Chapter 12. DE Domain

Authors: *Soonhoi Ha*
Edward A. Lee
Thomas M. Parks

Other Contributors: *Brian L. Evans*

12.1 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.

We assume in this chapter that the reader is thoroughly familiar with the DE model of computation. Refer to the *User's Manual*. Moreover, we assume the reader is familiar with chapter 2, "Writing Stars for Simulation". In this chapter, we give the additional information required to write stars for the DE domain.

12.2 Programming Stars in the DE Domain

A DE star can be viewed as an event-processor; it receives events from the outside, processes them, and generates output events after some latency. In a DE star, the management of the time stamps of the particles (events) is as important as the input/output mapping of particle values.

Generating output values and time stamps are separable tasks. For greatest modularity, therefore, we dedicate some DE stars, so-called *delay star*s, to time management. Examples of such stars are `Delay` and `Server`. These stars, when fired, produce output events that typically have larger time stamps than the input events. They usually do not manipulate the *value* of the particles in any interesting way. The other stars, so-called *functional star*s, avoid time-management, usually by generating output events with the same time stamp as the input events. They, however, *do* manipulate the *value* of the particles.

Delay stars should not be confused with the delay marker on an arc connecting two stars (represented in `pigi` by a small green diamond). The latter delay is not implemented as a star. It is a property of the arc. In the DE domain, the delay marker does not introduce a time delay, in the sense of an incremented time stamp. It simply tells the scheduler to ignore the arc while assigning dataflow-based firing priorities to stars. A star whose outputs are all marked with delays will have the lowest firing priority, and so will be fired last among those stars eligible to be fired at the current time.

The scheduler's assignment of firing priority also uses properties of the individual stars: each star type can indicate whether or not it can produce zero-delay outputs. If a star indicates that it does not produce any output events with zero delay, then the scheduler can break the dataflow priority chain at that star. This saves the user from having to add explicit delay markers. A star class can make this indication either globally (it never produces any immediate output event) or on a port-by-port basis (only some of its input ports can produce immediate outputs, perhaps on only a subset of its output ports).

For managing time stamps, the `DEStar` class has two DE-specific members: `arrivalTime` and `completionTime`, summarized in table 12-1.. Before firing a star, a DE scheduler sets the value of the `arrivalTime` member to equal the time stamp of the event triggering the current firing. When the star fires, before returning, it typically sets the value of the `completionTime` member to the value of the time stamp of the latest event produced by the star. The schedulers do not use the `completionTime` member, however, so it can actually be used in any way the star writer wishes. `DEStar` also contains a field `delayType` and a method `setMode` that are used to signal the properties of the star, as described below.

12.2.1 Delay stars

*Delay-star*s manipulate time stamps. Two types of examples of delay stars are *pure delays*, and *servers*. A *pure-delay* star generates an output with the same value as the input sample, but with a time stamp that is greater than that of the input sample. The difference between the input sample time stamp and the output time stamp is a fixed, user-defined value. Consider for example the `Delay` star:

```
defstar {
      name {Delay}
      domain {DE}
      desc { Delays its input by a fixed amount }
      input {
            name {input}
            type {anytype}
      }
      output {
            name {output}
            type {=input}
      }
      defstate {
            name {delay}
            type {float}
            default {"1.0"}
            desc { Amount of time delay. }
      }
      constructor {
            delayType = TRUE;
      }
      go {
            completionTime = arrivalTime + double(delay);
            Particle& pp = input.get();
            output.put(completionTime) = pp;
      }
}
```

Inside the `go` method description, the `completionTime` is calculated by adding the delay to the arrival time of the current event. The last two lines will be explained in more detail below.

Another type of delay star is a *server*. In a *server* star, the input event waits until a simulated resource becomes free to attend to it. An example is the `Server` star:

```
defstar {
      name {Server}
      domain {DE}
      desc {
This star emulates a server. If an input event arrives when it
is not busy, it delays it by the service time (a constant parameter).
If it arrives when it is busy, it delays it by more than the service
time. It must become free, and then serve the input.
      }
      input {
            name {input}
            type {anytype}
      }
      output {
            name {output}
            type {=input}
      }
      defstate {
            name {serviceTime}
            type {float}
            default {"1.0"}
            desc { Service time. }
      }
      constructor {
            delayType = TRUE;
      }
      go {
            // No overlapped execution. set the time.
            if (arrivalTime > completionTime)
                  completionTime = arrivalTime + double(serviceTime);
            else
                  completionTime += double(serviceTime);
            Particle& pp = input.get();
      output.put(completionTime) = pp;
      }
}
```

This star uses the `completionTime` member to store the time at which it becomes free after processing an input. On a given firing, if the `arrivalTime` is later than the `completionTime`, meaning that the input event has arrived when the server is free, then the server delays the input by the `serviceTime` only. Otherwise, the time stamp of the output event is calculated as the `serviceTime` plus the time at which the server becomes free (the `completionTime`).

Both pure delays and servers are delay stars. Hence their constructor sets the `delayType` member, summarized in table 12-1. This information is used by the scheduler.

The technical meaning of the `delayType` flag is this: such a star guarantees that it will never produce any output event with zero delay; all its output events will have timestamps larger than the time of the firing in which they are emitted. Stars that can produce zero-delay events should leave `delayType` set to its default value of `FALSE`.

Actually, stars often cheat a little bit on this rule; as we just saw, the standard `Delay` star sets `delayType` even if the user sets the star's delay parameter to zero. This causes the star to be treated as though it had a positive delay for the purpose of assigning firing priorities, which is normally what is wanted. Both pure delays and servers are delay stars. Hence their constructor sets the `delayType` member, summarized in table 12-1. This information is used by the scheduler, and is particularly important when determining which of several simultaneous events to process first.

12.2.2 Functional Stars

In the DE model of computation, a star is *runnable* (ready for execution), if any input porthole has a new event, and that event has the smallest time stamp of any pending event in the system. When the star fires, it may need to know which input or inputs contain the events that triggered the firing. An input porthole containing a new particle has the `dataNew` flag set by the scheduler. The star can check the `dataNew` flag for each input. A functional star will typically read the value of the new input particles, compute the value of new output particles, and produce new output particles with time stamps identical to those of the new inputs. To see how this is done, consider the `Switch` star:

```
defstar {
      name {Switch}
      domain {DE}
      desc {
Switches input events to one of two outputs, depending on
the last received control input.
      }
      input {
            name {input}
            type {anytype}
      }
      input {
            name {control}
            type {int}
      }
      output {
            name {true}
            type {=input}
      }
      output {
            name {false}
            type {=input}
      }
      constructor {
            control.triggers();
            control.before(input);
      }
      go {
            if (input.dataNew) {
                  completionTime = arrivalTime;
```

```
                    Particle& pp = input.get();
                    int c = int(control%0);
              if(c)
                          true.put(completionTime) = pp;
                    else
                          false.put(completionTime) = pp;
              }
        }
  }
```

The `Switch` star has two input portholes: `input`, and `control`. When an event arrives at the `input` porthole, it routes the event to either the `true` or the `false` output porthole depending on the value of the last received `control` input. In the `go` method, we have to check whether a new `input` event has arrived. If not, then the firing was triggered by a `control` input event, and there is nothing to do. We simply return. If the `input` is new, then its particle is read using `get` method, as summarized in table 12-1. In addition, the most recent value from the control input is read. This value is used to determine which output should receive the data input. The statements in the constructor will be explained below in “Sequencing directives” on page 12-6.

There are three ways to access a particle from an input or output port. First, we may use the `%` operator followed by an integer, which is equivalent to the same operator in the SDF

InDEPort class

method	description
`Particle& operator %`	get a particle from the porthole without resetting dataNew
`void before (GenericPort& p)`	simultaneous inputs here should be processed before those at p
`int dataNew`	flag indicating whether the porthole has new data
`Particle& get ()`	get a particle from the porthole and reset dataNew
`void getSimulEvent ()`	fetch a simultaneous event from the global event queue
`int numSimulEvents ()`	return the number of pending simultaneous events at this input
`void triggers ()`	indicate that the input does not trigger any immediate output events
`void triggers (GenericPort& p)`	indicate that the input triggers an immediate output on port p

OutDEPort class

	description
`Particle& operator %`	get the most recent particle from the porthole
`Particle& put (double time)`	get a new writable particle with the given time stamp
`void sendData ()`	flush output porthole data to the global event queue (called by put)

TABLE 12-1: A summary of the members and methods of the `InDEPort` and `OutDEPort` classes that are used by star writers.

domain. For example, `control%0` returns the most recent particle from the `control` porthole. The second method, `get`, is specific to `InDEPort`. It resets the `dataNew` member of the port as well as returning the most recent particle from an input port. In the above example, we are not using the `dataNew` flag for the `control` input, so there is no need to reset it. However, we are using it for the `input` porthole, so it must be reset. If you need to reset the `dataNew` member of a input port after reading the newly arrived event (the more common case) you should use the `get` method instead of `%0` operator. Alternatively, you can reset the `dataNew` flag explicitly using a statement like:

```
input.dataNew = FALSE;
```

The `put` method is also specific to `OutDEPort`. It sets the `timeStamp` member of the port to the value given by its argument, and returns a reference to the most recent particle from an output port. Consider the line in the above example:

```
true.put(completionTime) = pp;
```

This says that we copy the particle `pp` to the output port with `timeStamp = completionTime`. We can send more than one output event to the same port by calling the `put` method repeatedly. A new particle is returned each time.

12.2.3 Sequencing directives

A special effort has been made in the DE domain to handle simultaneous events in a rational way. 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 12-1, 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.

The cryptic statements in the constructor in the above example reveal these triggering relationships to the scheduler. Consider for example the following problem. In the `Switch` star above suppose that on a given firing, an `input` with time stamp τ is processed, and the particle is sent to the `true` output. Suppose that the very next time the star fires, a `control`

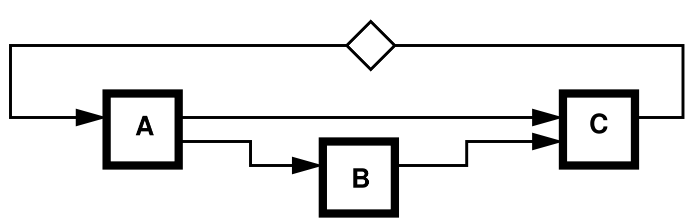

FIGURE 12-1: When DE stars are enabled by simultaneous events, the choice of which to fire is determined by priorities based on a topological sort. Thus if B and C both have events with identical time stamps, B will take priority over C. The delay on the path from C to A serves to break the topological sort.

input with time stamp τ arrives with value `FALSE`. Probably, the previous output should have gone to the `false` porthole. Consider the constructor statement:

```
control.before(input);
```

This tells the scheduler that if a situation arises where two simultaneous events might appear at the `control` and `input` portholes, then the one at the `control` porthole should appear first. This is implemented by giving the stars “upstream” from the `control` porthole higher firing priorities than those “upstream” from the `input` porthole. Thus, if for some reason the simultaneous events are processed in two separate firings (always a possibility), then the `control` event is sure to be processed first. A chain of `before` directives can assign relative priorities to a whole set of inputs.

The other statement in the constructor:

```
control.triggers();
```

has somewhat different objectives. It tells the scheduler that a `control` input does not trigger outputs on any porthole. If an input event causes an output event with the same time stamp, then the input event is said to have “triggered” the output event. In the above example, the `control` event does not trigger any immediate output event, but an `input` event does. By default, an input triggers all outputs, so it is not necessary to add the directive

```
input.triggers(output);
```

Providing `triggers` directives informs the scheduler that certain paths through the graph do not have zero delay, allowing it to ignore those paths in making its topological sort. The `triggers` directive is essentially a selective version of the `delayType` flag: setting `delayType` means the star contains **no** zero-delay paths, whereas providing `triggers` information tells the scheduler that only certain porthole-to-porthole paths through the star have zero delay. By default, the scheduler assumes that all paths through the star have zero delay.

In some stars, an input event conditionally triggers an output. In principle, if there is any chance of triggering an output, we set the triggering relation between the input and the output. The triggering relation informs the scheduler that there **may be** a delay-free path from the input to the output. It is important, therefore, that the star writer not miss any triggering relation when `triggers` directives are provided.

If an input triggers some, but not all outputs, then the constructor for the star should contain several `triggers` directives, one for each output that is triggered by that input. If an input triggers all outputs, then no directive is necessary for it.

If `delayType` is set to `TRUE`, it is not necessary to write any triggers directives; a delay star by definition never triggers zero-delay output events.

12.2.4 Simultaneous events

An input port may have a sequence of simultaneous events (events with identical time stamps) pending. Normally, the star will be fired repeatedly until all such events have been consumed. Optionally, a DE star may process simultaneous events during a single firing. The `getSimulEvent` method can be used as in the following example, taken from an up-down counter star:

```
go {
...   while (countUp.dataNew) {
            count++;
            countUp.getSimulEvent();
```

```
    }
    ... }
```

Here, `countUp` is an input porthole. The `getSimulEvent` method examines the global event queue to see if any more events are available for the porthole with the current timestamp. If so, it fetches the next one and sets the `dataNew` flag to `TRUE`; if none remain, it sets the `dataNew` flag to `FALSE`. (In this example, the actual values of the input events are uninteresting, but the star could use `get()` within the loop if it did need the event values.)

Sometimes, a star simply needs to know how many simultaneous events are pending on a given porthole. Without fetching any event, we can get the number of simultaneous events by calling the `numSimulEvents` method. This returns the number of simultaneous events still waiting in the global event queue; the one already in the porthole isn't counted.

If the star has multiple input ports, the programmer should carefully consider the desired behavior of simultaneous inputs on different ports, and choose the order of processing of events accordingly. For example, it might be appropriate to absorb all the events available for a control porthole before examining any events for a data porthole.

If a star will always absorb all simultaneous events for all its input portholes, it can use phase-based firing mode to improve performance. See section 12.3.

12.2.5 Non-deterministic loops

The handling of simultaneous events is based on assigning priorities to portholes, tracing the connectivity of a schematic, and using the relationships established by the `before` and `triggers` relationships. When we assign these priorities, we start from the input ports of sink stars, and rely primarily on a topological sort. Delay-free loops, which would prevent the topological sort for terminating, are detected and ruled out. But, another kind of loop, called a *non-deterministic loop*, can cause unexpected results. A non-deterministic loop is one in which the priorities cannot be assigned uniquely; there is more than one solution. Such a loop has at least one `before` relation. If a programmer can guarantee that there is no possibility of simultaneous events on such a loop, then system may be simulated in a predictable manner. Otherwise, the arbitrary decisions in the scheduler will affect the firing order.

If a non-deterministic loop contains exactly one `before` relation, the scheduler assigns priorities in a well-defined way, but unfortunately, in a way that is hidden from the user. For a non-deterministic loop with more than one `before` relation, the assignment of the priorities is a non-deterministic procedure. Therefore, the scheduler emits a warning message. The warning message suggests that the programmer put a delay element on an arc (usually a feedback arc) to break the non-deterministic loop. As mentioned before, the delay element has a totally different meaning from that in the SDF domain. In the SDF domain, a delay implies an initial token on the arc, implying a one-sample delay. In the DE domain, however, a delay element simply breaks a triggering chain. Therefore, the source port of the arc is assigned the lowest priority.

12.2.6 Source stars

The DE stars discussed so far fire in response to input events. In order to build signal generators, or source stars, or stars with outputs but no inputs, we need another class of DE star, called a *self-scheduling star*. A self-scheduling star fools the scheduler by generating its own input events. These feedback events trigger the star firings. An event generator is a special case of a delay star, in that its role is mainly to control the time spacing of source events. The values of the source events can be determined by a functional block attached to the output of the event generator (e.g. `Const`, `Ramp`, etc).

A self-scheduling star is derived from class `DERepeatStar`, which in turn is derived from class `DEStar`. The `DERepeatStar` class has two special methods to facilitate the self-scheduling function: `refireAtTime` and `canGetFired`. These are summarized in table 12-2. The `Poisson` star illustrates these:

```
defstar {
      name {Poisson}
      domain {DE}
      derivedfrom { RepeatStar }
      desc {
Generates events according to a Poisson process.
The first event comes out at time zero.
      }
      output {
            name {output}
            type {float}
      }
      defstate {
            name {meanTime}
            type {float}
            default {"1.0"}
            desc { The mean inter-arrival time. }
      }
      defstate {
            name {magnitude}
            type {float}
            default {"1.0"}
            desc { The value of outputs generated. }
      }
      hinclude { <NegExp.h> }
      ccinclude { <ACG.h> }
      protected {
            NegativeExpntl *random;
      }
      // declare the static random-number generator in the .cc file
```

DERepeatStar class

method	description
`int canGetFired ( )`	return 1 if the star is enabled for firing
`void refireAtTime (double t)`	schedule the star to fire again at time t
`void begin ()`	schedule the star to fire at time zero

TABLE 12-2: A summary of the methods of the `DERepeatStar` class used when writing a source star. Source stars are derived from this.

```
        code {
              extern ACG* gen;
        }
        constructor {
              random = NULL;
        }
        destructor {
              if(random) delete random;
        }
        begin {
              if(random) delete random;
              random = new NegativeExpntl(double(meanTime),gen);
              DERepeatStar::begin ();
        }
        go {
              // Generate an output event
              // (Recall that the first event comes out at time 0).
              output.put(completionTime) << double(magnitude);

              // and schedule the next firing
              refireAtTime(completionTime);

              // Generate an exponential random variable.
              double p = (*random)();

              // Turn it into an exponential, and add to completionTime
              completionTime += p;
        }
  }
```

The `Poisson` star generates a Poisson process. The inter-arrival time of events is exponentially distributed with parameter *meanTime*. Refer to "Using Random Numbers" on page 3-17 for information about the random number generation. The method `refireAtTime` launches an event onto a feedback arc that is invisible to the users. The feedback event triggers the self-scheduling star some time later.

Note that the feedback event for the next execution is generated in the current execution. To initiate this process, an event is placed on the feedback arc by the `DERepeatStar::begin` method, before the scheduler runs.

The `DERepeatStar` class can also be used for other purposes besides event generation. For example, a sampler star might be written to fire itself at regular intervals using the `refireAtTime` method.

Another strangely named method, `canGetFired` is seldom used in the star definitions. The method checks for the existence of a new feedback event, and returns `TRUE` if it is there, or `FALSE` otherwise.

The internal feedback arc consists of an input and an output porthole that are automatically created and connected together, with a delay marker added to prevent the scheduler from complaining about a delay-free loop. (This effectively assumes that refire requests will always be for times greater than the current time.)

Sometimes the programmer of a star derived from `DERepeatStar` needs to be explicitly aware of these portholes. In particular, they should be taken into account when considering whether a star is delay-type. Setting `delayType` in a `DERepeatStar` derivative asserts that not only do none of the star's visible input portholes trigger output events with zero delay, but refire events do not either. Frequently this is a false statement. It's usually safer to write `triggers` directives that indicate that specific input portholes cannot trigger zero-delay outputs. (Since the feedback portholes have a delay marker, it is never necessary to mention the feedback output porthole in `triggers` directives, even for an input porthole that gives rise to `refireAtTime` requests --- the scheduler is uninterested in zero-delay paths to the feedback output.)

The event passed across the feedback arc is an ordinary FLOAT particle, normally having value zero. Sometimes it can be useful to store extra information in the feedback event. Beginning in Ptolemy 0.7, the `refireAtTime` method accepts an optional second parameter that gives the numeric value to place in the feedback event. Fetching the value currently requires direct access to the feedback input port, for example

```
if (feedbackIn->dataNew) {
      double feedbackValue = double(feedbackIn->get());
      ...
```

A future version of `DERepeatStar` might provide some syntactic sugar to hide the details of this operation.

In Ptolemy versions prior to 0.7, `DERepeatStar` did not place a delay marker on the feedback arc, but instead used a hack involving special porthole priorities. This hack did not behave very well if the star also had ordinary input portholes. To work around it, writers of derived star types would sometimes set `delayType` or provide `triggers` directives. When updating such stars to 0.7, these statements should be examined critically --- they will often be found to be unnecessary, and perhaps even wrong.

12.3 Phase-Based Firing Mode

The ordering of simultaneous events is the most challenging task of the DE scheduler. In general, simultaneous events are caused by insufficient time resolution, particularly when the time unit is integral. In our case, simultaneous events are primarily caused by functional stars that produce output events with the same time stamp as the input events. Since the time stamp is a double-precision floating-point number, we have very high time resolution.

As explained earlier, the DE scheduler fetches at most one event for each input porthole for each firing of a DE star. In the body of the star code, the programmer can consume the simultaneous events onto a certain input porthole by calling the `getSimulEvent` method for the porthole. This mode of operation is called *simple* mode, which is the default mode of operation.

Suppose we program a new DE star, called `Counter`. The `Counter` star has one `clock` input and one `demand` input. A `clock` event will increase the counter value by one, and the `demand` input will send the counter value to the output. If there are multiple simultaneous `clock` inputs and a simultaneous `demand` input, we should count all the `clock` inputs before consuming the `demand` input and producing an output. Thus, the programmer should call the `getSimulEvent` method for the `clock` input. However, the `getSimulEvent` method is expensive when there are many simultaneous events, since it gets only one simultaneous event at a time. This runtime overhead is reduced in the *phase-based firing* mode.

In the *phase-based firing* mode, or simply the *phase* mode, before executing a star, the scheduler fetches all simultaneous events for the star. The fetched events are stored in the internal queue of each input porthole. The internal queue of inputs is created only if the star operates in phase mode. In phase mode, when a DE star fires, it consumes all simultaneous events currently available. It constructs a *phase*. Afterwards, other simultaneous events for the same star may be generated by a network of functional stars. Then, the star may be fired again with another set of simultaneous events, which forms another phase. We can set the operation mode of a star *phase* by calling method `setMode(PHASE)` in the constructor, as summarized in table 12-1 on page 12-5. The following example is written in the simple mode.

```
go {
      ...
      while (input.dataNew) {
            temp += int(input.get());
            input.getSimulEvent();
      }
      ...
}
```

If the star is re-written using the phase mode, it will look like:

```
constructor {
      setMode(PHASE);
}
go {
      ...
      while (input.dataNew) {
            temp += int(input.get());
      }
      ...
}
```

or,

```
go {
      ...
      for (int i = input.numSimulEvents(); i > 0; i--) {
            temp += int(input.get());
      }
      ...
}
```

The `get` method in phase mode fetches events from the internal queue one at a time. After consuming all events from the queue (now the queue is empty), it resets the `dataNew` flag. If a star in phase mode does not access all simultaneous input events in a particular firing, the unaccessed events are discarded.

The method, `numSimulEvent`, returns the current queue size in phase mode. Recall that in simple mode, the method returns the number of simultaneous events in the global event queue, which is one less than the actual number of simultaneous events. This difference of one between two modes is necessary for coding efficiency.

There is still inherent non-determinism in the handling of simultaneous events in the

DE domain. For example, suppose that the `Switch` star has more than one simultaneous control event. Which one is really the last one? Since the input is routed to either the `true` or `false` output depending on the last value of the `control` event, the decision is quite critical. We leave the responsibility of resolving such inherent non-determinism to the user.

12.4 Programming Examples

This section presents different examples of programming in the discrete-event domain. There are no pre-defined stars that work with matrices in the discrete-event domain. We will give several examples of DE stars that work with matrices.

12.4.1 Identity Matrix Star

This section develops a star in the DE domain that will create an identity matrix. Instead of creating a source star which must schedule itself, we will create a star that fires whenever it receives an new input value. For example, a clock or some other source can be attached to the star to set its firing pattern.

```
defstar {
      name { Identity_M }
      domain { DE }
      desc { Output a floating-point identity matrix.}
      author { Brian L. Evans }
      input {
            name { input }
            type { anytype }
      }
      output {
            name { output }
            type { FLOAT_MATRIX_ENV }
      }
      defstate {
            name { rowsCols }
            type { int }
            default { 2 }
            desc {
Number of rows and columns of the output square matrix. }
      }
      ccinclude { "Matrix.h" }
      go {
            // Functional Star: pass timestamp without change
            completionTime = arrivalTime;
            // For messages, you must pass dynamically allocated data
            FloatMatrix& result =
                  *(new FloatMatrix(int(rowsCols),int(rowsCols)));
            // Set the contents of the matrix to an identity matrix
            result.identity();
            // Send the matrix result to the output port
            output.put(completionTime) << result;
            }
      }
```

This is a functional star because the time stamp on the input particle is not altered. The output is a matrix message. The matrix is a square matrix. In order for the matrix to remain defined after the go method finishes, the matrix `result` cannot be allocated from local memory. Instead, it must be allocated from global dynamic memory via the `new` operator. In the syntax for the new operator, the `int` cast in `int(rowsCols)` extracts the value from `rowsCols` which is an instance of the `State` data structure. The dynamic memory allocated for the matrix will be automatically deleted by the `Message` class. Then, the matrix is reset to be an identity matrix. Finally, the matrix is sent to the output port with the same time stamp as that of the input data. Note that the syntax to output data in the discrete-event domain differs from the syntax of the synchronous dataflow domain due to the time stamp. In the SDF domain, the output code would be

```
output%0 << result
```

12.4.2 Matrix Transpose

In the next example, we will compute the matrix transpose.

```
defstar {
      name { Transpose_M }
      domain { DE }
      desc { Transpose a floating-point matrix.}
      author { Brian L. Evans }
      input {
            name { input }
            type { FLOAT_MATRIX_ENV }
      }
      output {
            name { output }
            type { FLOAT_MATRIX_ENV }
      }
      ccinclude { "Matrix.h" }
      go {
            // Functional Star: pass timestamp without change
            completionTime = arrivalTime;
            // Extract the matrix on the input port
            Envelope Xpkt;
            input.get().getMessage(Xpkt);
            const FloatMatrix& Xmatrix =
                  *(const FloatMatrix *)Xpkt.myData();
            // Create a copy of the input matrix
            FloatMatrix& xtrans = *(new FloatMatrix(Xmatrix));
            // Transpose the matrix
            xtrans.transpose();
            // Send the matrix result to the output port
            output.put(completionTime) << xtrans;
      }
}
```

The key difference between creating an identity matrix and taking a matrix transpose in the DE domain is the conversion of the input data to a matrix. The input data comes in the form of an envelope which is essentially an instance of a class embedded in a message particle. To extract the contents of the message, we first extract the message from the input envelope. Then, we extract the data field from the message and cast it to be a `FloatMatrix`. Just as in the previous example, we need to allocate dynamic memory to hold the value of the matrix to be output. In this case, we do not have to code the transpose operation since it is already built into the matrix classes.