by Buck [22], who uses the so-called balance equations, described below in section 2.5.3.

The difference between the two models for streams need not be important in practice, except that the choice of model may lead to unfortunate choices in language design. We prefer the channel model 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 suggest 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 [22][66]. The same objectives are addressed by *path analysis*, used to reduce the cost of lazy evaluation in functional languages through compile-time analysis [18].

In the channel model for streams, unlike the streams in the synchronous languages 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, as done in early dataflow schema [36].

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 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.

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 [59] 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 [55]. 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 [40]. Franco, *et al.* also argue in [43] 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 [50] blends the benefits of a declarative style with the simplicity of 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 [37], and in the Hyper project at U. C. Berkeley [83]. The use of a global cycle in a process network context has also been studied by Boussinot [20], who observes that it permits suspension and interruption of processes in a predictable way.

A more general approach is to associate with each stream a "clock," as done in Lustre [46] and Signal [12]. A clock is a logical signal that defines the alignment of 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 [12][68]. Caspi has described a preliminary attempt to abstract the notion of clocks so that it applies to process networks [29]. He has applied this abstraction to the Lucid language to solve certain problems like determining whether

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 network 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 monotonic process is clearly determinate. Since a network of monotonic processes is monotonic [78], then a network of monotonic 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 [11]. Skillcorn [92] 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 [97] 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 usually treats them functionally using lazy semantics. This view is apparently originally due to Landin [62]. Lazy semantics ensure that the entire stream need not be produced before its consumer operates on it. For example, Burge [26] 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 mechanism that avoids laziness is the so-called *I-structures* used in some dataflow languages [9].

PGM specification [60] in the form of what are called “graph variables.” A similar use of shared variables with “peek” and “poke” nodes appears in [79].

If B and C share a variable as described above, then they are potentially nonmonotonic. 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 monotonic 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 [78]). 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 has been shown that a nondeterminate merge must be either unfair or nonmonotonic, and hence not continuous [21]. Although rather involved technically, *unfair* intuitively means that it favors one input or the other.

Arvind and Brock [6] 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. Abramsky [2] points out that the functionality of a nondeterminate merge is widely used in practice in time-dependent systems, despite unsatisfactory formal methods for reasoning about it.

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.

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 over specifying 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.

Nondeterminism can be added to Kahn networks by any of five methods: (1) allowing processes to test inputs for emptiness, (2) allowing processes to be internally nondeterminate, (3) allowing more than one process to write to a channel, (4) allowing more than one process to consume data from a channel, and (5) allowing processes to share variables. Boussinot argues that (3) can implement (1) and (2), and gives the semantics of such extended process networks [19]. Shared variables, however, form a particular pitfall in a coordination language, since they are so easy to implement using the host language.

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

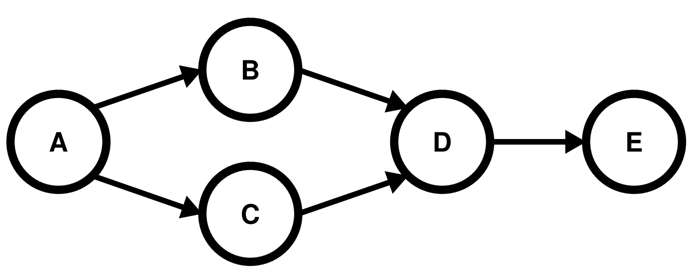


Figure 1. This process network does not specify the relative timing of the processing in nodes B and C. If D is a nondeterminate merge, it does not specify in which order the results should appear at E.

$$F(X) = \begin{cases} [0]; & \text{if } X \text{ is a finite sequence} \\ [0, 1]; & \text{otherwise} \end{cases} \quad . \tag{4}$$

Only two outputs are possible, both finite sequences. To show that this is monotonic, note that if the sequence X is infinite and $X \sqsubseteq X'$, then $X = X'$, so

$$Y = F(X) \sqsubseteq Y' = F(X') \ . \tag{5}$$

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 increasing chain

$$\chi = \{ X_0, X_1, \ldots \}, \text{ where } X_0 \sqsubseteq X_1 \sqsubseteq \ldots , \tag{6}$$

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

$$F(\sqcup \chi) = [0,1] \neq \sqcup F(\chi) = [0]. \tag{7}$$

Iterative computation of this function is clearly problematic.

A useful property is that a network of monotonic processes itself defines a monotonic process. This property is valid even for process networks with feedback loops, as is formally proven using induction by Panagaden and Shanbhogue [78]. It should not be surprising given the results so far that one can formally show that networks of monotonic 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. The problems arise because, as shown by Apt and Plotkin [4], nondeterminism leads to failures of continuity.

cessing, it provides a useful abstract analog to causality that works for multirate discrete-time systems without requiring the invocation of continuous time. Given an increasing chain χ, a monotonic process will map this set into another increasing chain Ψ.

For completeness, we now prove Kahn's claim that a continuous process is monotonic [58]. To do this, we prove that if a process is not monotonic, then it cannot be continuous. If the process F is not monotonic, then there exist $\mathbf{X}$ and $\mathbf{X'}$ where $\mathbf{X} \sqsubseteq \mathbf{X'}$, but $F(\mathbf{X}) \not\sqsubseteq F(\mathbf{X'})$. Let $\chi = \{ \mathbf{X}_0 \sqsubseteq \mathbf{X}_1 \sqsubseteq \ldots \}$ be any increasing chain such that $\mathbf{X}_0 = \mathbf{X}$ and $\sqcup \chi = \mathbf{X'}$. Then note that $F(\sqcup \chi) = F(\mathbf{X'})$. But this cannot be equal to $\sqcup F(\chi)$ because $\mathbf{X} \in \chi$ and $F(\mathbf{X}) \not\sqsubseteq F(\mathbf{X'})$. This concludes the proof.

A key consequence of these properties is that a process can be computed iteratively [70]. This means that given a prefix of the final input sequences, it may be 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. If we let $\mathbf{X}$ denote all the sequences in the network, including the outputs, and $\mathbf{I}$ the set of input sequences, then a network of functional processes can be represented by a mapping F where

$$\mathbf{X} = F(\mathbf{X}, \mathbf{I}) \tag{3}$$

Any $\mathbf{X}$ that forms a solution is called a *fixed point*. Kahn argues in [58] that continuity of F implies that there will be exactly one "minimal" fixed point (where minimal is in the sense of prefix ordering) for any inputs $\mathbf{I}$. Thus, we can get an execution of the network by first setting $\mathbf{I} = \perp$ and finding the minimal fixed point. Other solutions can then be found from this one by iterative computation, where the inputs are gradually extended; this works because of the monotonic property.

Note that continuity implies monotonicity, but not the other way around. One process that is monotonic but not continuous is $F: S \to S$ given by

empty sequence is denoted $\perp$ (*bottom*), and is obviously a prefix of any other sequence. Consider a (possibly infinite) increasing chain of sequences $\chi = \{ X_0, X_1, \ldots \}$, where $X_0 \sqsubseteq X_1 \sqsubseteq \ldots$. Such an increasing chain of sequences has one or more upper bounds Y, where $X_i \sqsubseteq Y$ for all $X_i \in \chi$. The *least upper bound* $\sqcup \chi$ is an upper bound such that for any other upper bound Y, $\sqcup \chi \sqsubseteq Y$. The least upper bound may be an infinite sequence.

Let S denote the set of finite and infinite sequences. This set is a *complete partial order* (*cpo*) with the prefix order defining the ordering. The "complete" simply means that every increasing chain has a least upper bound in S. Let S^p denote the set of p-tuples of sequences as in $\mathbf{X} = \{ X_1, X_2, \ldots, X_p \} \in S^p$. The set $\perp \in S^p$ is understood to be the set of empty sequences.

Such sets of sequences can be ordered as well; we write $\mathbf{X} \sqsubseteq \mathbf{X'}$ if $X_i \sqsubseteq X'_i$ for each i, $1 \leq i \leq p$. A set of p-tuples of sequences $\chi = \{ \mathbf{X}_0, \mathbf{X}_1, \ldots \}$ always has a *greatest lower bound* $\sqcap \chi$ (possibly $\perp$), but it may or may not have a *least upper bound* $\sqcup \chi$. If it is an increasing chain, $\chi = \{ \mathbf{X}_0, \mathbf{X}_1, \ldots \}$, where $\mathbf{X}_0 \sqsubseteq \mathbf{X}_1 \sqsubseteq \ldots$, then it has a least upper bound, so S^p is a cpo for any integer p.

A *functional process* $F : S^p \rightarrow S^q$ maps a set of input sequences into a set of output sequences. Given an increasing chain of sets of sequences χ, it will map this set into another set of sequences Ψ that may or may not be an increasing chain. Let $\sqcup \chi$ denote the least upper bound of the increasing chain χ. Then F is said to be *continuous* if for all such chains χ, $\sqcup F(\chi)$ exists and

$$F(\sqcup \chi) = \sqcup F(\chi). \tag{1}$$

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

$$\sqcup \chi = \sqcup \{ \mathbf{X}_0 \sqsubseteq \mathbf{X}_1 \sqsubseteq \ldots \} = \lim_{i \to \infty} \mathbf{X}_i. \tag{2}$$

Kahn sketches a proof that networks of continuous processes have a more intuitive property called *monotonicity* [58]. A process F is monotonic if $\mathbf{X} \sqsubseteq \mathbf{X'} \Rightarrow F(\mathbf{X}) \sqsubseteq F(\mathbf{X'})$. This can be thought of as a form of causality, but one that does not invoke time. Moreover, in signal pro-

tions of silicon implementations [37]. Often, considerable effort is put into optimized compilation (see for example [15][41][81][88]).

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 [52] or SCCS [74]. A better (and fortunately much simpler) formal underpinning is the Kahn process network [58].

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 $X = [x_1, x_2, \ldots]$, where each x_i is an atomic data object, or *token* drawn from some set. 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, we 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, $[x_1, x_2] \sqsubseteq [x_1, x_2, x_3]$. If $X \sqsubseteq Y$, it is common to say that X *approximates* Y, since it provides partial information about Y. The

flow languages for other applications is given by Hills [51]. These software environments all claim variants of dataflow semantics, but a word of caution is in order. The term “dataflow” is often used loosely for semantics that bear little resemblance to those outlined by Dennis in 1975 [38] or Davis in 1978 [35]. 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, we also show that such languages can make 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 [76], 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 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 [47], we call the language used to define the subprograms in nodes the *host language*. Following Jagannathan, we call the language defining the interaction between nodes the *coordination language* [56].

Many possibilities have been explored for precise semantics of coordination languages, including for example the computation graphs of Karp and Miller [61], the synchronous dataflow graphs of Lee and Messerschmitt [66], the cyclo-static dataflow of Lauwereins, *et al*. [63][17], the Processing Graph Method (PGM) of Kaplan, *et al.* [60], Granular Lucid [56], and others [3][28][33][56][94]. 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 [80], or even specifica-

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 of data. 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. A few graphical programming environments allow an arbitrary mixture of visual and textual specification, both based on the same language. For example, the Signal [12][68], Lustre [46], and Silage [50] languages all have a visual and a textual syntax, the latter available in the commercial Mentor Graphics DSP Station as DFL. Other languages with related semantics, such as Sisal [73], are used primarily or exclusively with textual syntax. The language Lucid [92][96], while primarily used with textual syntax, has experimental visual forms [10].

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 [84] (now distributed by Khoral Research, Inc.), Ptolemy, from the University of California at Berkeley [25], 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). MATLAB from The MathWorks, which is popular for signal processing and other applications, has a visual interface called SIMULINK. A survey of graphical data-

DATAFLOW PROCESS NETWORKS

Edward A. Lee
Thomas M. Parks

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

Published in *Proceedings of the IEEE*, May, 1995.

ABSTRACT

We review a model of computation used in industrial practice in signal processing software environments and experimentally in other contexts. We give this model the name "dataflow process networks," and study 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, among many others.

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 (often 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.

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

This research is part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), Semiconductor Research Corporation (project 94-DC-008), National Science Foundation (MIP-9201605), Office of Naval Technology (via Naval Research Laboratories), the State of California, and the following companies: Bell Northern Research, Dolby, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, Rockwell, Sony, and Synopsys.