Chapter 5. DE Domain

Authors: *Joseph T. Buck*
Soonhoi Ha
Paul Haskell
Edward A. Lee
Thomas M. Parks

Other Contributors: *Anindo Banerjea*
Philip Bitar
Rolando Diesta
Brian L. Evans
Richard Han
Ed Knightly
Gregory S. Walter

5.1 Introduction

The discrete event (DE) domain in Ptolemy provides a general environment for time-oriented simulations of systems such as queueing networks, communication networks, and high-level computer architectures. In the domain, each **`Particle`** represents an *event* that corresponds to a change of the system state. The DE schedulers process events in chronological order. Since the time interval between events is generally not fixed, each particle has an associated *time-stamp*. Time stamps are generated by the block producing the particle, using the time stamps of the input particles and the latency of the block.

5.2 The DE target and its schedulers

The DE domain, at this time, has only one target. This target has three parameters:

timeScale — A scaling factor for relating the local simulated time to the time of other domains that might be communicating with DE.

syncMode — An experimental optimization explained below, again aimed at mixed-domain systems.

calendar queue? — A Boolean indicating whether or not to use the faster "calendar queue" scheduler, explained below.

The DE schedulers in Ptolemy determine the order of execution of the blocks. There are two schedulers that have been implemented. They expect particular behavior (operational semantics) on the part of stars. In this section, we describe the semantics.

5.2.1 Events and chronology

A DE star models part of a system response to a change in the system state. The change of state, which is called an *event*, is signaled by a particle in the DE domain. Each particle is assigned a time stamp indicating when (in simulated time) it was generated. Since events are irregularly spaced in time and system responses are generally very dynamic, all scheduling actions are performed at run time. At run time, the DE scheduler processes the events in chronological order until simulated time reaches a global "stop time".

Each scheduler maintains a *global event queue* where particles currently in the system are sorted in accordance with their time stamps. The differences between the two schedulers that are distributed with the domain are primarily in the management of this event queue. Anindo Banerjea and Ed Knightly wrote the default DE Scheduler, which is based on the "calendar queue" mechanism developed by Randy Brown. (This was based on code written by Hui Zhang.) This mechanism handles large event queues much more efficiently than the alternative, more direct DE scheduler, which uses a single sorted list with linear searching. The alternative scheduler can be selected by changing a parameter in the default DE target.

Each scheduler fetches the event at the head of the event queue, which is the earliest event in simulated time, and sends it to the input ports of its destination block. A DE star is executed (fired) whenever there is a new event on any of its input portholes. Before executing the star, the scheduler searches the event queue to find out whether there are any simultaneous events to the other input portholes of the same star, and fetches those events. Thus, for each firing, a star can consume all simultaneous events for its input portholes. After a block is executed it may generate some output events on its output ports. These events are put into the global event queue. Then the scheduler fetches another event and repeats its action until the given stopping condition is met.

It is worth noting that the particle movement is not through `Geodesic`s, as in most other domains, but through the global queue in the DE domain. Since the geodesic is a FIFO queue, we cannot implement the un-chronological arrival of the incoming events if we put the particles into the geodesics. Instead, the particles are managed globally in the event queue.

There is a pitfall in managing time stamps. Two time stamps are not considered equal unless they are exactly equal, to the limit of double-precision floating-point arithmetic. If two time stamps were computed by two separate paths, they are likely to differ in the least significant bits, unless all values in the computation can be represented exactly in a binary representation. If simultaneity is critical in a given application, then exact-precision numbers such as integers should be used for time stamps. This change, however, would require modification of the DE domain.

5.2.2 Event generators

Some DE stars are event generators that do not consume any events, and hence cannot be triggered by input events. They are first triggered by system-generated triggering particles which are placed in the event-queue before the system is executed. Subsequent firings are requested by the star itself, which gives the time at which it wishes to be refired. All such stars are derived from the base class `RepeatStar`.

5.2.3 Simultaneous events

A special effort has been made to handle simultaneous events in a rational way. As noted above, all available simultaneous events at all input ports are made available to a star when it is fired. In addition, if two distinct stars can be fired because they both have events at their inputs with identical time stamps, some choice must be made as to which one to fire. A common strategy is to choose one arbitrarily. This scheme has the simplest implementation, but can lead to unexpected and counterintuitive results from a simulation.

The choice of which to fire is made in Ptolemy by statically assigning priorities to the stars according to a topological sort. Thus, if one of the two enabled stars could produce events with zero delay that would affect the other, as shown in figure 5-1, then that star will be fired first. The topological sort is actually even more sophisticated than we have indicated. It follows triggering relationships between input and output portholes selectively, according to assertions made in the star definition. Thus, the priorities are actually assigned to individual portholes, rather than to entire stars. See the programmer's manual for further details.

5.2.4 Delay-free loops

Many stars in the DE domain produce events with the same time stamps as their input events. These zero-delay stars can create some subtleties in a simulation. An *event-path* consists of the physical arcs between output portholes and input portholes plus zero-delay paths inside the stars, through which an input event instantaneously triggers an output event. If an event-path forms a loop, we call it a *delay-free loop*. While a delay-free loop in the SDF domain results in a deadlock of the system, a delay-free loop in the DE domain potentially causes unbounded computation. Therefore, it is advisable to detect the delay-free loop at compile-time. If a delay-free loop is detected, an error is signaled and simulation is halted.

Detecting delay-free loops reliably is difficult, however. Some stars, such as the **`Server`** and the **`Delay`**, take a parameter that specifies the amount of delay. If this is set to zero, it will fool the scheduler. It is the user's responsibility to avoid this pathological case. This is a special case of a more general problem, in which stars conditionally produce zero-delay events. Without requiring the scheduler to know a great deal about such stars, we cannot reliably detect zero-delay loops. What appears to be a delay-free path can be safe under conditions understood by the programmer. In such situations, the programmer can avoid the error message using a delay element on an arc. The delay element is the small green diamond found at the top of every star palette in Pigi. *It does not actually produce any time delay in simulated time*. Instead, it declares to the scheduler that the arc with the delay element should be treated as if it had a delay, even though it does not. A delay element on a directed loop thus suppresses the detection of a delay-free loop.

5.2.5 Wormholes

"Time" in the DE domain means simulated time. The DE domain may be used in combination with other domains in Ptolemy, even if the other domain does not have a notion of

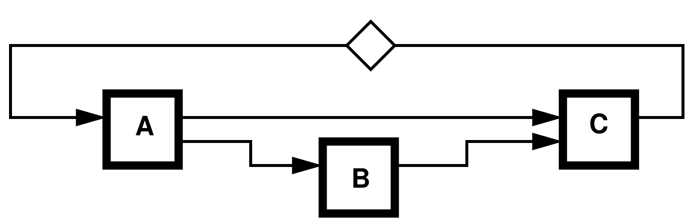

FIGURE 5-1: When DE stars are enabled by simultaneous events, the choice of which to fire is determined by priorities based on a topological sort. Thus if B and C both have events with identical time stamps, B will take priority over C. The delay on the path from C to A serves to break the topological sort.

simulated time. A given simulation, therefore, may involve several schedulers, some of which use a notion of simulated time, and some of which do not. There may also be more than one DE scheduler active in one simulation. The notion of time in the separate schedulers needs to be coordinated. This coordination is specific to the inner and outer domains of the wormhole. Important cases are described below.

SDF within DE

A common combination of domains pairs the SDF domain with the DE domain. There are two possible scenarios. If the SDF domain is inside the DE domain, as shown in figure 5-2, then the SDF subsystem appears to the DE system as a zero-delay block. Suppose, for example, that an event with time stamp T is available at the input to the SDF subsystem. Then when the DE scheduler reaches this time, it fires the SDF subsystem. The SDF subsystem runs the SDF scheduler through one iteration, consuming the input event. In response, it will typically produce one output event, and this output event will be given the time stamp T.

If the SDF subsystem in figure 5-2 is a multirate system, the effects are somewhat more subtle. First, a single event at the input may not be sufficient to cycle through one iteration of the SDF schedule. In this case, the SDF subsystem will simply return, having produced no output events. Only when enough input events have accumulated at the input will any output events be produced. Secondly, when output events are produced, more than one event may be produced. In the current implementation, all of the output events that are produced have the same time stamp. This may change in future implementations.

DE within SDF

The reverse scenario is where a DE subsystem is included within an SDF system. The key requirement, in this case, is that when the DE subsystem is fired, it must produce output events, since these will be expected by the SDF subsystem. A very simple example is shown in figure 5-3. The DE subsystem in the figure routes input events through a time delay. The events at the output of the time delay, however, will be events in the future. The `Sampler` star, therefore, is introduced to produce an output event at the current simulation time. This output event, therefore, is produced before the DE scheduler returns control to the output SDF scheduler.

The behavior shown in figure 5-3 may not be the desired behavior. The `Sampler` star,

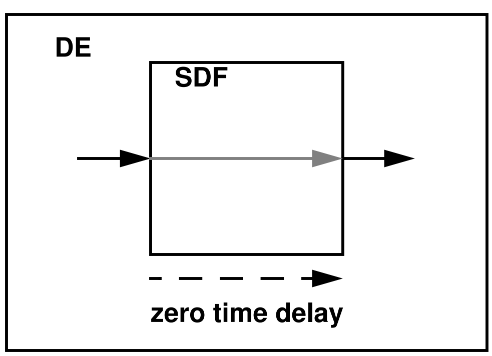

FIGURE 5-2: When an SDF domain appears within a DE domain, from the perspective of the DE domain, events at the input to the SDF subsystem result in zero-delay events at the output of the SDF subsystem. Thus, the time stamps at the output are identical to the time stamps at the inputs.

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 in. The user must understand the semantics of the interacting domains, and act accordingly.

Timed domains within timed domains

The DE domain is a timed domain. Suppose it contains another timed domain (such as Thor) in a DE wormhole. In this case, the inner domain may need to be activated at a given point in simulated time even if there are no new events on its input portholes. Suppose, for instance, that the inner domain contains a clock that internally generates events at regular intervals. Then these events need to be processed at the appropriate time regardless of whether the inner system has any new external stimulus.

The mechanism for handling this situation is simple. When the internal domain is initialized or fired, it can, before returning, place itself on the event queue of the outer domain, much the same way that an event generator star would. This ensures that the inner event will be processed at the appropriate time in the overall chronology. Thus, when a timed domain sits within a timed domain wormhole, before returning control to the scheduler of the outer domain, it requests rescheduling at the time corresponding to the oldest time stamp on its event queue, if there is such an event.

When an internal timed domain is invoked by another time domain, it is told to run until a given "stop time," usually the time of the events at the inputs to the internal domain that triggered the invocation. This "stop time" is the current time of the outer scheduler. Since the inner scheduler is requested to not exceed that time, it can only produced events with time stamp equal to that time. Thus, a timed domain wormhole, when fired, will always either produce no output events, or produce output events with time stamp equal to the simulated time at which it was fired.

To get a time delay through such a wormhole, two firings are required. The first firing is triggered by an input event at time T, say. The inside system generates an internal event at its (future) time $T + \tau$. Before returning control to the outer scheduler, the inner scheduler requests that it be reinvoked at time $T + \tau$. When the "current time" of the outer scheduler reaches $T + \tau$, it reinvokes the inner scheduler, which then produces an output event at time

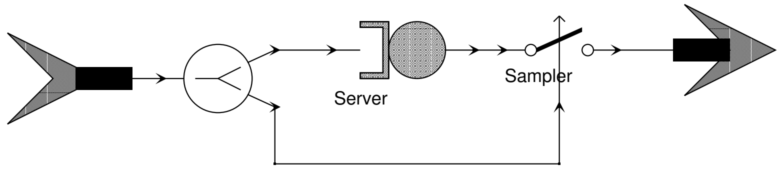

FIGURE 5-3: A typical DE subsystem intended for inclusion within an SDF system. When a DE subsystem appears within an SDF subsystem, the DE subsystem must ensure that the appropriate number of output events are produced in response to input events. This is typically accomplished with a "Sampler" star, as shown.

$T + \tau$.

With this conservative style of timed interaction, we say that the DE domain operates in the *synchronized mode*. Synchronized mode operation suffers significant overhead at run time, since the wormhole is called at every time increment in the inner timed domain. Sometimes, however, this approach is too conservative.

In some applications, when an input event arrives, we can safely execute the wormhole into the future until either (a) we reach the time of the next event on the event queue of the outer domain, or (b) there are no more events to process in the inner domain. In other words, in certain situations, we can safely ignore the request from the output domain that we execute only up until the time of the input event. As an experimental facility to improve run-time efficiency, an option avoids synchronized operation. Then, we say that the DE domain operates in the *optimized mode*. We specify this mode by setting the target parameter *syncMode* to **`FALSE`** (zero). This should only be done by knowledgeable users who understand the DE model of computation very well. The default value of the *syncMode* parameter is **`TRUE`** (one), which means synchronized operation.

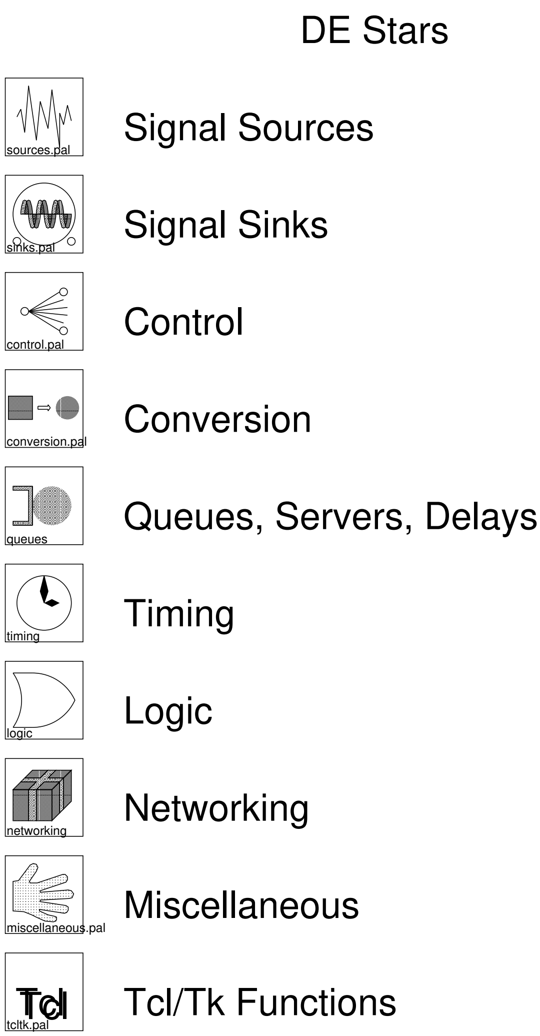

FIGURE 5-4: The top-level palette of discrete-event stars.

5.3 An overview of stars in DE

The model of computation in the DE domain makes it amenable to high-level system modeling. For this reason, stars in the DE domain are often more complicated, and more specialized than those in the SDF domain. The stars that are distributed with the domain, therefore, should be viewed primarily as examples. They do not form a comprehensive set.

We have made every attempt to include in the distribution all of the reasonably generic stars that have been developed, plus a selection of the more esoteric ones (as examples). Keep in mind that the star libraries of the other domains are also available through the wormhole mechanism. Users that find themselves frequently needing stars from other domains may wish to build a library of single-star galaxies. Such galaxies can be used in any domain, regardless of the domain in which the single star resides. Ptolemy automatically implements this as a wormhole.

The top-level palette is shown in figure 5-4. The following star is available in all the palettes:

`BlackHole` Discard all input particles.

5.3.1 Source stars

Strictly speaking, source stars are stars with no inputs. They generate signals, and may represent external inputs to the system, constant data, or synthesized stimuli. By convention, these stars are fired once at time zero automatically. During this and all subsequent firings, the star itself must determine when its next firing should occur. It schedules this next firing with a call to the method **`refireAtTime(`*`time`*`)`**. The source palette is shown in figure 5-5.

`Clock`	Generate events at regular intervals, starting at time zero.
`Impulse`	Generate a single event at time zero.
`Null`	Do nothing. It is useful for connecting to any unused input ports.
`Poisson`	Generate events according to a Poisson process. The first event comes out at time zero. The mean inter-arrival time and magnitude of the events are given as parameters.
`PulseGen`	Generate events with specified values at specified moments. The events are specified in the *value* array, which consists of time-value pairs, given in the syntax of complex numbers.

For convenience, some stars are included in the source palette that are not really source stars, in the above sense. They require an input event in order to produce an output. These are listed below. The value of the input event is ignored; it is only its time stamp that matters.

`Const`	Produce an output event with a constant value (the default value is zero) when stimulated by an input event. The time stamp of the output is the same as that of the input.
`Ramp`	Produce an output event with a monotonic value when stimu-

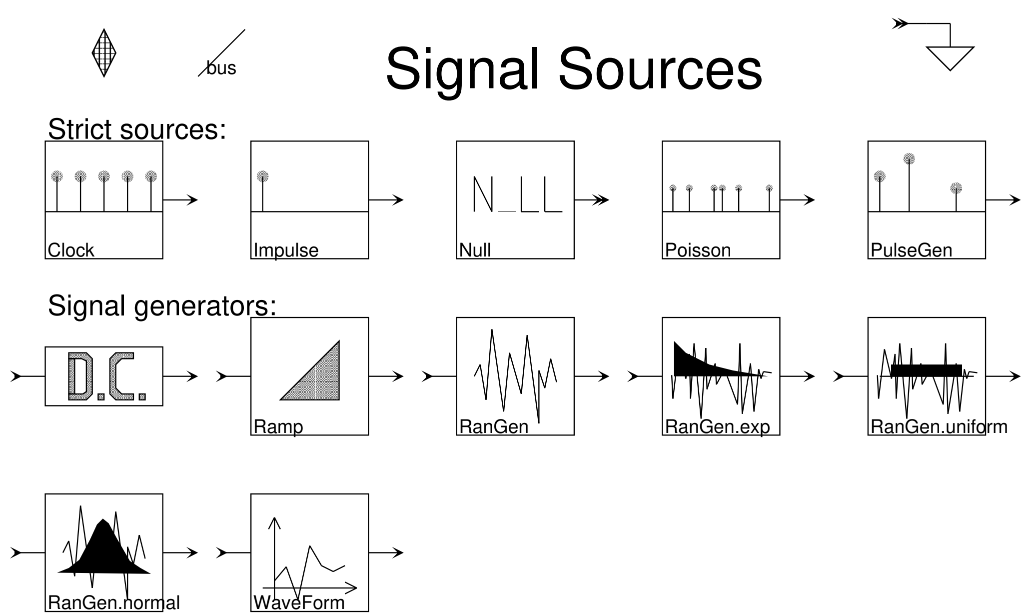

FIGURE 5-5: Source stars in the DE domain.

lated by an input event. The value of the output event starts at "value" and increases by "step" each time the star fires.

`RanGen` (four icons) Generate a sequence of random numbers. Upon receiving an input event, it generates a random number with uniform, exponential, or normal distribution, as determined by the "distribution" parameter. Depending on the distribution, parameters also specify either the mean and variance or the lower and upper extent of the range.

`WaveForm` Upon receiving an input event, output the next value specified by the array parameter *value* (default "1 -1"). This array can periodically repeat with any period, and you can halt a simulation when the end of the array is reached. The following table summarizes the capabilities:

haltAtEnd	*periodic*	*period*	*operation*
NO	YES	0	The period is the length of the array
NO	YES	N>0	The period is N
NO	NO	anything	Output the array once, then zeros
YES	anything	anything	Stop after outputting the array once

The first line of the table gives the default settings. The array may be read from a file by simply setting *value* to something of the form "< filename"

5.3.2 Sink stars

The sink stars pointed to by the palette in figure 5-6 are those with no outputs. They display signals in various ways, or write them to files. Several of the stars in this palette are based on the **`pxgraph`** program. This program has many options, summarized in "pxgraph — The Plotting Program" on page 7-1. The differences between stars often amount to little more than the choice of default options. Some, however, preprocess the signal in useful ways before passing it to the **`pxgraph`** program. The first five icons actually correspond to only one star, with five different configurations. The first allows only one input signal, the second allows two, etc., until the fifth, which allows any number (notice the double arrow on the input port).

`BarGraph` (five icons) Generate a plot with the **`pxgraph`** program that uses a zero-order hold to interpolate between event values. Two points are plotted for each event, one when the event first occurs, and the second when the event is supplanted by a new event. A horizontal line then connects the two points. If *draw_line_to_base* is YES then a vertical line to the base of the bar graph is also drawn for each event occurrence.

`Beep` Cause a beep on the terminal when fired.

`Printer` Print the value of each arriving event, together with its time of arrival. The *fileName* parameter specifies the file to be written; the special names <stdout> and <cout> (specifying the standard output stream), and <stderr> and <cerr> specifying the standard error stream, are also supported.

Xhistogram Generate a histogram with the **pxgraph** program. The parameter *binWidth* determines the width of a bin in the histogram. The number of bins will depend on the range of values in the events that arrive. The time of arrival of events is ignored. This star is identical to the SDF **Xhistogram**, but is used often enough in the DE domain that it is provided here for convenience.

XMgraph (five icons) Generate a plot with the **pxgraph** program with one point per event. Any number of event sequences can be plotted simultaneously, up to the limit determined by the **XGraph** class. By default, a straight line is drawn between each pair of events.

5.3.3 Control stars

Control stars (figure 5-7) manipulate the flow of tokens. All of these stars are polymorphic; they operate on any data type. From left to right, top to bottom, they are:

Discard Discard input events that occur before the threshold time. Events after the threshold time are passed immediately to the output. It is useful for removing transients and studying steady-state effects.

Fork (three icons) Replicate input events on the outputs with zero delay.

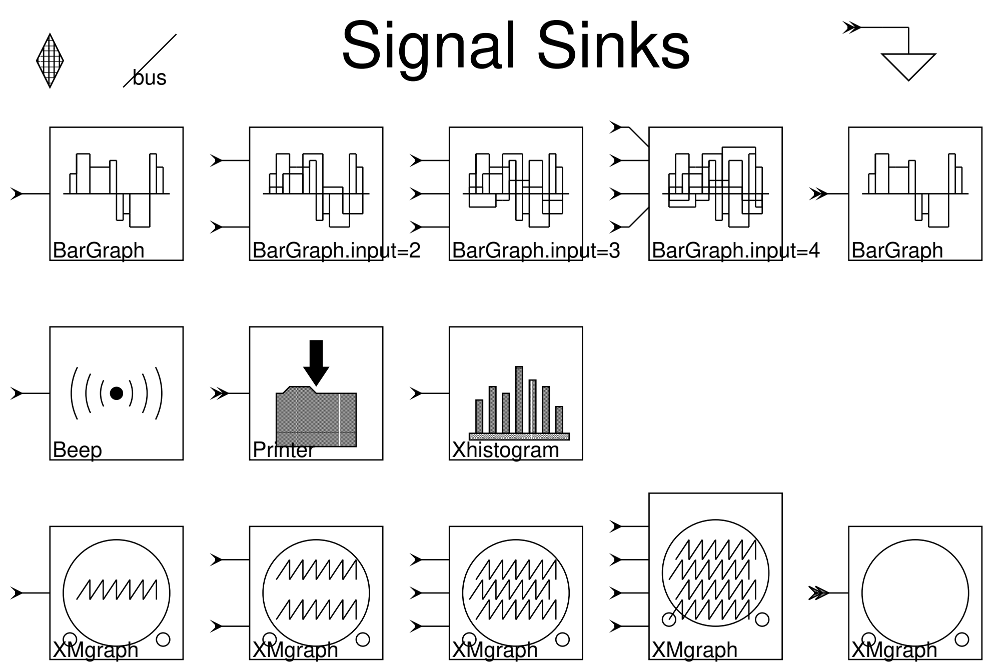

FIGURE 5-6: Sink stars in the DE domain.

`LeakBucket`	Discard inputs that arrive too frequently. That is, any input event that would cause a queue of a given size followed by a server with a given service rate to overflow are discarded. Inputs that are not discarded are passed immediately to the output.
`LossyInput`	Route inputs to its "sink" output with the probability *lossProbability* set by the user. All other inputs are sent immediately to the "save" output.
`Merge`	(three icons) Merge input events, keeping temporal order.
`PassGate`	If the gate (bottom input) is open, then particles pass from "input" (left input) to "output". When the gate is closed, no outputs are produced. If "input" particles arrive while the gate is closed, the most recent one will be passed to "output" when the gate is reopened.
`Router`	Route an input event randomly to one of its outputs. The probability is equal for each output. The time delay is zero.
`Sampler`	Sample the input at the times given by events on the "clock" input. The data value of the "clock" input is ignored. If no input is available at the time of sampling, the latest input is used. If there has been no input, then a "zero" particle is produced. The exact meaning of zero depends on the particle type.
`Switch`	Switch input events to one of two outputs, depending on the last received control input.

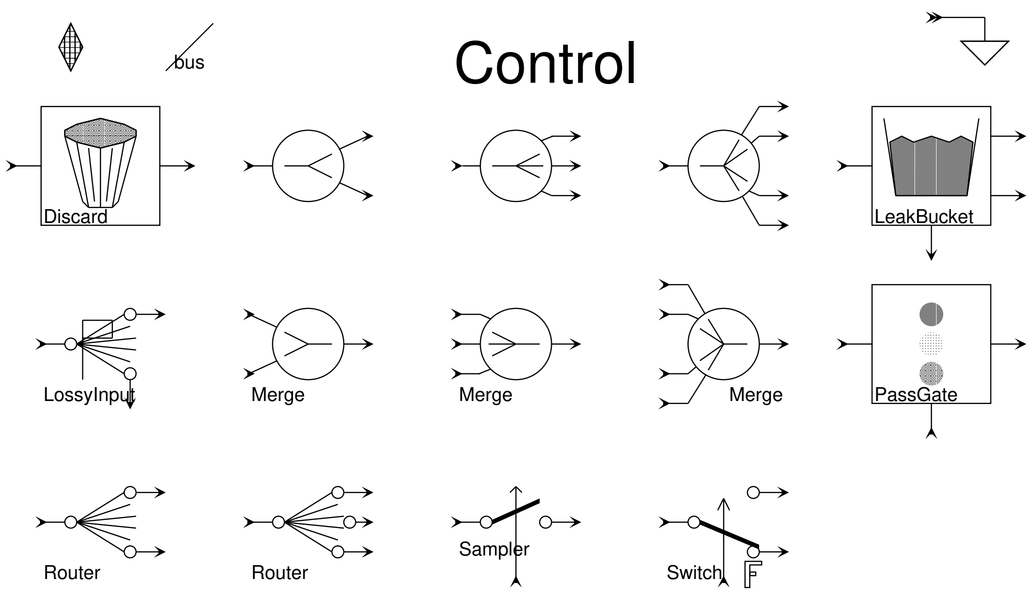

FIGURE 5-7: Control stars for the DE domain.

5.3.4 Conversion stars

The palette in figure 5-8 is intended to house a collection of stars for format conversions of various types. As of this writing, however, this collection is very limited. The two stars in the conversion palette illustrate the consolidation of multiple data sample into single particles that can be transmitted as a unit. These stars use the class **`FloatVecData`**, which is simply a vector of floating-point numbers.

`Packetize` Convert a number of floating-point input samples into a stream of a packet of type **`FloatVecData`**. A packet is produced when either an input appears on the demand input or when *maxLength* data values have arrived. Note that a null packet is produced if a demand signal arrives and there is no data.

`UnPacketize` Convert a stream of packet of type **`FloatVecData`** into a number of floating output samples. The "data" input feeds packets to the star. Whenever a packet arrives, the previous packet, if any, is discarded; any remaining contents are discarded. The "demand" input requests output data. If there is no data left in the current packet, the last output datum is repeated (zero is used if there has never been a packet). Otherwise the next data value from the current input packet is output.

5.3.5 Queues, servers, and delays

The palette in figure 5-9 contains stars that model queues, servers, and time delays of various types. In the DE domain, the delay icon (the small green diamond at the upper left of the palette) does not represent a time delay. See "The DE target and its schedulers" on page 5-1.

`Delay` Send each input event to the output with its time stamp incremented by an amount given by the *delay* parameter.

`VarDelay` Delay the input by a variable amount. The *delay* parameter gives the initial delay, and the delay is changed using the "newDelay" input.

`Server` Emulate a server. If input events arrive when it is not busy, it delays them by the service time (a constant parameter). If they arrive when it is busy, it delays them the service time plus however long it takes to become free of previous tasks.

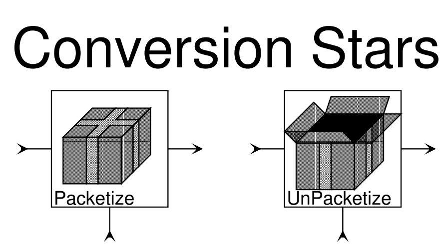

FIGURE 5-8: Type conversion stars for the DE domain.

VarServer Emulate a server with a variable service time. If input events arrive when it is idle, they will be serviced immediately and will be delayed only by the service time. If input events arrive while another event is being serviced, they will be queued. When the server becomes free, it will service any events waiting in its queue.

PSServer Emulate a deterministic, processor-sharing server. If input events arrive when it is not busy, it delays them by the nominal service time. If they arrive when it is busy, the server is shared. Hence prior arrivals that are still in service will be delayed by more than the nominal service time.

FIFOQueue Implement a first-in first-out (FIFO) queue with finite or infinite length. Events on the "demand" input trigger a dequeue on the "outData" port if the queue is not empty. If the queue is empty, then a "demand" event enables the next future "inData" particle to pass immediately to "outData". The first particle to arrive at "inData" is always passed directly to the output, unless *numDemandsPending* is initialized to 0. If *consolidateDemands* is set to TRUE (the default), then *numDemandsPending* is not permitted to rise above unity. The size of the queue is sent to the *size* output whenever an "inData" or "demand" event is processed. Input data that doesn't fit in the queue is sent to the "overflow" output.

Stack Implement a stack with either finite or infinite length. Events on the "demand" input pop data from the stack to "outData" if the stack is not empty. If it is empty, then a "demand" event enables the next future "inData" particle to pass immediately to "outData". By default, *numDemandsPending* is initialized to 1, so the first particle to arrive at "inData" is passed directly to the

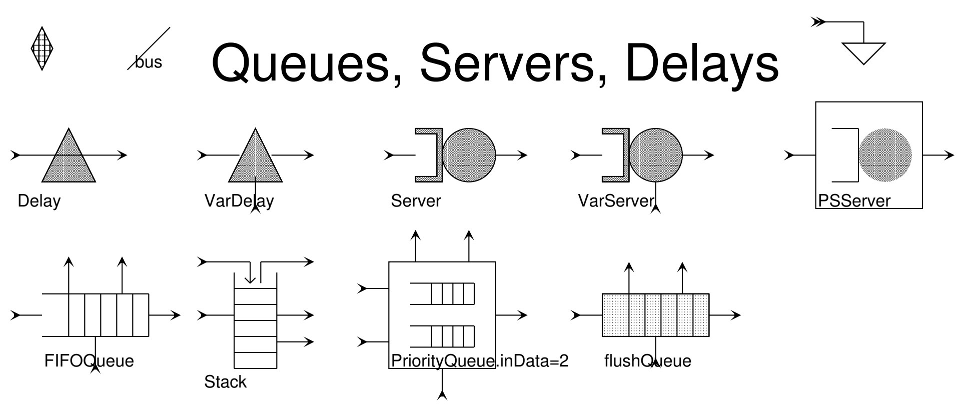

FIGURE 5-9: Queues, servers, and delays in the DE domain.

output. If *consolidateDemands* is set to TRUE (the default), then *numDemandsPending* is not permitted to rise above unity. The size of the stack is sent to the “size” output whenever an “inData” or “demand” event is processed. Input data that doesn’t fit on the stack is sent to the “overflow” output.

`PriorityQueue` Emulate a priority queue. Inputs have priorities according to the number of the input, with “inData#1” having highest priority, “inData#2” being next, etc. When a “demand” is received, outputs are produced by selecting first based on priority, and then based on time of arrival, using a FIFO policy. A finite total capacity can be specified by setting the “capacity” parameter to a positive integer. When capacity is reached, further inputs are sent to the “overflow” output, and not stored. The *numDemandsPending* and *consolidateDemands* parameters have the same meaning as in other queue stars. The size of the queue is sent to the “size” output whenever an “inData” or “demand” event is processed.

`FlushQueue` Implement a FIFO queue that when full, discards all inputs until it empties completely.

The following star does not appear in the palette.

`QueueBase` Serve as the base class for FIFO and LIFO queues. This star is not intended to be used except to derive useful stars. All inputs are simply routed to the “overflow” output. None are stored.

5.3.6 Timing stars

The palette in figure 5-8 contains stars that are primarily concerned with either simulated or real time.

`MeasureDelay` Measure the time difference between the first arrival and the second arrival of an event with the same value. The second arrival and the time difference are each sent to outputs.

`StopTimer` Generate an output at the **`stopTime`** of the **`DEScheduler`** under which this block is running. This can be used to force actions at the end of a simulation. Within a wormhole, it can

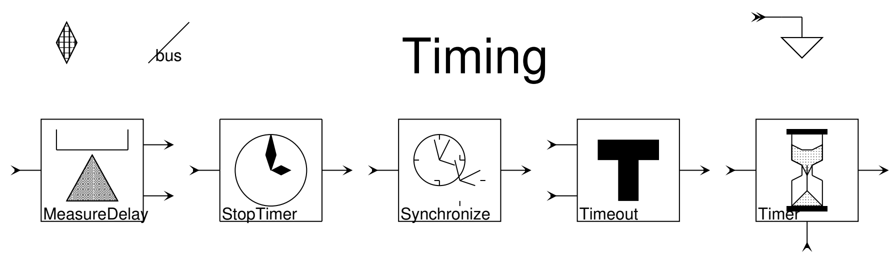

FIGURE 5-10: Timing stars for the DE domain.

used to force actions at the end of each invocation of the wormhole. An input event is required to enable the star.

`Synchronize` Hold input events until the time elapsed on the system clock since the start of the simulation is greater than or equal to their time stamp. Then they are passed to the output.

`Timeout` Detect time-out conditions and generates an alarm if too much time elapses before resetting or stopping the timer. Events arriving on the "Set" input reset and start the timer. Events arriving on the "Clear" input stop the timer. If no "Set" or "Clear" events arrive within *timeout* time units of the most recent "Set", then that "Set" event is sent out the "alarm" output.

`Timer` Upon receiving a trigger input, output the elapsed real time in seconds, divided by *timeScale*, since the last reset input, or since the start of the simulation if no reset has been received. The time stamp of the output is the same as that of the trigger input. The time in seconds is related to the scheduler (simulated) time through the scaling factor "timeScale".

The following star does not appear in the palette, because it is not intended to be used directly in Ptolemy applications.

`TimeoutStar` Serve as the base class for stars that check time-out conditions. The methods "set", "clear", and "expired" are provided for setting and testing the timer.

5.3.7 Logic stars

The logic palette in figure 5-8 is made up of only three stars, with the multiplicity of icons representing simply different configurations of these stars.

`Test` Compare two inputs. The test condition can be any of {EQ NE LT LE GT GE} or {== != < <= > >=}, resulting in equals, not equals, less-than, less-than or equals, etc.

If *crossingsOnly* is TRUE, then an output event is generated only when the value of the output changes. Hence the output events will always alternate between true and false.

`Logic` Apply a logical operation to any number of inputs. The inputs are integers interpreted as Booleans, where zero is a FALSE and nonzero is a TRUE. The logical operations supported are {NOT AND NAND OR NOR XOR XNOR}.

`TestLevel` Detect threshold crossings if "crossingsOnly" is TRUE. Otherwise, it simply compares the input against the "threshold".

If "crossingsOnly" is TRUE, then: (1) A TRUE is sent to "output" when the "input" particle exceeds or equals the "threshold" value, having been previously smaller. (2) A FALSE is sent when the "input" particle is smaller than "threshold" having been previously larger. Otherwise, no output is produced.

If "crossingsOnly" is FALSE, then a TRUE is sent to "output" whenever any "input" particle greater than or equal to "threshold" is received, and a FALSE is sent otherwise.

5.3.8 Networking stars

The palette shown in figure 5-12 includes stars that have been designed to model communication networks. These are illustrative of a common use of the DE domain, for modeling packet-switched networks. However, many of the stars are specialized to a particular type of network design. Thus, they should be viewed as illustrative examples, rather than as a comprehensive library.

A **`NetworkCell`** class is used in many of these stars. It models packetized data that is transmitted through cell-relay networks. Each **`NetworkCell`** object can carry any user data of type **`Message`**. In addition to this user data, the **`NetworkCell`** contains a destination address and a priority. These are used by stars and galaxies to route the cell through the network. The definition of the **`NetworkCell`** class may be found in **`$PTOLEMY/src/domains/sdf/image/kernel`**, since it is used in the SDF and DE domains, and was developed primarily for modeling packet-switched video.

Cell creation and access

`CellLoad` Read in an **`Envelope`**, extract its **`Message`**, and output that **`Message`** in a **`NetworkCell`**. Append a destination and priority to the packet.

`CellUnload` Remove a **`Message`** from a **`NetworkCell`**.

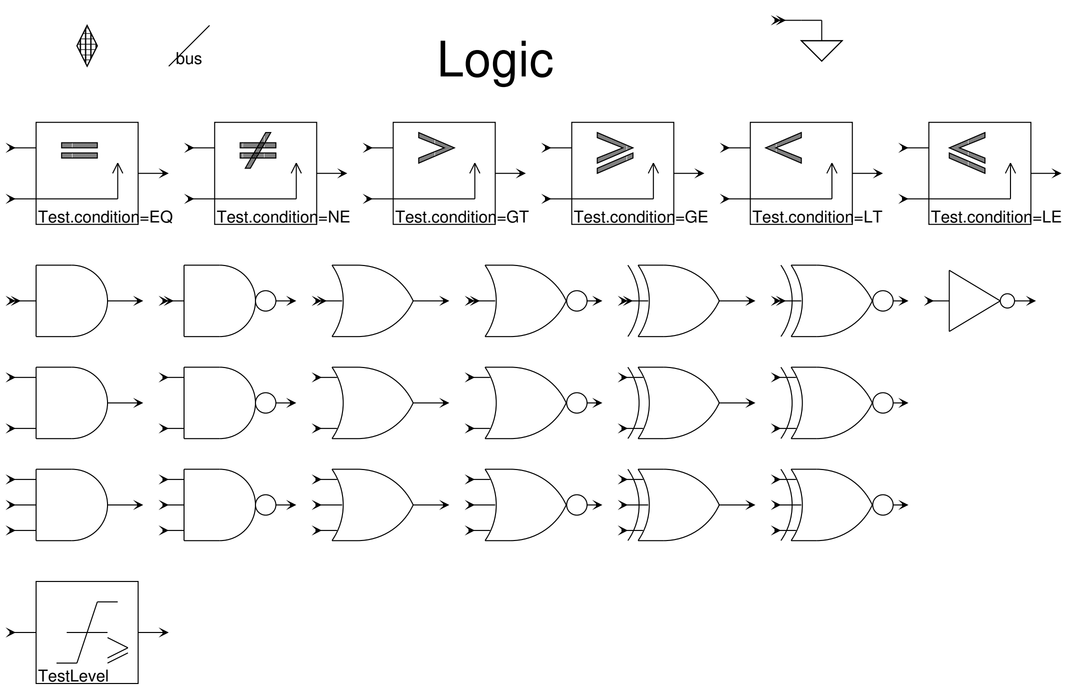

FIGURE 5-11: Logic stars for the DE domain.

`CellToImage` Read **`NetworkCell`**s containing image data and outputs whole images. The current image is sent to the output when the star reads image data with a higher frame id than the current image. For each frame, the fraction of input data that was lost is sent to the "lossPct" output.

`ImageToCell` Packetize an image. Each image is divided up into chunks no larger than *CellSize*. Each cell is delayed from its predecessor by *TimePerCell*. If a new input arrives while an older one is being processed, the new input is queued.

Cell routing, control, and service

`CellRoute` Read **`NetworkCell`**s from multiple input sources and routes them to the appropriate output using a routing table that maps addresses into output ports.

`PriorityCheck` Read **`NetworkCell`** packets from multiple input sources. If the priority of an input **`NetworkCell`** is less than the most recent "priority" input, then the cell is sent to the "discard" output. Otherwise it is sent to the "output" port.

`Switch4x4` Implement a four-input, four-output network switch that can process objects of type **`NetworkCell`**, or any type derived from **`NetworkCell`**. Each **`NetworkCell`** object contains a destination address. This galaxy uses the destination address as an index into its *Routes* array parameter to choose an output port over which the input object will leave. A prioritized queueing scheme is used.

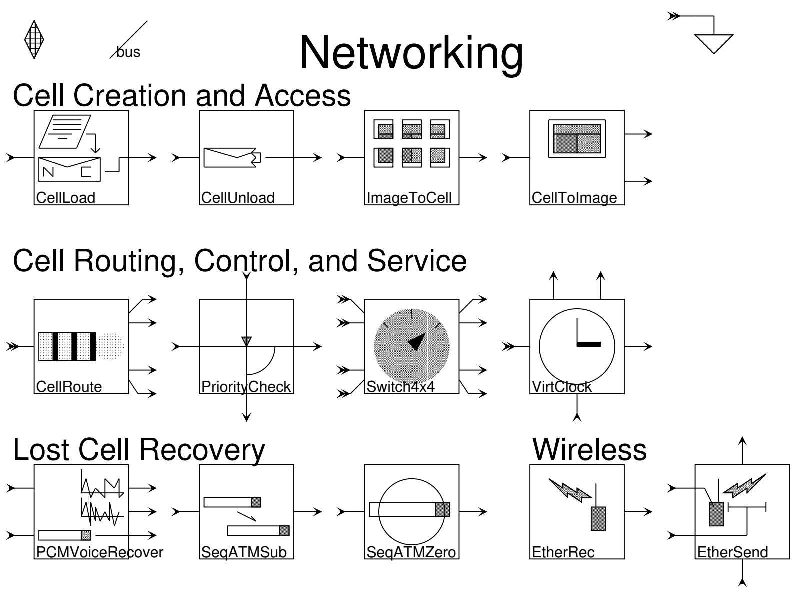

FIGURE 5-12: A palette of DE stars dedicated to modeling of communication networks.

VirtClock Read a **NetworkCell**. It identifies which virtual circuit number the cell belongs to and then computes the virtual time stamp for the cell by applying the (special!) virtual clock algorithm (see the full manual page). It then outputs all cells in order of increasing virtual time stamp.

Upon receiving a “demand” input, the cell with the smallest time stamp is output. An output packet is generated for every demand input unless all of the queues are empty. Demand inputs arriving when all queues are empty are ignored.

The number of stored cells is output after the receipt of each “input” or “demand”.

When a cell arrives and the number of stored cells equals *MaxSize* then the cell with the biggest virtual time stamp is discarded. This cell may or may not be the new arrival. If *MaxSize* is zero or negative, then infinitely many cells can be stored.

Lost cell recovery

The stars in this subgroup implement a variety of mechanism for replacing lost cells in a packet-switched network. They use a class called **SeqATMCell** that is designed to model packets in the proposed broadband integrated services digital network (BISDN). The class is derived from **Message**, but has added facilities for marking the packet with a sequence number, and setting and reading individual bits. The sequence number is used to determine when packets have been lost.

PCMVoiceRecover Input a stream of **SeqATMCell** objects. All the information bits in objects received with correct sequence numbers are sent to “output”.

If a missing **SeqATMCell** object is detected, this star sends the most recent 8 * *tempSize* received bits to the “temp” output, and the most recent (8 * *searchWindowSize* + *numInfoBits*) received bits to the “window” output.

The bits output on the “window” and “temp” outputs can be used by the **PatternMatch** galaxy to implement lost-speech recovery.

SeqATMSub Reads a sequence of **SeqATMCells**. It will check sequence numbers, and if a **SeqATMCell** is found missing, the information bits of the previously arrived **SeqATMCell** will be output in its place.

The information bits from each correctly received **SeqATMCell** are unloaded and sent to the output port.

SeqATMZero Read a sequence of **SeqATMCell** objects. For each object input correctly in sequence, *headerLength* bits are skipped over and the next *numInfoBits* bits in the cell are output.

If this star finds that a **SeqATMCell** is missing by checking sequence numbers, it will substitute *numInfoBits* 0-bits for the missing bits.

Wireless network simulation

Ether (not shown in the palette) This is the base class for transmitter and receiver stars that communicate over a shared medium. Each transmitter can communicate with any or all receivers that have the same value for the "medium" parameter. The communication is accomplished without graphical connections, and the communication topology can be continually changing. This base class implements the data structures that are shared between the transmitters and receivers.

EtherRec Receive floating-point particles transmitted to it by an **EtherSend** star. The particle is produced at the output after some duration of transmission specified at the transmitter.

EtherSend Transmit particles of any type to any or all receivers that have the same value for the "medium" parameter. The receiver address is given by the "address" input, and it must be an integer. If the integer is negative, then copies of the particle are sent to all receivers that use the same medium.

The transmitter "occupies" the medium for the specified duration. A collision occurs if the medium is occupied when a transmission is requested. In this case, the data to be transmitted is sent to the "collision" output.

5.3.9 Tcl stars

This palette contains the (still small) collection of DE stars that use Tcl and Tk. The **TclScript** star is the most generic Tcl star, with no useful function on its own. It must have a Tcl script associated with it to make it useful. There is a chapter of the Programmer's Manual of the The Almagest devoted to how to write such scripts. The palette of Tcl stars is shown in figure 5-13.

TkBarGraph (five icons) Take any number of inputs and dynamically displays their values in bar-chart form.

TclScript (14 icons) Invoke a Tcl script that can optionally define a procedure that is invoked every time the star fires. That procedure can read the star's inputs and update the value of the outputs.

5.3.10 Miscellaneous stars

These stars are shown in figure 5-12.

Hardware modeling

Arbitrate Acts as a non-preemptive arbitrator, granting requests for exclusive control. If simultaneous requests arrive, priority is given to port A. When control is released, any pending requests on the other port will be serviced. The “requestOut” and “grantIn” connections allow interconnection of multiple arbitration stars for more intricate control structures.

HandShake Cooperate with a (possibly preemptive) arbitrator through the

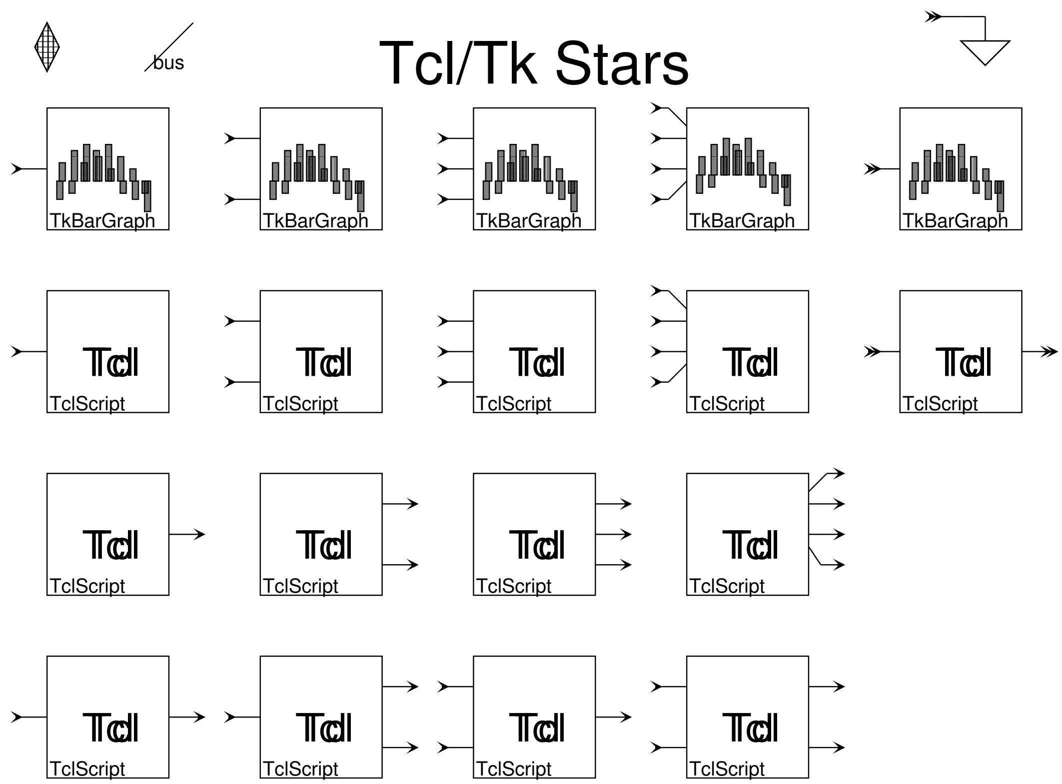

FIGURE 5-13: The Tcl/Tk palette.

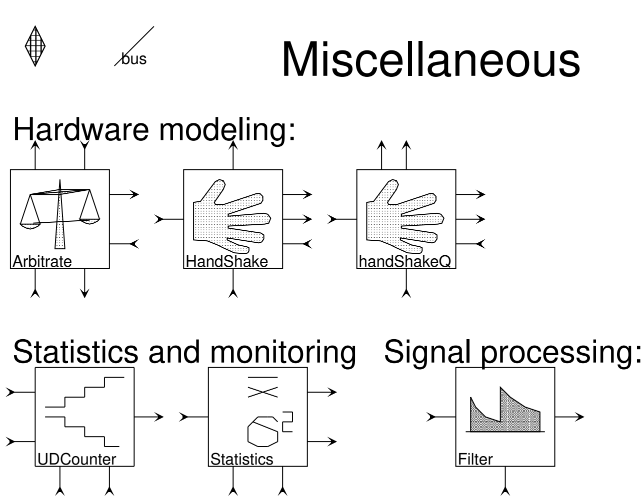

FIGURE 5-14: A palette miscellaneous DE stars.

"request" and "grant" controls. "Input" particles are passed to "output", and an "ackIn" particle must be received before the next "output" can be sent. This response is made available on "ackOut".

`handShakeQ` Handshake with queued input events.

Statistics and monitoring

`Statistics` Calculate the average and variance of the input values that have arrived since the last reset. An output is generated when a "demand" input is received. When a "reset" input arrives, the calculations are restarted. When "demand" and "reset" particles arrive at the same time, an output is produced before the calculations are restarted.

`UDCounter` Implement an up/down counter. The processing order of the ports is: countUp -> countDown -> demand -> reset. Specifically, all simultaneous "countUp" inputs are processed. Then all simultaneous "countDown" inputs are processed. If there are multiple simultaneous "demand" inputs, all but the first are ignored. Only one output will be produced.

Signal processing

`Filter` Filter the input signal with a first-order, autoregressive (AR) impulse response. The data input is interpreted as weighted impulses (Dirac delta functions). An output is triggered by a clock input.

5.4 An overview of DE demos

The number of DE demos is considerably smaller than SDF. Many of the demos, however, are much more complex, often incorporating SDF subsystems to accomplish audio or video encoding. The top-level palette for demos in the discrete-event domain is shown in figure 5-15. The subpalettes are described below.

5.4.1 Basic demos

These demos illustrate the use of certain stars without necessarily performing functions that are particularly interesting. The palette is shown in figure 5-16. The individual demos are summarized below.

`conditionals` Demonstrate the use of the **`Test`** block in it various configurations to compare the values of input events with floating-point values. The input test signal is a pair of ramps, with each event repeated once after some delay. Since the ramps have different steps, they will cross.

`logic` Demonstrate the use of the **`Logic`** star in its various instantiations as AND, NAND, OR, NOR, XOR, XNOR and inverter gates. The three test signals consist of square waves with periods 2, 4, and 6.

`merge` Demonstrate the **`Merge`** star. The star is fed two streams of regular arrivals, one a ramp beginning at 10.0, and one a ramp beginning at 0.0. The two streams are merged into one, in chronological order, with simultaneous events appearing in arbitrary order.

`realTime` Demonstrate the use of the **`Synchronize`** and **`Timer`** blocks. Input events from a **`Clock`** star are held by the **`Synchronize`** star until their time stamp, multiplied by the universe parameter

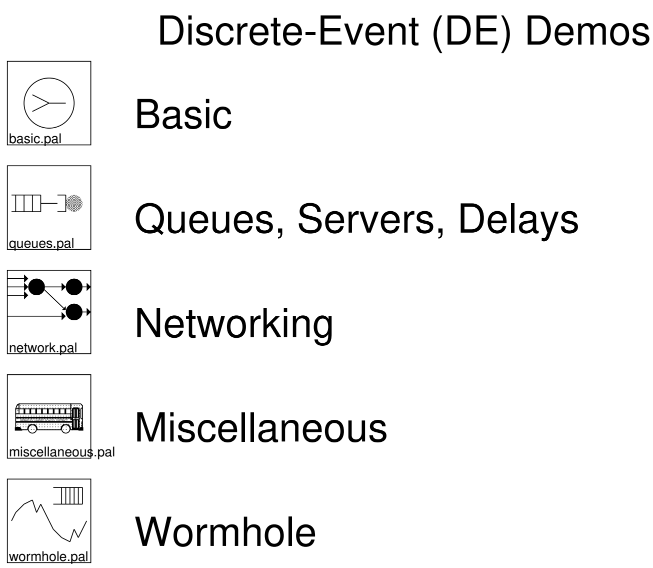

FIGURE 5-15: The top-level palette for DE demos.

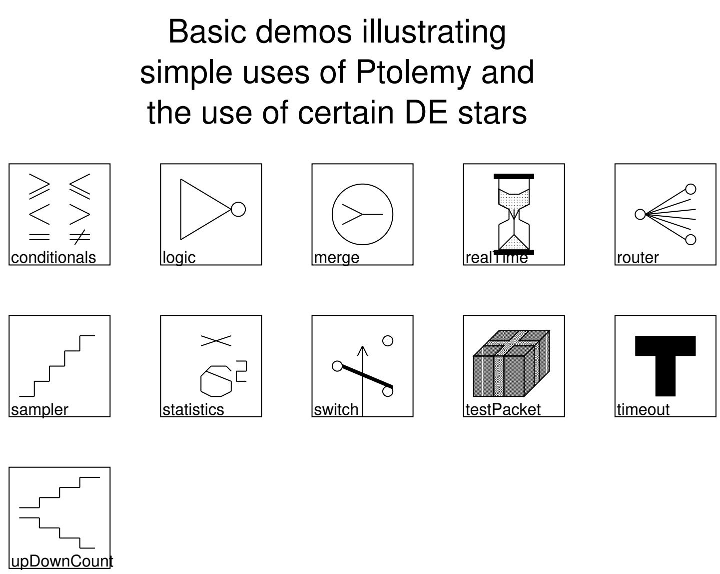

FIGURE 5-16: Basic DE demos.

timeScale, is equal to or larger than the elapsed real time since the start of the simulation. The **`Timer`** star then measures the actual (real) time at which the **`Synchronize`** output is produced. The closer the resulting plot is to a straight line with a slope of one, the more precise the timing of the **`Synchronize`** outputs are.

`router` Randomly route an irregular but monotonic signal (a Poisson counting process) through two channels with random delay, and merge the channel outputs.

`sampler` Demonstrate the **`Sampler`** star. A counting process with regular arrivals at intervals of 5.0 is sampled at regular intervals of 1.0. As expected, this produces 5 samples for each level of the counting process.

`statistics` Compute the mean and variance of a random process using the **`Statistics`** star. The mean and variance are sent to the standard output when the simulation stops. This action is triggered by an event produced by the **`StopTimer`** star.

`switch` Demonstrate the use of the **`Switch`** star. A Poisson counting process is sent to one output of the switch for the first 10 time units, and to the other output of the switch for the remaining time.

`testPacket` Construct packets consisting of five sequential values from a ramp, sends these packets to a server with a random service time, and then deconstructs the packets by reading the items in the packet one by one.

`timeout` Demonstrate the use of the **`Timeout`** star. Every one time unit, a timer is set. If after another 0.5 time units have elapsed the timer is not cleared, an output is produced to indicate that the timer has expired. The signal that clears the timer is a Poisson process with a mean interarrival time of one time unit.

`upDownCount` Demonstrate the **`UDCounter`** star, an up/down counter. Events are generated at two different rates to count up and down. The up rate is faster than the down rate, so the trend is upwards. The value of the count is displayed every time it changes.

5.4.2 Queues, servers, and delays

The palette of demos illustrating queueing systems is shown in figure 5-17. It includes:

`delayVsServer` Illustrate the difference between the **`Delay`** and **`Server`** blocks. The **`Delay`** passes the input events to the output with a fixed time offset. The **`Server`** accepts inputs only after the previous inputs have been served, and then holds that input for a fixed offset.

`queue` Demonstrate the use of the **`FIFOQueue`** and **`Stack`** stars. A Poisson counting process is enqueued on both stars, and is dequeued at a regular rate, every 1.0 time units. The output of the **`FIFOQueue`** is always monotonically increasing, because of the FIFO policy, but the output of the **`Stack`** need not be. The Stack is set with a smaller capacity than the **`FIFOQueue`**, so it overflows first. Overflow events are displayed.

`qAndServer` Demonstrate the use of the **`FIFOQueue`** and **`Stack`** stars together with **`Server`**s. A regular counting process is enqueued on both stars. The particles are dequeued whenever the server is free. The **`Stack`** is set with a larger capacity than the **`FIFOQueue`**, so it overflows second. Overflow events are displayed.

`priority` Demonstrates the use of the **`PriorityQueue`** block together with a **`Server`**. The upper input to the **`PriorityQueue`** has priority over the lower input. Thus, when the queue overflows, data is lost from the lower input.

`testServers` Demonstrate servers with random service times (uniform and exponential).

`psServer` Demonstrate the processor-sharing server. Unlike other servers, this server accepts new inputs at any time, regardless of how busy it is. Accepting a new input, however, slows down the service to all particles currently being served.

`blockage` Demonstrate a blocking strategy in a queueing network. In a cascade of two queues and servers, when the second queue fills up, it prevents any further dequeueing of particles from the first queue until it once again has space.

`measureDelay` Demonstrate the use of the **`MeasureDelay`** block to measure the sojourn time of particles in a simple queueing system with a single server with a random service time.

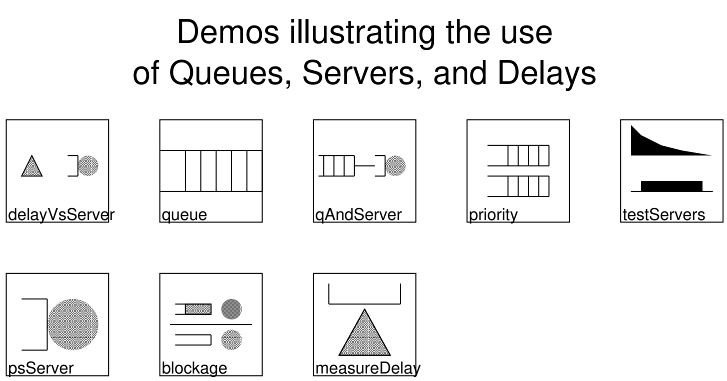

FIGURE 5-17: Queueing system demos

5.4.3 Networking demos

A major application of the DE domain is the simulation of communication networks. The palette in figure 5-18 contains such network simulations. Left to right, top to bottom, the demos are:

`aloha` Simulate the (pure) ALOHA multiple access protocol. The simulation models a composite of independent Poisson sources transmitting on a common channel. All overlapping transmissions are lost and are retransmitted after a random delay. This simulation is built out of very low-level DE stars, and its complexity illustrates very clearly that this is not a reasonable approach.

`FlushNet` Simulate a queue with "input flushing" during overflow. If the queue reaches capacity, all new arrivals are discarded until all items in the queue have been served.

`LBTest` Simulate leaky bucket network rate controllers. These controllers moderate the flow of packets to keep them within specified rate and burstiness bounds.

`VClock` Model a network with four inputs and virtual clock buffer service.

`wireless` Demonstrate shared media communication without graphical connectivity, using **`EtherSend`** and **`EtherRec`** stars. Two clusters on the left transmit to two clusters on the right over two distinct media, radio and infrared. The communication is implemented using shared data structures between the stars.

The following demonstrations all focus on various aspects of ATM (asynchronous transfer

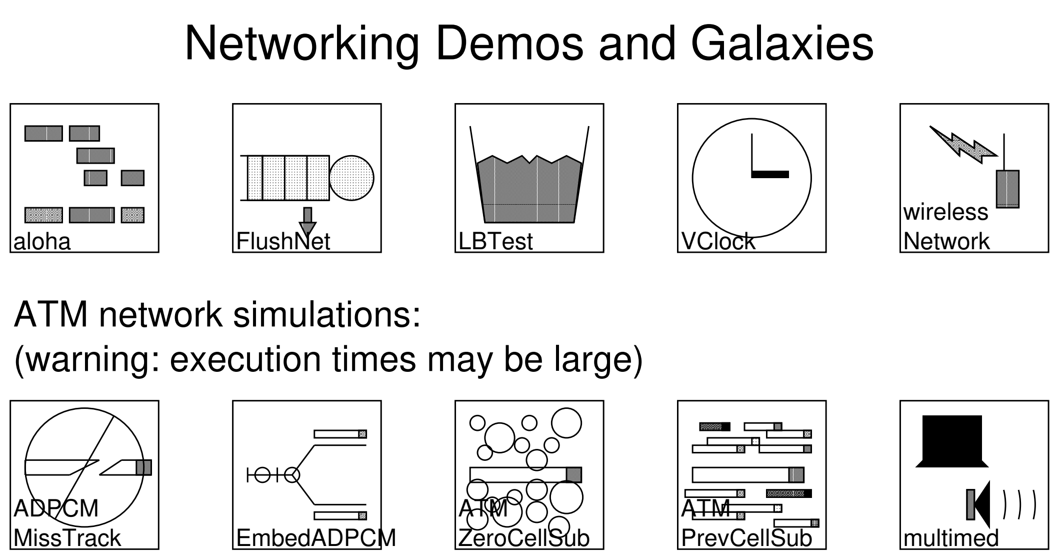

FIGURE 5-18: Networking demos

mode) networks. Some of these demos are quite sophisticated, and the execution times may be large. In addition, some make use of audio playback or video display software, if it is available. The execution times of these demos may be quite large.

`ADPCMMissTrack` Demonstrate the pitfalls of using a simple voice-recovery technique such as zero-sample substitution in an ADPCM packet-switched network.

`EmbedADPCM` Demonstrate shows how embedded-ADPCM with LSB dropping may be used in an ATM network to prevent mistracking between the coder and decoder.

`ATMZeroCellSub` Demonstrate implements a simple voice-cell recovery technique. Cells which have been dropped during transmission are replaced with all 0 bits.

`ATMPrevCellSub` Implement a slightly better voice-cell recovery technique. Cells which have been dropped during transmission are replaced with the cell which arrived at the receiver immediately prior to the dropped cell.

`multimed` Feed data from several types of sources into a single network. Within the network, the various data streams are routed from switch to switch. One video stream and one audio stream are decoded and played. Data losses may occur within the network and reduce the final quality of the presented video and audio streams. The video display requires that the Utah Raster Toolkit be installed and available in the user's path.

5.4.4 Miscellaneous demos

The palette in figure 5-19 shows miscellaneous demos. The first two of these model continuous-time random processes, although only discrete-time samples of these processes can be displayed.

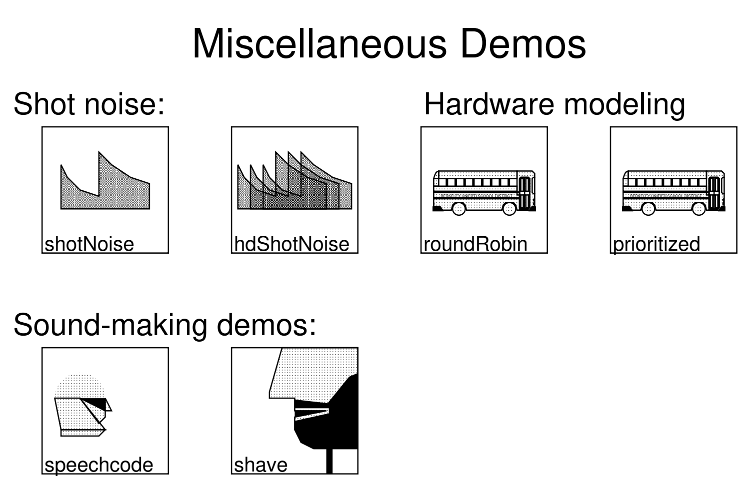

FIGURE 5-19: Miscellaneous demos.

shotNoise Generate a continuous-time shot-noise process and display regularly spaced samples of it. The shot noise is generated by feeding a Poisson process into a **Filter** star.

hdShotNoise Generate a high-density shot noise process and verify its approximately Gaussian distribution by displaying a histogram.

The following demos illustrate the use of the DE domain for high-level modeling of protocols for sharing hardware resources.

roundRobin Simulate shared memory with round-robin arbitration at a high level.

prioritized Simulate of a shared memory with prioritized arbitration at a high level.

The following demos make sounds.

speechcode Perform speech compression with a combination of silence detection, adaptive quantization, and adaptive estimation. After speech samples are read from a file, they are encoded, packetized, depacketized, decoded, and played on the workstation speaker.

shave Demonstrate the Synchronize star to generate a beeping sound with a real-time rhythm.

5.4.5 Wormhole demos

The palette in figure 5-20 shows some simple demonstrations of multiple domain simulations. Each of these combines SDF with DE. The demos are:

worm Show how easy it is to use the SDF stars to perform computation on DE particles. A Poisson process where particles have value 0.0 is sent into an SDF wormhole, where Gaussian noise is added to the samples.

distortion Show the effects on real-time signals of a highly simplified packet-switched network. Packets can arrive out of order, and they can also arrive too late to be useful. In this simplified system, a sinusoid is generated in the SDF domain, launched into the communication network implemented in the DE domain,

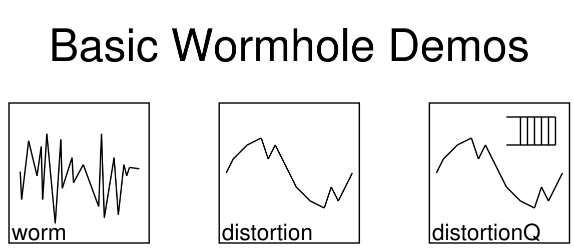

FIGURE 5-20: Wormhole demos

and compared to the output of the communication network. Plots are given in the time and frequency domain of the sinusoid before and after the network.

distortionQ Similar to the distortion demo. The only difference is in the **reorderQ** wormhole, which introduces queueing.