[29] E. A. Lee and T. M. Parks, “Dataflow Process Networks,” *Proceedings of the IEEE*, May 1995. (http://ptolemy.eecs.berkeley.edu/papers/processNets)

[30] D. G. Messerschmitt, “Synchronization in Digital System Design,” *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 8, pp. 1404-1419, October 1990.

[31] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, NJ, 1989.

[32] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, “Software Synthesis for DSP Using Ptolemy,” *Journal on VLSI Signal Processing*, Vol. 9, No. 1, pp. 7-21, Jan., 1995. (http://ptolemy.eecs.berkeley.edu/papers/jvsp_codegen).

[33] E. L. Post, “Formal Reductions of the General Combinatorial Decision Problem,” *Am. J. Math.*, Vol. 65, pp. 197-215, 1943.

[34] 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.

[35] S. Ritz, M. Pankert, and H. Meyr, “High Level Software Synthesis for Signal Processing Systems,” in *Proc. of the Int. Conf. on Application Specific Array Processors,* IEEE Computer Society Press, August 1992.

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

[37] W. T. Trotter, *Combinatorics and Partially Ordered Sets*, Johns Hopkins University Press, Baltimore, Maryland, 1992.

[38] A. M. Turing, “Computability and λ-Definability,” *J. Symbolic Logic,* Vol. 2, pp. 153-163, 1937.

[39] D. Verkest, K. Van Rompaey, I. Bolsens, and H. De Man, “POPE — A Design Environment for Heterogeneous Hardware/Software Systems,” to appear, *Design Automation for Embedded Systems*, 1996.

[40] M. von der Beeck, “A Comparison of Statecharts Variants,” in Proc. of Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863, pp. 128-148, Sprinter-Verlag, Berlin, 1994.

[41] P. Zepter and T. Grötker, “Abstract Multirate Dynamic Data-Flow Graph Specification for High Throughput Communication Link ASICs”, *IEEE VLSI DSP Workshop*, The Netherlands, 1993.

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

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

[14] C. Ellingson and R. J. Kulpinski, “Dissemination of System-Time,” *IEEE Trans. on Communications*, Vol. Com-23, No. 5, pp. 605-624, May, 1973.

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

[16] D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” *Sci. Comput. Program.,* vol 8, pp. 231-274, 1987.

[17] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, “STATEMATE: A Working Environment for the Development of Complex Reactive Systems,” *IEEE Trans. on Software Engineering,* Vol. 16, No. 4, April 1990.

[18] H. Hsieh, L. Lavagno, and A. Sangiovanni-Vincentelli, “Embedded System Codesign: Synthesis and Verification,” presented at NATO-ASI Workshop on Hardware/Software Codesign, Lake Como, June, 1995.

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

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

[21] A, Kalavade, Asawaree Kalavade, System Level Codesign of Mixed Hardware-Software Systems, Tech. Report UCB/ERL 95/88, Ph.D. Dissertation, Dept. of EECS, University of California, Berkeley, CA 94720, September, 1995.

[22] A. Kalavade and E. A. Lee, “A Hardware/Software Codesign Methodology for DSP Applications,” *IEEE Design and Test*, Vol. 10, No. 3, pp. 16-28, September 1993.

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

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

[25] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM*, Vol. 21, No. 7, July, 1978.

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

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

[28] E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow,” *IEEE Proceedings*, September, 1987.

In this paper we have discussed the interaction semantics of dataflow and discrete-event MoCs. Ongoing efforts in the Ptolemy project are studying the interactions between these MoCs and both synchronous languages and FSMs.

7. Acknowledgments

We would like to thank the Ptolemy team for building a magnificent laboratory for experimenting with the concepts discussed in this paper. The Ptolemy project is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), the Semiconductor Research Corporation (SRC) (project 96-DC-324-016), the National Science Foundation (MIP-9201605), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, LG Electronics, Mentor Graphics, Mitsubishi, Motorola, NEC, Philips, and Rockwell.

8. References

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

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

[3] 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.

[4] J. Bier, P. Lapsley, E. A. Lee, and F. Weller, “DSP Design Tools and Methodologies,” *Technical Report*, Berkeley Design Technology, 39355 California St., Suite 206, Fremont, CA 94538, 1995.

[5] 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.

[6] F. Boussinot, R. De Simone, “The ESTEREL Language,” *Proceedings of the IEEE*, Vol. 79, No. 9, September 1991.

[7] 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.

[8] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *Int. Journal of Computer Simulation*, special issue on “Simulation Software Development,” vol. 4, pp. 155-182, April, 1994. (http://ptolemy.eecs.berkeley.edu/papers/JEurSim.ps.Z).

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

[10] C. Cassandras, *Discrete Event Systems, Modeling and Performance Analysis*, Irwin, Homewood IL, 1993.

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

5. A Typical Mixed DE/Dataflow Simulation

In this section we outline a typical mixed DE/dataflow simulation, one studying packet video and audio. There are some terminals at the periphery of the network. The terminals perform signal compression and decompression, a good match for the dataflow MoC. Compressed signals are then packetized and sent into the network. The network loses some packets and delays the rest. This behavior is a good match for DE. At the destination, the packets must be converted back to compressed signals, with the lost packets and network delays handled properly. Then the compressed signals are decompressed and played back, again a good match for dataflow.

Note that although we suggest the use of DE for modeling the packet loss and delay in the network, some simple packet loss and delay mechanisms can be adequately modeled by SDF, resulting in a simpler, more efficient simulation. For more elaborate simulations, the combination of DE dataflow will be more convenient.

Consider a packet speech application (see figure 14). The top level may be the DE domain, which models the networking as well as providing the “clock” signals driving the conversion at the terminals. The terminals are internally modeled using SDF. In the network, some modules may internally contain SDF subsystems, which would usually model numeric computation that is activated in an event-driven fashion. For example, we may need to compute the checksum whenever a packet arrives (an event). The Ptolemy event horizon is adequate.

6. Conclusions

To support heterogeneity in implementation technologies and design styles in system-level design, we hierarchically nest distinct models of computation. Each MoC is selected to capture certain properties of a system specification or model, such as timing, functionality, concurrency, or communication. When one MoC is embedded within another, an interaction semantics is needed. There are usually several reasonable alternative interaction semantics; the most appropriate depends on the characteristics of the systems being described. As a principle in the Ptolemy system, we try to implement the “minimal” semantics, allowing users to implement alternative semantics by adding detail. This has worked well in practice, and results in a reasonably modular heterogeneous environment.

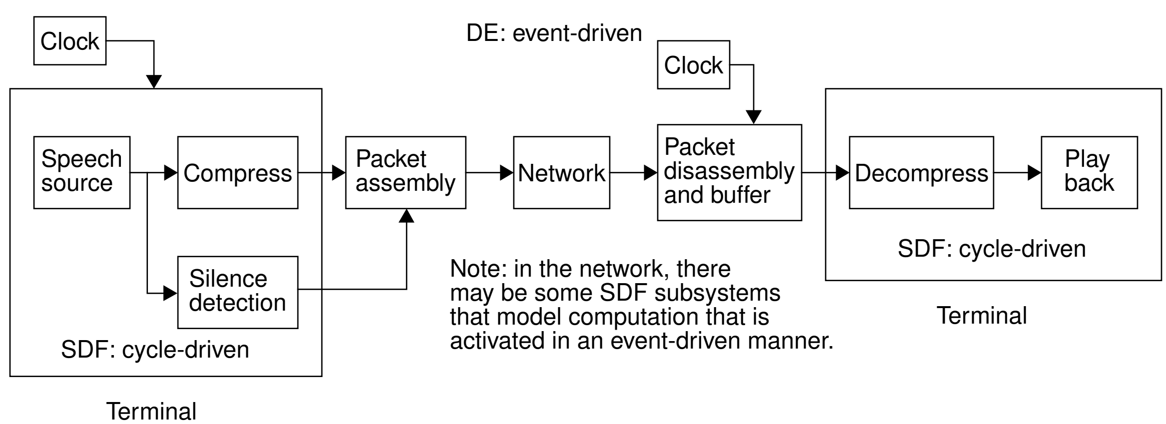


FIGURE 14. A packet speech simulation that combines discrete-event and dataflow models of computation.

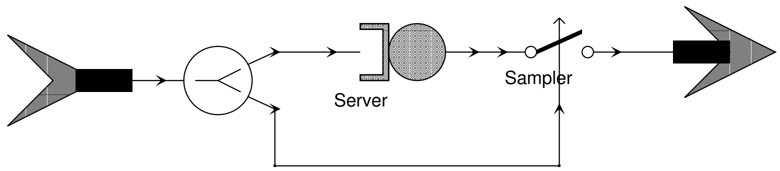


FIGURE 13. A DE subsystem designed for inclusion within an SDF system.

events it expects.

The behavior shown in figure 13 may not be the desired behavior. The *Sampler* block, given an event on its control input (the bottom input), copies the most recent event from its data input (the left input) to the output. If there has been no input data event, then a zero-valued event is produced. There are many alternative ways to ensure that an output event is produced. For this reason, the mechanism for ensuring that this output event is produced is not built into the Ptolemy event horizon. The programmer must understand the semantics of the interacting domains, and act accordingly.

Finally, consider a DE subsystem within a multirate SDF system. The SDF system may provide multiple tokens to the DE subsystem in each cycle of its schedule. By default, these tokens have time stamps that are uniformly spread over the user-specified schedule period. If this is not the desired behavior, the designer can always add one more level of hierarchy, where an outermost DE domain provides the time stamps to the inner SDF domain, which then passes these time stamps to the innermost DE domain. Uniformly spreading the tokens is a convenient policy, but it seems somewhat adhoc, so it may change in future versions of Ptolemy.

4.5 ALTERNATIVE INTERACTION SEMANTICS

The choices made in Ptolemy are by no means unique. The DE and dataflow domains could have been designed to use a different interaction semantics. For example, a reasonable way to embed a foreign subsystem within DE would be to provide a clock input to the wormhole that explicitly triggers a firing of the subsystem. Thus, if the inside system is dataflow, it would perform its quantum of computation when an event on the clock input is the oldest event in the event queue. Moreover, instead of checking to see whether there is enough data on the inputs to satisfy the dataflow actors inside, it could generate the data that is needed using the most recently seen values on each of the inputs. This alternative interaction semantics makes sense for certain kinds of systems. For example, the DE system might be modeling a digital circuit.

This alternative behavior can be implemented using the interaction semantics that we have implemented in Ptolemy. It would simply require a Sampler like that in figure 13 at each input to the wormhole. This set of Samplers would be driven by the clock.

In deciding which interaction semantics to implement, we were guided by a desire to minimize the amount of implicit activity. Thus, for example, generation of data tokens should be explicit in the design, not implicit in the interface between domains. This is a design choice, and alternative approaches are certainly arguable.

input tokens to fire. In this case, the SDF domain will simply return control to the DE domain, having refused to execute its quantum of computation. Only when enough input events have accumulated will the complete cycle be executed. Secondly, when output events are produced, more than one token may be produced per quantum of computation. In Ptolemy, all such output tokens are assigned the same time stamp by default. However, the user can change this behavior and specify that the time stamps should be uniformly spread through a given time interval. This latter behavior is often useful for modeling multirate signal processing systems, where the DE domain models the environment.

The notion of a complete cycle gets more difficult with more general dataflow models [27]. Unfortunately, for dataflow graphs supported by the BDF and DDF domains in Ptolemy, the existence of a complete cycle is undecidable [7]. The only solution we have identified is to define a quantum of computation of computation to be a complete cycle when it exists and can be found. Otherwise, it will be implementation dependent (and hence nondeterminate). Fortunately, most signal processing algorithms have dataflow graphs for which a complete cycle exists, can be found and is finite.

We need to consider one more special case. Suppose you want a dataflow subsystem to serve as a source of events in a discrete-event domain. Recall that source actors in the DE domain have to schedule themselves. One solution is to create a dataflow subsystem that takes a dummy input that triggers a quantum of computation.

4.4.2 Discrete event within dataflow

Consider the reverse scenario, where a DE subsystem is included within an dataflow system. The policy followed in Ptolemy is that a global notion of *current time* is maintained for use in the event horizon. For domains (such as the dataflow domains) that have no notion of time, this global time is maintained transparently. A dataflow system can be configured so that each complete cycle advances the global time by some fixed amount called the *schedule period*. This corresponds naturally to the representation of a sample rate in a signal processing system. Thus, again, we require that the dataflow graph have a finite complete cycle that is well-defined and determinate.

The inner DE subsystem must behave externally like a dataflow actor, but it also has some additional restrictions. SDF actors in general can produce or consume any fixed constant numbers of tokens on each port. In Ptolemy, the DE subsystem is constrained to behave like a *homogeneous* SDF actor, which consumes a single token on each input and produces a single token on each output. Only when the DE-in-dataflow wormhole has an event on every input port is it fired.

When the outer dataflow system chooses to fire the inner DE subsystem, the input events to the DE subsystem are assigned the global time as their time stamps. The inner DE scheduler is told to then process all input events up to and including events with that global current time as their time stamp. This is certainly not the only possibility, but it is unambiguous and seems to work well in practice.

Suppose that the outer domain is SDF. Recall that the SDF MoC requires that every firing consume and produce a fixed constant number of tokens. Therefore, a key requirement in this case is that when the DE subsystem is fired, it must produce an output event on each output port, since these will be expected by the SDF subsystem. A very simple example is shown in figure 13. The DE subsystem in the figure routes input events through a time delay (the *Server* block). The events at the output of the time delay, however, will be events in the future (in simulated time). The *Sampler* block, therefore, is introduced to produce an output event at the current simulation time. This output event is produced before the DE scheduler returns control to the output SDF scheduler, and the SDF system gets the

Because the outer MoC, discrete-event, has a total ordering on events, and the inner MoC is invoked in discrete quanta of computation, the events in the inner MoC are more ordered than they would be otherwise. Consider the example shown in figure 12(a). This homogeneous SDF subsystem consists of three actors A, B, and C. The firing schedule (A B C) forms a finite complete cycle, and hence a quantum of computation. If considered by itself as a dataflow graph, the firings have the partial ordering constraints shown in figure 12(b). However, if this subsystem is nested within a DE system, then it must be executed as a totally ordered sequence of complete cycles. This imposes additional ordering constraints on the firings, resulting in the partial order shown in figure 12(c).

In the more general form of SDF, actors can produce or consume more than one token when they fire (but they always produce and consume the same number on each firing). SDF graphs that do not have a finite complete cycle are considered defective [28], and are therefore ruled out.

If the SDF subsystem in figure 10 is a multirate system, the effects of the combined DE/SDF system are somewhat more subtle. First, a single event at the input of the subsystem may not be sufficient to cycle through one iteration of the SDF schedule. Suppose for example that actor B requires two

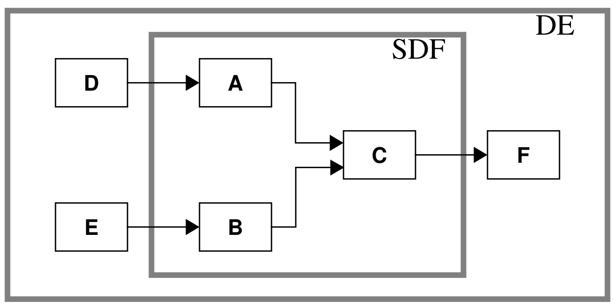
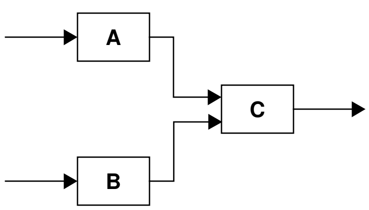
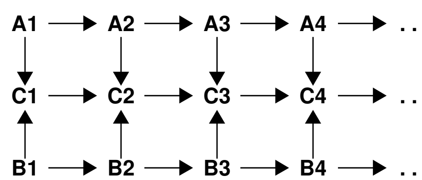


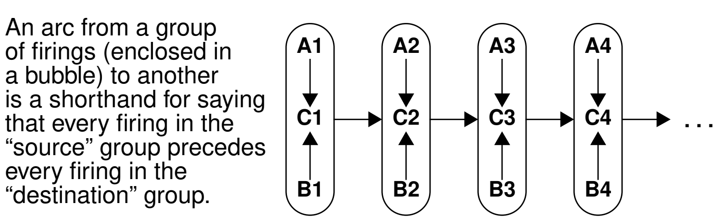
FIGURE 11. Illustration of a multi-input dataflow subsystem.



(a) A homogeneous SDF subsystem.



(b) Partial ordering of firings imposed by the dataflow MoC.



(c) Partial ordering of firings after imposing a total ordering on complete cycles.

FIGURE 12. Effect on partial orders of the quantum of computation.

flow MoC has no notion of time, it seems not only awkward but also infeasible to assign any execution delay to the inner dataflow subsystem. We can simulate any desired time delay of the inner dataflow subsystem by attaching a *Delay* block in the outer DE domain to the inputs or outputs of the wormhole that encapsulates the dataflow subsystem.

The question remains, what is a quantum of computation? How much work should the dataflow scheduler do before returning control to the DE scheduler? One possibility would be to fire a single dataflow actor, say B in figure 10, to respond to the event. But this will produce no output, and it is unclear when the dataflow scheduler should be invoked again to continue responding to the input event. Moreover, if a cluster of actors were to be replaced by a functionally equivalent monolithic actor, the behavior of the dataflow graph would change in a fundamental way. It would take several invocations to produce the same result.

A more reasonable alternative is to fire enough dataflow actors to return the dataflow graph to its original state. Thus, if all arcs start with zero tokens, and a token arrives at the input, the scheduler should fire the actors a minimal number of times to return all arcs to zero tokens. This set of firings is called a *complete cycle* [27]; it forms a *quantum of computation*, and is found by solving the *balance equations* for the graph.

Consider the simplest form of dataflow, known as *homogeneous synchronous dataflow*, where all actors produce and consume exactly one token on each input or output port. For a homogeneous SDF graph, a complete cycle always consists of exactly one firing of each actor. Suppose that the dataflow graph in figure 10 is homogeneous SDF. Then when the dataflow subsystem is invoked by the DE scheduler in response to an event with time stamp T, actors B and C will each fire once, in that order. Actor C will produce an output event, which when passed to the DE domain will be assigned the time stamp T. The DE scheduler continues processing by firing actor D to respond to this event.

Consider the slightly more elaborate example shown in figure 11. Suppose the inner system is homogeneous SDF, so a quantum of computation could consist of firing actors (A, B, C) in that order (this is a complete cycle). Suppose that D produces an event with time stamp T_1, and that this is the oldest event in the event queue. The wormhole is fired in response to this event, but there is still not enough data available for the firing sequence (A, B, C) to proceed as a quantum. Thus, the inner domain declines to act and returns control to the DE domain. Only when an event is produced by E will the inner system be able to proceed to execute its complete cycle. At that point, it will use the value produced by D, but will assume the time stamp produced by E.

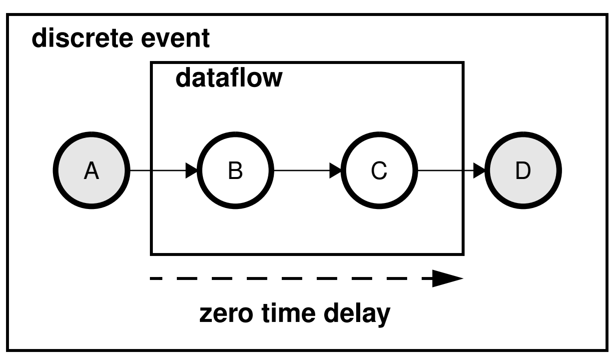


FIGURE 10. A dataflow subsystem as a module within a DE simulation.

- Properties in X that are not present in Y are ignored.
- Properties present in Y that are not present in X are created with reasonable default values.

In the latter case, there are usually several possible policies for choosing the default values. The event horizon provides a policy for certain properties. The domain must provide the rest.

The Ptolemy event horizon captures certain essential semantic properties relating to concurrency and communications. Events that cross the event horizon are totally ordered and carry time stamps represented as a floating point number. Every domain also provides a wormhole, which can contain a subsystem specified in a foreign domain. When the wormhole fires, it will request of the inner domain a quantum of computation. The inner domain may refuse to act (for example if there is not enough data at its inputs), or may perform some computation. In either case, it's behavior is expected to be determinate, implying that the quantum of computation must be well-defined.

To preserve the total ordering of events that cross the event horizon, each domain must also support a rudimentary notion of time. In particular, when asked to perform a quantum of computation, the inner domain will be given a *stop time*, the current time in the outer domain. It is expected to avoid progressing beyond that time internally. For untimed domains, such as dataflow domains, the stop time may be ignored.

4.4 INTERACTION SEMANTICS IN PTOLEMY

4.4.1 Dataflow inside discrete-event

Consider the case of a dataflow model inside a DE model, as shown in figure 10. In Ptolemy, the dataflow subsystem appears to the DE simulator as a zero-delay block. Suppose, for example, that an event with time stamp T is available at the input to the dataflow subsystem. Then when the DE scheduler reaches this simulated time, it fires the dataflow subsystem, turing control over to the dataflow scheduler. If the dataflow subsystem produces output data, that output data will have time stamp T.

The motivation for assuming zero execution time of the dataflow subsystem is that, since the data-

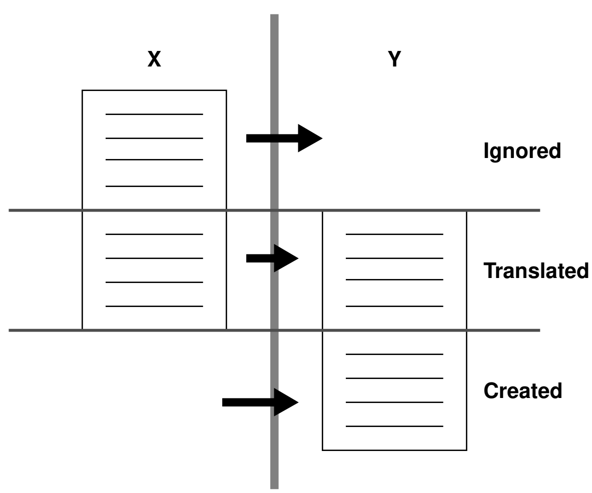


FIGURE 9. Conversion from the semantic properties of MoC X to the semantic properties of MoC Y at the interface

Boolean dataflow (BDF). This domain supports a more general model where a Boolean valued input or output from a process may determine how many tokens are consumed or produced on an input or output [7].

Dynamic dataflow (DDF). This domain is a further generalization of BDF where mutable graphs are supported and the only constraint on firings is that they be *sequential functions* [29]. A complete discussion of sequential functions is beyond the scope of this paper.

The dataflow domains in Ptolemy are commonly used for specifying signal processing or other computation-intensive applications. It is easy to envision situations where dataflow and DE would be naturally combined. For example, a signal processing subsystem might be embedded within a communication network simulation (dataflow within DE). Similarly, a model of a hardware component might be embedded within a specification of a signal processing system during a design refinement (DE within dataflow). Or a functional model of a DSP ASIC might be embedded within a behavioral model of a software architecture.

A MoC captures some properties of a system, and omits those that are irrelevant to the problem domain. Different MoCs capture different sets of properties. A discrete-event MoC models the timing of interactions between concurrent modules, while a dataflow MoC models the exchange of data and the associated transformation of that data.

4.3 INTERFACING DIFFERENT MODELS OF COMPUTATION

Ptolemy domains are designed to interact with one another in a polymorphic way. A module in one domain can internally implement another. These domains interface to a common infrastructure, a set of conventions in the Ptolemy kernel, rather than explicitly interfacing to one another. This permits us to have N domains with N interfaces, rather than the N^2 interfaces that would be required if each domain had to explicitly interface to each other domain. This one, “universal” interface is called the *event horizon* in Ptolemy. This interface has to be sufficiently rich to be capable of transporting key semantic properties from one MoC to another.

We can draw an analogy to the conversion between the numeric types (short, int, long, float, double, and Complex, for example). To preserve numeric accuracy, a “universal” numeric type must be the type with the highest *precision*, which for the given set is Complex. Choosing any other type as the universal type would result in loss of information when a higher-precision type is converted to the universal type. The same principle must be applied in the design of the event horizon.

Consider for example the notion of time. Since time defines a total order, a timed model is more “precise” than an untimed model, which defines only a partial order. An untimed model can be converted to a timed model with no loss of correctness, but not vice versa. The event horizon, therefore, must support a notion of time.

Differences between MoCs can lead to ambiguities when two MoCs are brought together. These must be resolved by an *interaction semantics*. For the purpose of discussion, suppose that a MoC can be represented as a list of its semantic properties. Consider an interface between MoCs X and Y, as shown in figure 9, with information or control flowing from X to Y. At the interface, the following conversion from one set of semantic properties to the other must occur:

- The common properties are translated.

will. Object-oriented principles are used to hide information about the semantics of one domain from another. Thus, multiple MoCs can be combined in one system specification and/or model. A key constraint is that processes in the MoCs have discrete firings[1]. Ptolemy permits the construction of a process in one domain for which each firing invokes some *quantum of computation* in another domain. A second key constraint, therefore, is that each MoC have a well-defined quantum of computation. By this we mean that if we tell the domain to "execute one quantum of computation," the domain will perform a determinate and finite operation.

Note that in Ptolemy, MoCs are mixed *hierarchically*, as shown in figure 8. This means that two MoCs do not interact as peers. Instead, a foreign MoC may appear inside a process. In Ptolemy, such a process is called a *wormhole* [8]. It encapsulates a subsystem specified using one MoC within a system specified using another. The wormhole must obey the semantics of the outer MoC at its boundaries, and the semantics of the inner MoC internally. Information hiding insulates the outer MoC from the inner one.

Contrast this hierarchical heterogeneity to alternative approaches. Two MoCs may be used to describe orthogonal and complementary aspects of the same system, as for example in the control/dataflow graphs commonly used in VLSI CAD. Another example is the provision of alternative "views" of the same system that reflect different properties, as for example structural, functional, and behavioral descriptions of circuits [17]. A third possibility is to allow an intermixing of protocols for communication between computational modules [18][39]. These styles of heterogeneity are not hierarchical, and may be used in combination with hierarchical heterogeneity.

4.2 CHOOSING MODELS OF COMPUTATION

The discrete event (DE) domain in Ptolemy is used for time-oriented simulations of systems such as queueing networks, communication networks, and high-level hardware architectures (processors, disks, caches, etc.).

There are several dataflow domains in Ptolemy, ordered here by increasing generality:

Synchronous dataflow (SDF). This domain supports fixed dataflow process networks where each firing in a process consumes and produces a fixed, constant number of tokens on each input and output [28].

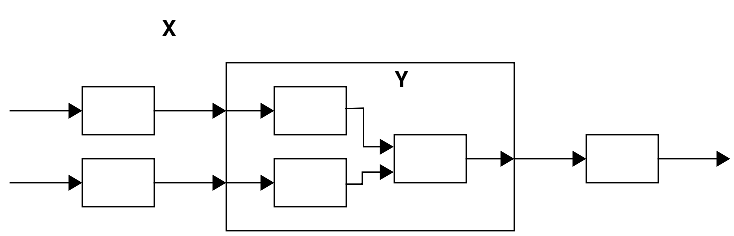


FIGURE 8. Mixing MoCs using hierarchy: shown here is a subsystem of domain Y embedded in domain X as a hierarchical node.

1. Some Ptolemy domains, such as the process networks (PN) domain, are not based on discrete firings, but instead use threading or multitasking. But these domains cannot, at our current stage of understanding, be intermixed as freely with other domains.

$i < j$. Let the output be $s_2 = \{e_{2,i}\}$. Then the process imposes the additional ordering constraint that $e_{1,i} < e_{2,i}$. for all i.

Lamport [25] considers a similar model for distributed systems where events occur inside processes, instead of in signals modeling communication between processes. The set of events inside a process is totally ordered, thus giving the process a sequential nature. Partial ordering constraints exist between events in different processes, thus modeling communication. This perspective is only slightly different from ours, where partial ordering constraints between events are imposed by the processes, rather than the communication between them.

3.2 TAGGED FIRING

In many MoCs, a process will be specified as a sequence of atomic actions that we will call firings. These firings can be tagged in much the way that events are. Let A denote the set of all possible actions. In this case, a process can be described as a set of firings, $P = \{p_i; i \geq 0\}$, where each firing p_i consists of an action and a tag, $p_i = (a_i, t_i)$. A *sequential process* is one where the firing tags are totally ordered. These tags will typically also have an ordering relationship with the input and output events.

A *dataflow process* is a Kahn process that entirely consists of discrete firings. A *dataflow process network*, is a network of such processes [29].

For example, consider a dataflow process $P = \{p_i; i \geq 0\}$ with one input signal and one output signal that *consumes* one input event and *produces* one output event on each firing p_i. Denote the input signal $s_1 = \{e_{1,i}\}$, where $e_{1,i} < e_{1,j}$ if the index $i < j$. Let the output be $s_2 = \{e_{2,i}\}$, which will be similarly ordered. Then the inputs and outputs are related to the firings as $e_{1,i} < p_i < e_{2,i}$.

The firings in a discrete-event system can now also be described in terms of tags. For any process P with input signal s containing event $e = (v, t)$, there exists a firing $p \in P$ with tag t. Moreover, output events produced by that firing cannot have tag less than t. If the events are totally ordered, so are the firings. A sequential execution of a discrete-event system simply follows this ordering when carrying out the firings. I.e., it uses the ordering of firing tags to determine the ordering of firing times (in real time).

4. Mixing discrete events and dataflow

In this section, we will use the Ptolemy system, developed at the University of California at Berkeley, as a case study to illustrate some of the issues in mixing discrete events and dataflow. Ptolemy is a research project and software environment focused on design methodology for signal processing and communications systems. Its scope ranges from designing and simulating algorithms to synthesizing hardware and embedded software, parallelizing algorithms, and rapidly prototyping real-time systems. Research ideas developed in the project are implemented and tested in the Ptolemy software environment. The Ptolemy software environment is a system-level design framework that allows mixing models of computation and implementation languages

4.1 DOMAINS AND WORMHOLES

In Ptolemy, a *domain* defines the semantics of a coordination language. Each domain has a *scheduler* that implements the semantics. But domains are modular objects that can be mixed and matched at

overly restrictive. A specification should not be constrained by one particular physical implementation, and therefore need not be based on the semantics of the physical world. Thus, for specification, often the tags *should not* mark time.

In a *model* of a physical system, by contrast, tagging the events with the time at which they occur may seem natural. They must occur at a particular time, and if we accept that time is uniform, then our model should reflect the ensuing ordering of events. However, when modeling a large concurrent system, the model should probably reflect the inherent difficulty in maintaining a consistent view of time in a distributed system [14][25][30]. If an implementation cannot maintain a consistent view of time, then it may be inappropriate for its model to do so (it depends on what questions the model is expected to answer).

Fortunately, there are a rich set of untimed models of computation. In these models, the tags are more abstract objects, often bearing only a partial ordering relationship among themselves. We now give one way to classify these models of computation.

3.1.1 Types of Systems

A *partially ordered tagged system (POTS)* is a tagged system where T is a countable, partially ordered set. *Partially ordered* means that there exists an irreflexive, antisymmetric, transitive relation between members of the set [37]. We denote this relation using the symbol “<”. Of course, we can define a reflexive version of this relation, denoted “≤”, where $t_1 \le t_2$ if $t_1 = t_2$ or $t_1 < t_2$.

The ordering of the tags provides an ordering of events as well. Given two events $e_1 = (v_1, t_1)$ and $e_2 = (v_2, t_2)$, $e_1 < e_2$ if $t_1 < t_2$.

A *timed system* is a tagged system where T is totally ordered. That is, for any distinct t_1 and t_2 in T, either $t_1 < t_2$ or $t_2 < t_1$. A *discrete-event system* is a timed system where T is a countable. The use of the term “timed” here stems from the observation that in the standard model of the physical world, time is viewed as globally ordering events. Any two events are either simultaneous (have the same tag), or one unambiguously precedes the other. Some timed MoCs include a distance metric between tags, where for example $t_2 - t_1$ has some meaning. These MoCs are said to have *metric time*.

In some communities, a discrete-event MoC also requires that V be countable [10]. For our purposes, the distinction is technically moot, since all representations of values in a computer simulation are drawn from a countable set.

Two events are *synchronous* if they have the same tag. Two signals are synchronous if all events in one signal are synchronous with an event in the other signal and vice versa. A system is synchronous if every signal in the system is synchronous with every other signal in the system. A *discrete-time system* is a synchronous discrete-event system.

By this definition, the “synchronous languages” [2] (such as Lustre, Esterel, and Signal) are not synchronous. They are discrete-event systems. Neither is the so-called Synchronous Dataflow (SDF) model of computation [28].

Let T_i denote the tags in signal s_i. In a *Kahn process network,* T_i is totally ordered for all signals s_i, but T may be only partially ordered. In fact, in one possible POTS model for Kahn process networks, $T_i \cap T_j = \varnothing$ for all $i \ne j$. Kahn process networks also impose partial ordering constraints on the tags of their input and output signals. For example, consider a simple process that produces one output event for each input event. Denote the input signal $s_1 = \{e_{1,i}\}$, where $e_{1,i} < e_{1,j}$ if the index

ficiently constrained, it can provide a framework within which the semantics of the interaction between models of computation can be understood. Generality can be achieved through heterogeneity, where more than one model of computation is used.

The objective of communication between tools is harder to achieve. Fundamentally, a translation from a language used by one tool to that used by another is primarily a translation of syntax. Translation of semantics is much harder. Thus, to get interoperability between tools by translation, we force each tool to implement each semantic model of interest. This, needless to say, severely complicates tool development, and serves as a barrier to innovation.

To address these issues, we develop a human language that will enable description and differentiation of models of computation. To be sufficiently precise, this language is a mathematical one. We will define precisely a set of terms like "process", "discrete-event systems", and "synchronous" that are unambiguous, but will often conflict with common usage in some communities. We have made every attempt to maintain the spirit of that usage with which we are familiar, but have discovered that terms are used in contradictory ways in different communities (sometimes even within a community). Maintaining consistency with all prior usage is impossible without going to the unacceptable extreme of abandoning the use of these terms altogether. Our definitions in this paper conflict even with our own prior usage in certain cases. We apologize to anyone who is offended by our abuse of some term for which they have a different meaning that they are comfortable with. But there is no way to accomplish our objectives without this abuse.

3.1 THE TAGGED SIGNAL MODEL

Given a set of *values* V and a set of *tags* T, we define an event e to be a member of $V \times T$. I.e., an event has a tag and a value. We define a *signal* s to be a partial function from T to V. By "partial function" we mean that the signal may be defined only for a subset of T. A signal s can be viewed therefore as a collection s of events, i.e. a subset of $V \times T$, where if $e_1 = (v_1, t) \in s$ and $e_2 = (v_2, t) \in s$, then $v_1 = v_2$. We call the set of all signals S. It is often useful to form a collection $\mathbf{s}$ of n signals. The set of all such signals will be denoted S^n.

A *process* P with n inputs and m outputs is a subset of $S^n \times S^m$. In other words, a process defines a *relation* between input signals and output signals. Signals provide communication between processes. Note that with this definition, non-determinacy is supported. A particular set of input signals may have several possible sets of output signals that satisfy the relation. A *functional process F*, or simply a *function*, is a single valued mapping from some subset of S^n to S^m. That is, if $(\mathbf{s}_1, \mathbf{s}_2) \in F$ and $(\mathbf{s}_1, \mathbf{s}_3) \in F$, then $\mathbf{s}_2 = \mathbf{s}_3$.

A *tagged system* is a set of signals and processes defining relations between these signals. A tagged system is said to be *fixed* if each process and each signal exists throughout the existence of the system. Otherwise, it is said to be *mutable*. Thus, in a mutable tagged system, processes or signals may come and go. Our term "tagged system" here does not refer to an implementation. It refers to a model or a specification. This usage of "system" is common in system theory, but uncommon in other disciplines.

Frequently, a natural interpretation for the tags is that they mark time in a physical system. Neglecting relativistic effects, time is the same everywhere, so tagging events with the time at which they occur puts them in a certain order (if two events are genuinely simultaneous, then their order is arbitrary). For *specifying* systems, however, the global ordering of events in a timed system may be

ical sort views this as a precedence constraint. Ignoring the feedback arc in figure 5, this would resolve all ambiguities. The topological sort would indicate that B should always be invoked before C when they have events at their inputs with identical time stamps. This sort of precedence analysis is identical to that done in synchronous languages (Esterel, Lustre, and Signal) to ensure that simultaneous events are processed in a deterministic way.

Of course, the feedback loop in figure 5 creates a problem. The same problem occurs in synchronous languages, where such loops are called causality loops. No precedence analysis can resolve the ambiguity. In synchronous languages, the compiler may simply fail to compile such a program. In the discrete-event domain in Ptolemy, we permit the user to annotate the arcs the graph to break the precedences. Thus, the programmer could annotate the leftward pointing arc in figure 5, again resolving the ambiguities. If the user fails to provide such annotation, a warning is issued, and the precise behavior is arbitrary (nondeterminate).

A slight subtlety arises in discrete-event simulators with sources of events. Consider the example shown in figure 7(a). An initial event on the self loop with time stamp T causes A to fire at simulated time T. At that time, it can produce an event to B and another event on the self loop with a larger time stamp. By continuing to produce events on the self loop, it continually acts as a source of events for B. In the Ptolemy DE domain, we support the syntactic sugar shown in figure 7(b), where the self loop is hidden from view. Conceptually, any source of events is allowed to put its own future firings on the event queue.

3. Time and Partial Orders

A major impediment to further progress in heterogeneous modeling and specification is the confusion that arises from different usage of common terms. Terms like “synchronous”, “discrete event”, and “dataflow” are used in different communities to mean significantly different things. No doubt, many readers have interpreted the descriptions above in ways quite different from what we intended.

Recognition of this problem has led to efforts to “unify” the work of different communities by defining a “grand unifying language” that embraces the semantic models of a set of competitors. A common language, it is suggested, provides a means by which people and tools can communicate more effectively. While we agree that such languages provide these benefits, we argue that they do so at great expense. In order to be sufficiently rich to encompass the varied semantic models of the competing approaches, they become unwieldy, too complex for formal analysis and high quality synthesis.

We argue that small, simple, specialized models of computation are valuable. Efforts to subsume them into large, unwieldy, “general” models are misguided. The objectives of human communication can be achieved more simply by developing a human language that is sufficiently precise and formal to unambiguously differentiate approaches. Moreover, if the objectives of the resulting formalism are suf-

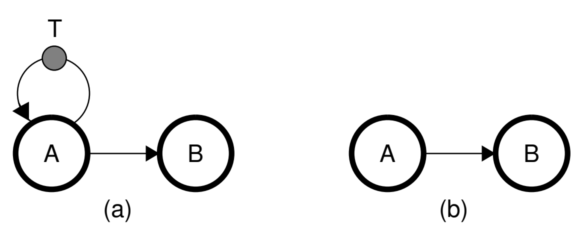


FIGURE 7. Sources of events in DE with and without explicit feedback loops.

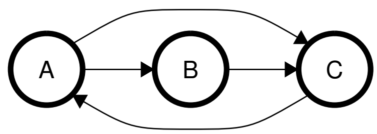


FIGURE 5. A discrete-event example.

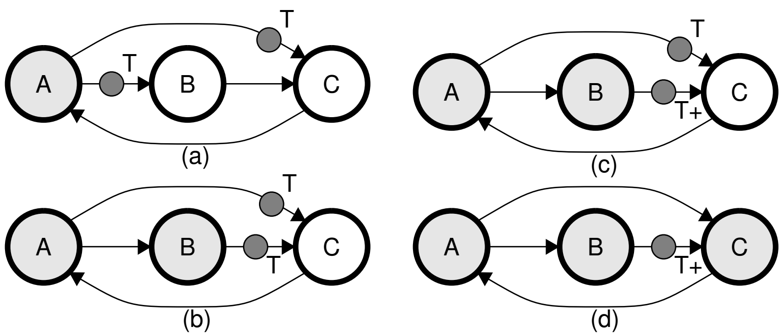


FIGURE 6. Simultaneous events in discrete-event systems.

Consider the graph shown in figure 5. Suppose it specifies a program in a discrete-event coordination language. Suppose further that B is a zero-delay component. This means that its output has the same time stamp as the most recent input. Thus, if A produces one event on each of its two outputs with the same time stamp *T*, then there is an ambiguity about whether B or C should be invoked next. This situation is illustrated in figure 6(a). B and C have events at their inputs with identical time stamps, so either could be invoked next. But the behavior of C could be different in the two circumstances.

Suppose B is invoked first, resulting in the configuration shown in figure 6(b). Now, depending on the simulator, C might be invoked once, observing both input events in one invocation. Or it might be invoked twice, processing the events one at a time. In the latter case, there is no clear way to determine which event should be processed first.

Some discrete-event simulators leave this situation ambiguous. Such simulators are *nondeterminate*. In most applications, this is not desirable. A partial solution provided in some simulators is the *infinitesimal delay*. If B has an infinitesimal delay, then its output events will have time stamps that are ordered after those of the inputs even if they represent the same simulated time. Then, firing A followed by B will result in the situation shown in figure 6(c), where the effect of the infinitesimal delay is indicated by the "*T+*". The next firing of C will observe only the first event, the one with time stamp *T*. This is the next one in the event queue. After this firing of C, the event with time stamp *T+* remains to be processed, as shown in figure 6(d).

Infinitesimal delays are not an entirely satisfactory solution. Suppose the designer wishes for C to see both events at once, as in figure 6(b). There is no way to ensure that B will be invoked before C. For this reason, the discrete event domain in Ptolemy uses a different solution [8]. Graphs specifying discrete event programs are topologically sorted, and a priority is assigned to each arc. The topological sort is based on an annotation of the nodes in the graph indicating whether the node can have zero delay from any particular input to any particular output. When such zero delay is possible, the topolog-

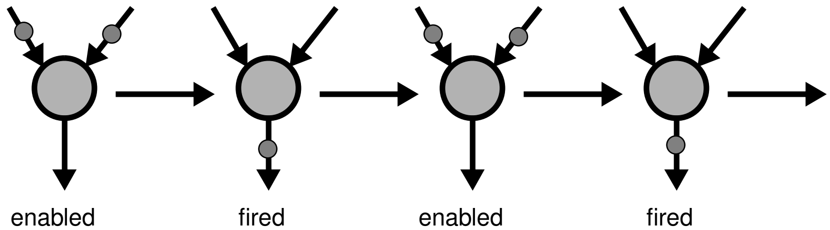


FIGURE 3. A dataflow process consists of repeated firings of an actor.

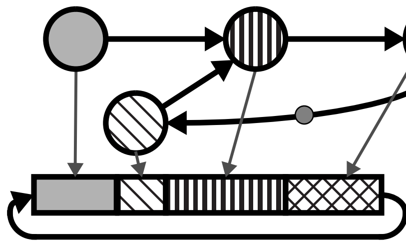


FIGURE 4. Static scheduling of a dataflow process network.

schedule should return the graph to its original *state* (defined as the number of tokens on each arc). This is not always possible, but when it is, considerable simplification results.

Many possibilities have been explored for precise semantics of dataflow coordination languages, including for example the computation graphs of Karp and Miller [24], the synchronous dataflow graphs of Lee and Messerschmitt [28], the cyclo-static dataflow model of Lauwereins, *et al*. [26][5], the Processing Graph Method (PGM) of Kaplan, *et al.* [23], Granular Lucid [19], and others [1][9][12][36]. Many of these limit expressiveness in exchange for considerable advantages such as compile-time predictability.

Synchronous dataflow (SDF) and cyclo-static dataflow both have the particularly useful property that a finite static schedule can always be found that will return the graph to its original state. This allows for extremely efficient implementations. For more general dataflow models, it is undecidable whether such a schedule exists [7].

A key property of dataflow processes is that the computation consists of atomic firings. Within a firing, anything can happen. In many existing environments, what happens can only be specified in a host language with imperative semantics, such as C and C++. In the Ptolemy system [8], it can consist of a quantum of computation specified with any of several models of computation. We will return to this notion of a "quantum of computation."

2.2 DISCRETE EVENT

As described above, the discrete-event model of computation has events with time stamps. The role of the simulator is to keep a list of events sorted by time stamp and to process the events in chronological order. There are, however, some subtleties that are dealt with differently in different systems. The main difficulties concern how *simultaneous events* (those with the same time stamp) are dealt with, and how *zero-delay* feedback loops are managed.

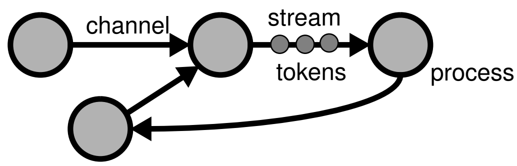


FIGURE 2. In a process network, where processes communicate through unidirectional FIFO channels, writes are non-blocking, and reads are blocking.

2.1 DATAFLOW PROCESS NETWORKS

In dataflow, a program is specified by a directed graph where the nodes represent computations (*actors*) and the arcs represent *streams* of data *tokens*. The graphs are often represented visually and are typically hierarchical, in that an actor 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. In the latter case, we are already mixing two of the models of computation from figure 1, where dataflow serves as the coordination language for subprograms written in an imperative host language.

Some examples of graphical dataflow programming environments intended for signal processing (including image processing) are Khoros, from the University of New Mexico [34] (now distributed by Khoral Research, Inc.), Ptolemy, from the University of California at Berkeley [8], 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).

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 deviate significantly from those outlined by Dennis in 1975 [13]. Most, however, can be described formally as special cases of *dataflow process networks* [29], which are in turn are a special case of *Kahn process networks* [20].

In Kahn process networks, a number of concurrent processes communicate through unidirectional FIFO channels, where writes to the channel are non-blocking, and reads are blocking (see figure 2). This means that writes to the channel always succeed immediately, while reads block until there is sufficient data in the channel to satisfy them. In particular, a process cannot test an input channel for the availability of data and then branch conditionally. Testing for available data constitutes a read, and will block the entire process until data is available. This restriction helps to assure that the program is *determinate*, meaning that its outputs are entirely determined by its inputs and the behavior specified by the programmer.

In dataflow process networks, each process consists of repeated *firings* of a dataflow *actor* (see figure 3). An actor defines a (often functional) quantum of computation. By dividing processes into actor firings, the multitasking overhead of context switching incurred in direct implementations of Kahn process networks is avoided. In fact, in many of the signal processing environments, a major objective is to statically (at compile time) schedule the actor firings (see figure 4). The firings are organized into a list (for one processor) or set of lists (for multiple processors). In figure 4, a dataflow graph is shown mapped into a single processor schedule. Thus, the lower part of the figure represents a list of firings that can be repeated indefinitely. A basic requirement of such a schedule is that one cycle through the

Visual dataflow programming languages, for example, are commonly used in the signal processing community for specification. Hierarchical finite-state machine languages are used for specifying control-oriented systems. Symbolic processing languages are used to specify functionality in scientific computing. Imperative languages are used for everything (whether or not they are appropriate). Discrete-event MoCs are used to specify concurrent communicating systems.

Discrete-event MoCs, such as those used in VHDL, Verilog, and other simulation environments, are also used for modeling implementations. They describe physical components with states that evolve over time and with interactions that occur at discrete points in time. Imperative MoCs provide a natural way to model software implementations. They are also used for modeling hardware components at higher levels of abstraction. For example, an instruction set architecture model for a hardware processor might be implemented in the imperative language C rather than using a discrete-event language to model its detailed implementation.

Often, it makes sense to combine modeling and specification. *Design elaboration*, for example, is the process of replacing portions of an executable specification with ever more detailed models of the implementation. Often, for efficient simulation, it makes sense to maintain multiple levels of abstraction in a system simulation. This requires a simulation environment where diverse models of computation can interact.

1.6 FROM SPECIFICATION TO MODELING

In figure 1, the synthesis paths are somewhat constrained by the choice of MoC used for specification. It is well known, for example, that a discrete-event MoC is difficult to implement efficiently on a sequential machine. This is the main reason that VHDL simulations surprise the designer by taking so long. A model that heavily uses entities communicating through signals will burden the discrete-event scheduler and bog down the simulation. Thus, a specification built on discrete-event semantics is a poor match for implementation in software.

By contrast, VHDL that is written as strictly sequential code, using imperative semantics, runs relatively quickly, but may not translate well into hardware. Imperative C, of course, runs very quickly, and is well suited to specifying components that will be implemented in software. However, it is poorly suited to specifying hardware.

Dataflow and finite-state machine MoCs have been shown to be reasonably retargettable. Hierarchical FMS such as Statecharts [16][17][40], for example, can be used effectively to design hardware or software. Similarly, a number of commercial and research tools use dataflow to specify signal processing systems that can be implemented either way [32][35][41]. It has also been shown that a single dataflow specification can be partitioned for combined hardware and software implementation [21][22].

2. Multi-paradigm design

In this paper, we will focus on the interaction between dataflow models, with their partially ordered events, and discrete-event models, with their notion of time defining a (mostly) total order of events. A variety of interaction semantics are explored, and an implementation based on the Ptolemy system from U. C. Berkeley is described. We begin with some background on each. This will be followed by a formal framework that unambiguously defines the essential features of each MoC and their interaction.

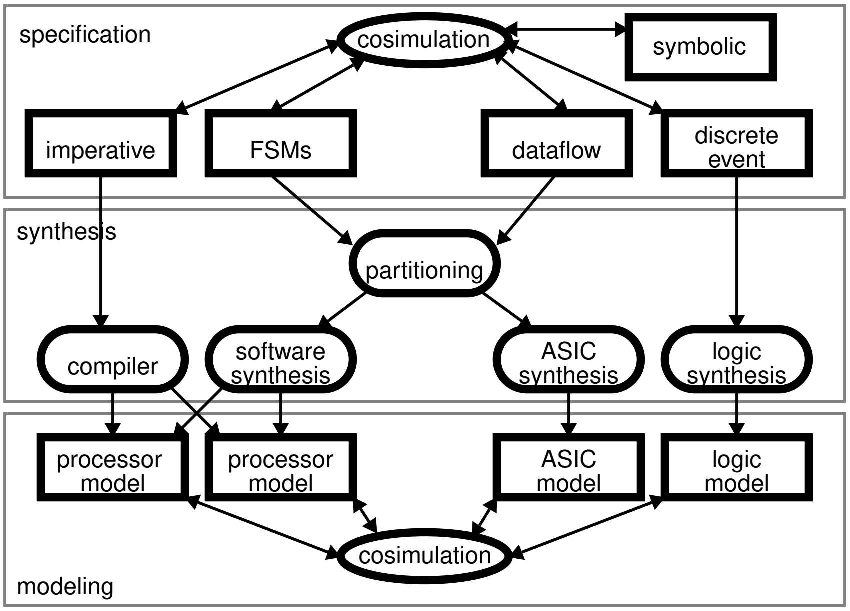


FIGURE 1. Models of computation and cosimulation.

2. Integration may not be complete. For example, once a design is migrated from one tool to another, without tight integration, it may be difficult to back annotate the original design.

We believe that with system-level design problems, the advantages outweigh the disadvantages, and that the disadvantages will at least partially disappear later as the technology solidifies.

Our experience suggests that several models of computation are required for the design of complete systems. In particular, in order to successfully apply formal methods, and in order to obtain good results from high-level synthesis, the smallest (most restrictive) models of computation are best. Thus, to achieve generality, one has to embrace heterogeneity.

1.5 MODELING AND SPECIFICATION

There is a subtle relationship between the specification of a system and the modeling of a system. An executable specification, for example, is also a model of an implementation. The difference is in emphasis. A *specification* describes the functionality of a system, and may also describe one or more implementations. A *model* of a system describes an implementation, and may also describe the functionality. In a specification, it is important to avoid overspecifying the design, to leave implementation options open. In a model, often the key criteria are precision, simplicity, and efficient simulation. A model should be the most abstract model that represents the details being tested.

Figure 1 shows the role that specification and modeling might take in system design. Specification is closer to the problem level, at a higher level of abstraction, and uses one or more models of computation. A specification undergoes a synthesis process (which may be partly manual) that generates a model of an implementation. That model itself may harbor multiple models of computation, mainly to model components at varying levels of abstraction, or to separately model hardware and software components.

tical expressiveness of the language become central.

This tension between theorists and practitioners is healthy, and the best solutions emerge from compromise. But a major risk in this compromise is "creeping featurism." In an effort to win the broadest support, features and options are added to a language until all semblance of simplicity has been lost. This is frequently how languages come to contain more than one MoC. The down side of such large languages with multiple MoCs is that formal analysis may become very difficult. This compromises our ability to generate efficient implementations or simulations. It also makes it more difficult to be sure that a design is correct; it precludes such formal verification techniques as reachability analysis, safety analysis, and liveness analysis.

Usually, features are added to a language for good reason. Their advocates can cite numerous applications, and win their acceptance through compelling arguments about the utility of the features. The key is that both excessive simplicity and excessive complexity can interfere with utility.

1.4 HETEROGENEITY

A reasonable way to balance the pressures of breadth and simplicity is to support heterogeneity. Failure to do so has doomed some otherwise laudable efforts. For example, Common Lisp did not at first define a "foreign function interface," presumably because everything could be done in Lisp. Practitioners, who built Common Lisp systems, realized that this was not a practical approach, and built their own, mutually incompatible foreign function interfaces. As a result, Lisp systems were only interchangeable if applications used no foreign functions. And few significant applications qualified. Standardization failed.

Small, specialized languages and tools are useful. For example, viewed as languages, spreadsheets are extremely useful. But they are certainly not general, in that there are many applications for which they are inappropriate. Such tools are therefore most useful if they can be combined with other specialized tools. Each tool is developed independently, making the development effort manageably small. But by embedding the tool in an environment with appropriate interfaces between tools (a file system, cutting and pasting, and a multitasking operating system, to name some examples), the utility of the tool is greatly magnified.

Looser integration of diverse capabilities has numerous advantages:

1. Existing, familiar tools do not need to be discarded in favor of a new, all-encompassing tool. This is particularly valuable when expertise with complex tools has built up over time.
2. Capabilities can come in "bite sized" modules, making them easier to learn, and rendering them more acceptable to a cautious clientele.
3. Tools from different vendors can be mixed, drawing from the strengths of each.
4. Competition among vendors is enhanced because fewer customers are "locked in." This results in better tools at lower prices.
5. Innovative tools with specialized capabilities have a market. They do not need to be all-encompassing to be salable. So innovation is encouraged.

There are also significant disadvantages:

1. User interfaces are likely to be different with different tools, making them more difficult to learn.

fication are called *cycle-based* or *cycle-driven* simulators. Processing all events at a given clock tick constitutes a cycle. Within a cycle, the order in which events are processed may be determined by data precedences, which therefore define *microsteps*. These precedences are not allowed to be cyclic, and typically impose a partial order. Cycle-based models are excellent for clocked synchronous circuits. They have also been applied successfully at the system level in certain signal processing applications.

A cycle-based model is inefficient for modeling systems where events do not occur at the same rate in all signals. While conceptually such systems can be modeled (using for example special tokens to indicate the absence of an event), the cost of processing such tokens is considerable. Fortunately, the cycle-based model is easily generalized to multirate systems. In this case, every n-th event in one signal aligns with the events in another.

A multirate cycle-based model is somewhat limited. It is an excellent model for synchronous signal processing systems where sample rates are related by known rational multiples, but in situations where the alignment of events in different signals is irregular, it can be inefficient.

A more general model is embodied in the so-called *synchronous languages* [2]. Examples of such languages include Esterel [6], Signal [3], and Lustre [15]. In synchronous languages, every signal is conceptually (or explicitly) accompanied by a *clock signal*. The clock signal has meaning relative to other clock signals. It defines the global ordering of events. Thus, when comparing two signals, the associated clock signals indicate which events are simultaneous and which precede or follow others. A clock calculus allows a compiler to reason about these ordering relationships and to detect inconsistencies in the definition.

Various looser models of computation specify only a partial ordering between events. This means that while events within any given signal are ordered, events in different signals may or may not have an ordering relationship. This type of specification has the advantage that it avoids *overspecifying* a design. If an ordering relationship is not important in a design, why specify it? Specifying it may severely constrain the implementation options. Thus, for example, while discrete-event simulators are difficult to parallelize, dataflow models, which are usually partially ordered [29], are comparatively easy to parallelize.

1.3 EXPRESSIVE POWER

Theoreticians strive for simple but expressive models of computation. “Simple” in this case means that the MoC can be defined by a language with only a few primitives. For example, Turing machines, which define an imperative MoC, are defined by the Turing-Post language, which has only seven instructions [33][38]. Church’s lambda calculus is based on only a few formal rules for transforming strings [11]. “Expressive” in this case means that the MoC can specify many different systems. Both Turing machines and the lambda calculus can describe a set of functions that are called the “effectively computable functions.” This set is so large that many people regard any computation that is not in the set to be not computable.

Practitioners view such efforts much the way they view Turing machines: they make interesting abstractions, but they do not tell us much about how to build systems. To a practitioner, the utility of a MoC stems from a more pragmatic view of expressiveness: how easy is it to construct a given system description? How expensive is the compiled implementation? How sure can we be that the design is correct? It is largely irrelevant that it is theoretically possible to construct such a description. Moreover, since the practitioner works more directly with a language than with a MoC, the syntax and prac-

variants [40], combine FSMs with a concurrent MoC, typically that of synchronous languages. Languages that are fundamentally based on one MoC may also be used to implement another. For example, C, which is fundamentally imperative, may be used to implement a dataflow MoC [8].

A MoC is often most easily defined in terms of a language. The language may be very incomplete and/or very abstract. For example, it may specify only the interaction between computational modules, and not the computation performed by the modules. Instead, it provides an interface to a *host language* that specifies the computation, and is called a *coordination language*. Or the language may specify only the causality constraints of the interactions without detailing the interactions themselves nor providing an interface to a host language. In this case, the language is used as a tool to prove properties of systems, as done, for example, in process calculi.

1.2 TIME

Some concurrent MoCs have a built-in notion of time. Time provides such a natural conceptual model of concurrency that we might define concurrency in terms of time (using the phrase "at the same time"). A broader definition, however, would be more consistent with the etymology of the word "concurrent," which comes from the Latin *concurrere*, *con-* (together) plus *currere* (to run). Dataflow MoCs are concurrent with no notion of time.

In the discrete-event (DE) MoC, time is an integral part of the model. Events in this MoC will typically carry a *time stamp*, which is an indicator of the time at which the event occurs within the model. A DE simulator will typically maintain a global event queue that sorts events by time stamp. Note that a DE simulator has an internal notion of simulated time that need not correspond to real time. A key to designing successful simulators is to not rely on real time to maintain a correct model of simulated time.

In synchronous languages, the notion of time is more abstract. The word "instantaneously" used above is not to be taken literally. Time progresses in discrete jumps, called *ticks*, rather than continuously as in nature. "Instantaneously" means "at the same tick."

Simulation of hardware designs is typically accomplished using a discrete-event simulator, such as that embodied in VHDL or Verilog simulators. A *signal* is a sequence of events. A time stamp tags each event, giving the set of events an order. The tag may be an integer, a floating-point number, or a data structure representing both the advance of time and possibly the sequencing of *microsteps* within a time instant. In all cases, the job of the simulator is to sort events so that those with the earliest time stamps are processed first, and so that the events seen by any particular component have monotonically increasing time stamps. Time stamps, therefore, define an ordering of events.

Discrete-event modeling can be expensive. The sorting of time stamps can be computationally costly. Moreover, ironically, although discrete-event is ideally suited to modeling distributed systems, it is very challenging to build a parallel discrete-event simulator. The global ordering of events requires much tighter coordination between parts of the simulation than would be ideal for parallel execution.

In an alternative model, events occur synchronously, according to a clock. Events that occur at different clock ticks are globally ordered (one unambiguously precedes the other). Simultaneous events (those at the same clock tick) may be totally ordered, partially ordered, or unordered, depending on the MoC. Unlike the discrete-event model, however, all signals have events at all clock ticks. This results in considerably simpler simulators, because sorting is not required. Simulators that exploit this simpli-

Again, this time on the hardware side, we find heterogeneity. The design styles used to specify and simulate the ASICs and the interconnected commodity components are likely to be quite different, and may not be supported by the same design and simulation tools. A typical system, therefore, not only mixes hardware design with software design, but also mixes design styles within each of these categories.

Two opposing approaches for such system-level design are possible. One is the unified approach, which seeks a consistent semantics for specification of the complete system. The semantics must be rich enough to support heterogeneous design. The other is a heterogeneous approach, which seeks to systematically combine disjoint semantics. Although the intellectual appeal of the unified approach is compelling, we have adopted the heterogeneous approach. We believe that the diversity in design styles commonly used today precludes a unified solution in the foreseeable future.

1.1 MODELS OF COMPUTATION

Key to the heterogeneous approach is the notion of models of computation. A *model of computation* (MoC) is the semantics of the interaction between modules or components. MoCs are used in computer programming as well as in the design of electronic systems. They can be viewed as the organizing principles of a design specification or model. They relate strongly to the *design style*, but may or may not relate strongly to the *implementation technology.* Classes of MoCs include:

Imperative. In an imperative model of computation, modules are executed sequentially to accomplish a task.

Finite state machine (FSM). In an FSM MoC, a specification enumerates the set of states that a system can be in together with the rules for transitioning from one state to another.

Dataflow. In a dataflow MoC, modules react to the availability of data at their inputs by performing some computation and producing data on their outputs. Communication between modules is via *streams*, which are sequences of data *tokens*. Each token is an arbitrary data structure that is treated monolithically by the MoC.

Discrete event. In the discrete-event MoC, modules react to events that occur at a given time instant and produce other events either at the same time instant or at some future time instant. Execution is chronological.

Synchronous languages. In synchronous languages, modules *simultaneously* react to a set of input events and *instantaneously* produce output events. If cyclic dependencies are allowed, then execution involves finding a *fixed point*, or a consistent value for all events at a given time instant.

It is important to recognize the distinction between a MoC and the way that the MoC might be implemented. For example, while the first two of the above MoCs are fundamentally sequential and the last three are fundamentally concurrent, it is possible to use the first two on parallel machines and the last three on sequential machines.

It is also important to recognize the distinction between a model of computation and a *language*. A *syntax* is an important part of a language and not of a MoC. A language may add little more than a syntax to a MoC, but more commonly it will implement more than one MoC. For example, VHDL can be used in an imperative or discrete-event style. Hierarchical FSMs, like Statecharts [16] and at least 22

— Invited paper for RASSP special issue of the *J. on VLSI Signal Processing* —

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720

February 19, 1996

HETEROGENOUS SIMULATION
— MIXING DISCRETE-EVENT MODELS WITH DATAFLOW

W.-T. CHANG, S. HA, and E. A. LEE
Chang and Lee are with the EECS Dept., U. C. Berkeley, USA, and Ha is with the Department of Computer Engineering, Seoul National University, Korea.

Abstract

This paper relates to system-level design of signal processing systems, which are often heterogeneous in implementation technologies and design styles. The heterogeneous approach, by combining small, specialized models of computation, achieves generality and also lends itself to automatic synthesis and formal verification. Key to the heterogeneous approach is to define interaction semantics that resolve the ambiguities when different models of computation are brought together. For this purpose, we introduce a *tagged signal model* as a formal framework within which the models of computation can be precisely described and unambiguously differentiated, and their interactions can be understood. In this paper, we will focus on the interaction between dataflow models, which have partially ordered events, and discrete-event models, with their notion of time that usually defines a total order of events. A variety of interaction semantics, mainly in handling the different notions of time in the two models, are explored to illustrate the subtleties involved. An implementation based on the Ptolemy system from U.C. Berkeley is described and critiqued.

1. Introduction

This paper relates to system-level design of signal processing systems. Such systems are often embedded, and their implementation mixes hardware and software. This inevitably complicates the design process by forcing a heterogeneous approach. Even within the software or hardware portions themselves there is often heterogeneity. In software, control-oriented processes might be mixed under the supervision of a multitasking real-time kernel running in a microcontroller. In addition, hard-real-time tasks may run cooperatively on one or more programmable DSPs. The design styles used for these two software subsystems are likely to be quite different from one another, and testing the interaction between them is unlikely to be trivial.

The hardware side of the design will frequently contain one or more ASICs, perhaps designed using logic or behavioral synthesis tools. On the other hand, a significant part of the hardware design most likely consists of interconnections of commodity components, such as processors and memories.