Chapter 7. DDF Domain

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

Other Contributors: *Joseph T. Buck*

7.1 Introduction

The dynamic dataflow (DDF) domain in Ptolemy is a superset of the synchronous dataflow (SDF) and Boolean dataflow (BDF) domains. In the SDF domain, a star consumes and produces a fixed number of particles per invocation (or "firing"). This static information (the number of particles produced or consumed for each star) makes possible compile-time scheduling. In the BDF domain, some actors with data-dependent production or consumption are allowed. The BDF schedulers attempt to construct a compile-time schedule; however, they may fail to do so and fall back on a DDF scheduler. In the DDF domain, the schedulers make no attempt to construct a compile-time schedule. For this reason, there are few constraints on the production and consumption behavior of stars in this domain.

In DDF, a run-time scheduler detects which stars are runnable and fires them one by one until no star is runnable (the system is deadlocked), or until a specified stopping condition has been reached. A star is runnable if it has enough data on its inputs to satisfy its requirements. Thus, the only constraint on DDF stars is that they must specify on each firing how much data they require on each input to be fired again later.

In practice, stars in the DDF domain are written in a slightly simpler way. They are either SDF stars, in which case the number of particles required at each input is a constant, or they are dynamic, in which case they always alert the scheduler before finishing a firing that to be refired they expect some specific number of particles on one particular input. The input that a star is waiting for data on is called the *waitPort*.

Since the DDF domain is a superset of the SDF domain, all SDF stars can be used in the DDF domain. Similarly for BDF stars. Besides the SDF stars, the DDF domain has some DDF-specific stars that will be described in this chapter. The DDF-specific stars overcome the main modeling limitation of the SDF domain in that they can model dynamic constructs such as *conditionals*, *data-dependent iteration*, and *recursion*. All of these except recursion are also supported by the BDF domain. It is even possible, in principle, to dynamically modify a DDF graph as it executes (the implementation of recursion does exactly this). The lower run-time efficiency of dynamic scheduling is the cost that we have to pay for the enhanced modeling power.

Run-time scheduling is expensive. In figure 7-1 we have plotted the execution time of a simple example (setup and run, not including the pigi "compile" or the wrapup). The example contains 17 stars, all simple, all homogeneous synchronous dataflow (producing or consuming a single sample at each port). The tests were run on a Sparc 10 using the `ptrim` executable (on August 26, 1995). The default schedulers in the SDF and DDF domains were used. Note that both schedulers took approximately 13ms at startup, and then exhibited a close to linear increase in execution time. For the SDF scheduler, the slope is approximately 650 μs per iteration, while for the DDF scheduler, it is approximately 1,370 μs per iteration. With 17 stars, this comes to about 38 μs per firing for SDF and 81 μs per firing for DDF. For multirate systems, both of these schedulers will perform poorly compared to the loop scheduler in SDF. Note that for this simple system, DDF is more than twice as expensive as SDF. For systems that require DDF, Ptolemy allows us to regain much of this efficiency by grouping SDF stars in a wormhole that contains an SDF domain. For critical systems that are executed for many iterations, this can provide for considerably faster execution.

There are some subtleties, however, in DDF scheduling. Due to these subtleties, there have been three DDF schedulers implemented, all accessible by setting appropriate target parameters. In the next section, we explain these schedulers.

7.2 The DDF Schedulers

In Ptolemy, a scheduler determines the order of execution of blocks. This would seem to be a simple task in the DDF domain, since there is nothing to do at setup time, and at run time, the scheduler only needs to determine which blocks are runnable and then fire those blocks. Experience dictates, however, that this simple-minded policy is not adequate. In particular, it may use more memory than is required (it may even require an unbounded amount of memory when a bounded amount of memory would suffice). It may also be difficult for a user to specify for how long an execution should proceed.

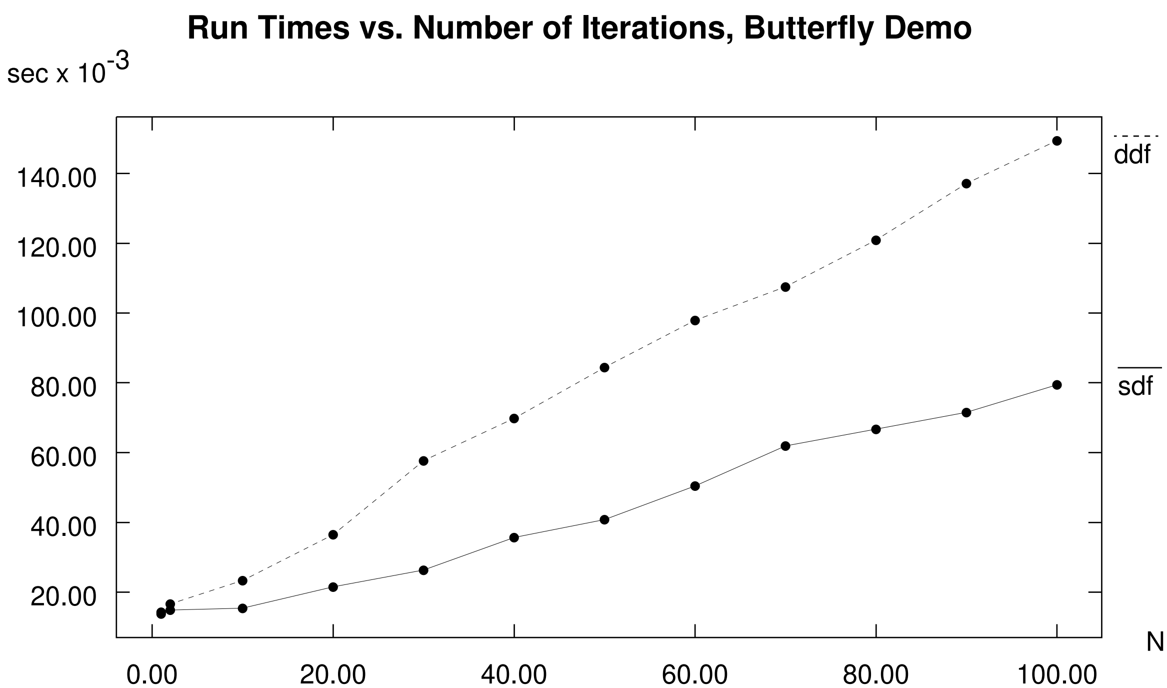

FIGURE 7-1: Time (in milliseconds) vs. number of iterations for the default SDF and default DDF schedulers for a 17 star, single-sample-rate example.

In the SDF domain, an *iteration* is well-defined. It is the minimum number of firings that brings the buffers back to their original state. In SDF, this can be found by a compile-time scheduler by solving the balance equations. In both BDF and DDF, it turns out that it is *undecidable* whether such a sequence of firings exists. This means that no algorithm can answer the question for all graphs of a given size in finite time. This explains, in part, why the BDF domain may fail to construct a compile-time schedule and fall back on the DDF schedulers.

We have three simple and obvious criteria that a DDF scheduler should satisfy:

a. The scheduler should be able to execute a graph forever if it is possible to execute a graph forever. In particular, it should not stop prematurely if there are runnable stars.

b. The scheduler should be able to execute a graph forever in bounded memory if it is possible to execute the graph forever in bounded memory.

c. The scheduler should execute the graph in a sequence of well-defined and determinate iterations so that the user can control the length of an execution by specifying the number of iterations to execute.

Somewhat surprisingly, it turns out to be extremely difficult to satisfy all three criteria at once. The first few versions of the DDF scheduler (up to and including release 0.5.2) did not satisfy (b) or (c). The older scheduler is still available (set the *useFastScheduler* target parameter to *YES*), but its use is not recommended. Its behavior is somewhat unpredictable and sometimes counterintuitive. For example, told to run a graph for one iteration, it may in fact run it forever. Nonetheless, it is still available because it is significantly faster than the newer schedulers. We have not found a way (yet) to combine its efficient and clever algorithm with the criteria above.

The reason that these criteria are hard to satisfy is fundamental. We have already pointed out that it is undecidable whether a sequence of firings exists that will return the graph to its original state. This fact can be used to show that it is undecidable whether a graph can be executed in bounded memory. Thus, no finite analysis can always guarantee (b). The trick is that the DDF scheduler in fact has infinite time to run an infinite execution, so, remarkably, it is still possible to guarantee condition (b). The new DDF schedulers do this.

Regarding condition (a), it is also undecidable whether a graph can be executed forever. This question is equivalent to the *halting problem*, and the DDF model of computation is sufficiently rich that the halting problem cannot always be solved in finite time. Again, we are fortunate that the scheduler has infinite time to carry out an infinite execution. This is really what we mean by dynamic scheduling!

Condition (c) is more subtle and centers around the desire for *determinate* execution. What we mean by this, intuitively, is that a user should be able to tell immediately what stars will fire in one iteration, knowing the state of the graph. In other words, which stars fire should not depend on arbitrary decisions made by the scheduler, like the order in which it examines the stars.

To illustrate that this is a major issue, suppose we naively define an iteration to consist of “firing all enabled stars at most once.” Consider the simple example in figure 7-2. Star A is enabled, so we can fire it. Suppose this makes star B enabled. Should it be fired in the same iteration? Will the order in which we fire enabled stars or determine whether stars are enabled impact the outcome?

We have implemented two policies in DDF. These are explained below.

7.2.1 The default scheduler

The default scheduler, realized in the class `DDFSimpleSched`, first scans all stars and determines which are enabled. In a second pass, it then fires the enabled stars. Thus, the order in which the stars fire has no impact on which ones fire in a given iteration.

Unfortunately, as stated, this simple policy still does not work. Suppose that star A in figure 7-2 produces two particles each time it fires, and actor B consumes 1. Then our policy will be to fire actor A in the first iteration and both A and B in all subsequent iterations. This violates criterion (b), because it will not execute in bounded memory. More importantly, it is counterintuitive. Thus, the `DDFSimpleSched` class implements a more elaborate algorithm.

One iteration, by default, consists of firing all enabled and non-deferrable stars once. If no stars fire, then one deferrable star is carefully chosen to be fired. A *deferrable star* is one with any output arc (except a self-loop) that has enough data to satisfy the destination actor. In other words providing more data on that output arc will not help the downstream actor become enabled; it either already has enough data, or it is waiting for data on another arc. If a deferrable star is fired, it will be the one that has the smallest maximum output buffer sizes. The algorithm is formally given in figure 7-3.

This default iteration is defined to fire actors at most once. Sometimes, a user needs several such *basic iterations* to be treated as a single iteration. For example, a user may wish for a *user iteration* to include one firing of an `XMgraph` star, so that each iteration results in

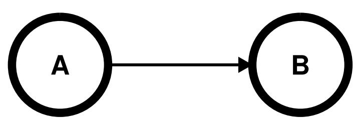

FIGURE 7-2: A simple example used to illustrate the notion of an iteration.

```
At the start of the iteration compute {
      E = enabled actors
      D = deferrable actors
}

One default iteration consists of {
      if (E-D != 0) fire stars in (E-D)
      else if (D != 0) fire the minimax star in D
      else deadlocked.
}
The minimax star is the one with the smallest
maximum number of tokens on its output paths.
```

FIGURE 7-3: The algorithm implementing one basic iteration in the `DDFSimpleSched` class.

one point plotted. The basic iteration may not include one such firing. Another more critical example is a wormhole that contains a DDF system but will be embedded in an SDF system. In this case, it is necessary to ensure that one user iteration consists of enough firings to produce the expected number of output particles.

This larger notion of an iteration can be specified using the target *pragma* mechanism to identify particular stars that must fire some specific number of times (greater than or equal to one) in each user iteration. To use this, make sure the domain is DDF and the target is `DDF-default`. Then in pigi, place the mouse over the icon of the star in question, and issue the *edit-pragmas* command ("a"). One pragma (the one understood by this target) will appear; it is called *firingsPerIteration*. Set it to the desired value. This will then define what makes up an iteration.

7.2.2 The clustering scheduler

If you set the target parameter *restructure* to YES, you will get a scheduler that clusters SDF actors when possible and invokes the SDF scheduler on them. The scheduler is implemented in the class `DDFClustSched`. **WARNING**: As of this writing, this scheduler will not work with wormholes, and will issue a warning. Nonetheless, it is an interesting scheduler for two reasons, the first of which is its clustering behavior. The second is that it uses a different definition of a basic iteration. In this definition, a basic iteration (loosely) consists of as many firings as possible subject to the constraint that no actor fires more than once and that deferrable actors are avoided if possible. The complete algorithm is given in figure 7-4. Use of this scheduler is not advised at this time, however. For one thing, the implementation of clustering adds enough overhead that this scheduler is invariably slower than the default scheduler.

```
The following sets are updated every time a star fires:
      E = enabled actors
      D = deferrable actors
      S = source actors
      F = actors that have fired once already in this iteration

One default iteration consists of:
      while (E-D-F != 0) {
            fire actors in (E-D-F)
      }
      if (F == 0) {
            // All enabled actors are deferrable.
            // Try the non-sources first.
            if (E-S != 0) {
                  fire (E-S);
            } else {
                  fire (S);
            }
      }
      if (F == 0) deadlock
```

FIGURE 7-4: A basic iteration of the DDFClustSched scheduler.

7.2.3 The fast scheduler

In case the new definition of an iteration is inconvenient for legacy systems, we preserve an older and faster scheduler that is not guaranteed to satisfy criteria (b) and (c) above. The basic operation of the fast scheduler is to repeatedly scan the list of stars in the domain and execute the runnable stars until no more stars are runnable, with certain constraints imposed on the execution of sources. For the purpose of determining whether a star is runnable, the stars are divided into three groups. The first group of the stars have input ports that consume a fixed number of particles. All SDF stars, except those with no input ports, are included in this group. For this group, the scheduler simply checks all inputs to determine whether the star is runnable.

The second group consists of the DDF-specific stars where the number of particles required on the input ports is unspecified. An example is the `EndCase` star (a multi-input version of the BDF `Select` star). The `EndCase` star has one control input and one multiport input for data. The control input value specifies which data input port requires a particle. Stars in this group must specify at run time how many input particles they require on each input port. Stars specify a port with a call to a method called *waitPort* and the number of particles needed with a call to *waitNum*. To determine whether a star is runnable, the scheduler checks whether a specified input port has the specified number of particles.

For example, in the `EndCase` star, the *waitPort* points to the *control* input port at the beginning. If the *control* input has enough data (one particle), the star is fired. When it is fired, it checks the value of the particle in the *control* port, and changes the *waitPort* pointer to the input port on which it needs the next particle. The star will be fired again when it has enough data on the input port pointed by *waitPort*. This time, it collects the input particle and sends it to the output port. See Figure 7-5.

The third group of stars comprises sources. Sources are always runnable. Source stars introduce a significant complication into the DDF domain. In particular, since they are always runnable, it is difficult to ensure that they are not invoked too often. This scheduler has a reasonable but not foolproof policy for dealing with this. Recall that the DDF domain is a superset of the SDF domain. The definition of one iteration for this scheduler tries to obtain the same results as the SDF scheduler when only SDF stars are used. In the SDF domain, the number of firings of each source star, relative to other stars, is determined by solving the balance equations. However, in the DDF domain, the balance equations do not apply in the same

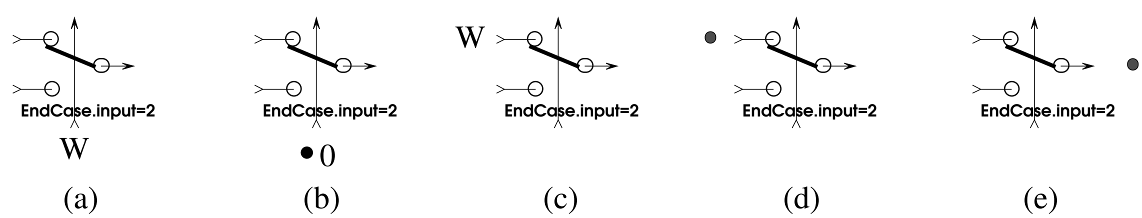

FIGURE 7-5: (a) The EndCase star waits on the *control* port. (b) The star fires when data arrives on the *control* port (the value of the data is 0). (c) Now the star waits for input to arrive on input port 0. (d) The star fires again when data arrives on input port 0. (e) The data that arrived on input port 0 is transmitted by the output port of the EndCase star.

form[1]. The technique we use instead is *lazy-evaluation*.

Lazy evaluation

At the beginning of each iteration of a DDF application, we fire all source stars exactly once, and temporarily declare them "not runnable." We also fire all stars that have enough *initial* tokens on their inputs. After that, the scheduler starts scanning the list of stars in the domain. If a star has some particles on some input arcs, but is not runnable yet, then the star initiates the (lazy) evaluation of those stars that are connected to the input ports requiring more data. This evaluation is "lazy" because it occurs only if the data it produces are actually needed. The lazy-evaluation technique ensures that the relative number of firings of source stars is the same under the DDF scheduler as it would be under the SDF scheduler.

We can now define what is meant by *one iteration* in DDF. An iteration consists of one firing of each source star, followed by as many lazy-evaluation passes as possible, until the system deadlocks. One way to view this (loosely) is that enough stars are fired to consume all of the data produced in the first pass, where the source stars were each fired once. This may involve repeatedly firing some of the source stars. However, a lazy-evaluation is only initiated if a star in need of inputs already has at least one input with enough tokens to fire. Because of this, in some circumstances, the firings that make up an iteration may not be exactly what is expected. In particular, when there is more than one sink star in the system, and the sink stars fire at different rates, the ones firing at higher rates may not be fired as many times as expected. It is also possible for one iteration to never terminate.

When a DDF wormhole is invoked, it will execute one iteration of the DDF system contained in it. This is a serious problem in many applications, since the user may need more control over what constitutes one firing of the wormhole.

7.3 Inconsistency in DDF

So far, we have assumed an error-free program. In the SDF domain, compile-time analysis detects errors due to inconsistent rates of production and consumption of tokens because the balance equations cannot be solved. In DDF, however, such inconsistencies are harder to detect. Our strategy is to detect them at run time, an approach that has two disadvantages. First, it is costly, as will be explained shortly. Second, it is not easy to isolate the sources of errors.

We call a dataflow graph *consistent* if on each arc, in the long run, the same number of particles are consumed as produced [Lee91a]. One source of inconsistency is the sample-rate mismatch that is common to the SDF domain. The DDF domain has more subtle error sources, however, due to the dynamic behavior of DDF stars. In an inconsistent graph, an arc may queue an unbounded number of tokens in the long run. To prevent this, we examine the number of tokens on each arc to detect whether the number is greater than a certain limit (the default is 1024). If we find an arc with too many tokens, we consider it an error and halt the execution. We can modify the limit by setting the target parameter named *maxBufferSize*. The two new schedulers will interpret a negative number here to be infinite capacity. An inconsistent system will run until your computer runs out of memory.

1. Note that the BDF domain adapts the balance equations to the dynamic dataflow case.

The value of the *maxBufferSize* parameter will be the maximum allowed buffer size. Since the source of inconsistency is not unique, isolating the source of the error is usually not possible. We can just point out which arc has a large number of tokens. Of course, if the limit is set too high, some errors will take very long to detect. Note however that there exist perfectly correct DDF systems (which are consistent) that nonetheless cannot execute in bounded memory. It is for this reason that the new schedulers support infinite capacity.

7.4 The default-DDF target

The DDF domain only has one target. The parameters of the target are:

maxBufferSize (`INT`) Default = `1024`
The capacity of an arc (in particles). This is used for the run-time detection of inconsistencies, as explained above. If any arc exceeds this capacity, an error is flagged and the simulation halts. A negative number is interpreted as infinite capacity (unless *useFastScheduler* is `YES`). The value of this parameter does not specify how much memory is allocated for the buffers, since the memory is allocated dynamically.

schedulePeriod (`FLOAT`) Default = `0.0`
This defines the amount of time taken by one iteration (defined above) of the DDF schedule. This is used only for interface with timed domains, such as DE. Note that if you want the count given in the debug panel of the run control panel to indicate the number of iterations, you should set this parameter to one.

runUntilDeadlock (`INT`) Default = `NO`
Unless *useFastScheduler* is set, this modifies the definition of a single iteration to invoke all stars as many times as possible, until the system halts. It is risky to use this because the system may not halt. But in wormholes it is sometimes useful.

restructure (`INT`) Default = `NO`
This specifies that the experimental scheduler `DDFClustSched` should be used. This scheduler attempts to form SDF clusters for more efficient execution. Its use is not advised at this time, however, since it does not work properly with wormholes and is slower than the default scheduler.

useFastScheduler (`INT`) Default = `NO`
This specifies that the older and faster DDF scheduler (from version 0.5.2) should be used. It is difficult, however, to control the length of a run with this scheduler.

numOverlapped (`INT`) Default = `1`
For the fast scheduler only, this gives the number of iteration cycles that can be overlapped for execution. When a DDF system starts up, it normally begins by firing each source star once, as explained above. It then goes into a lazy evaluation mode.

Setting this parameter to an integer *N* larger than one allows the scheduler to begin with *N* firings of the source stars instead of just one. This can make execution more efficient, because stars downstream from the sources will be able to fire multiple times in each pass through the graph. The default value of this parameter is 1.

logFile (`STRING`) Default =
The default is the empty string. If non-empty, this gives the name of a file to be used for recording scheduler information.

7.5 An overview of DDF stars

The "open-palette" command in pigi ("O") will open a checkbox window that you can use to open the standard palettes in all of the installed domains. For the DDF domain, the star library is small enough that it is contained entirely in one palette, shown in figure 7-6.

`Case` (Three icons.) Route an input particle to one of the outputs depending on the control particle. The control particle should be between zero and $N-1$, inclusive, where N is the number of outputs.

`EndCase` (Three icons.) Depending on the control particle, consume a particle from one of the data inputs and send it to the output. The control particle should have value between zero and $N-1$, inclusive, where N is the number of inputs.

`DownCounter` Given an integer input with value N, produce a sequence of output integers with values $(N-1)$, $(N-2)$, ... 1, 0.

`LastOfN` Given a control input with integer value N, consume N particles from the data input and produce only the last of these at the output.

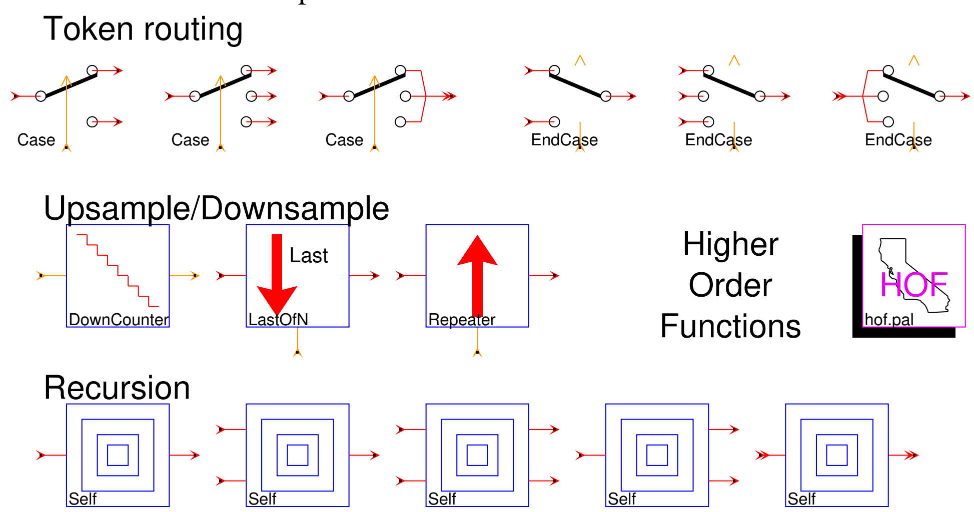

FIGURE 7-6: The palette of stars for the DDF domain.

`Repeater` Given a control input with integer value *N*, and a single input data particle, produce *N* copies of the data particle on the output.

The Higher Order Functions icon leads to the HOF palette that contains HOF stars that can be used within DDF.

`Self` (Five icons.) This is a first exploration of recursion and higher-order functions in dataflow. It is still experimental, so do not expect it to be either efficient or bug-free.

The star "represents" the galaxy given by the parameter *recurGal*, which must be above it in the hierarchy. That is, when the `Self` star fires, it actually invokes the galaxy that it represents. Since that galaxy is above the `Self` star in the hierarchy, it contains the `Self` star somewhere within it. Thus, this star implements recursion. Since the `Self` star takes an argument (*recurGal*) that specifies the function to invoke, it is itself a higher-order function.

The instance of the *recurGal* galaxy is not created until it is actually needed, so the number of instances (the depth of the recursion) does not need to be known *a priori*. If the parameter *reinitialize* is `NO` or `FALSE`, then the instance of the galaxy is created the first time it fires and reused on subsequent firings. If *reinitialize* is `YES` or `TRUE`, then the galaxy is created on every firing and destroyed after the firing. Inputs are sent to the instance of the galaxy and outputs are retrieved from it. The inputs of the named galaxy must be named "input#?" and the outputs must be named "output#?", where "?" is replaced with an integer starting with zero. This allows the inputs and outputs of this star to be matched unambiguously with the inputs and outputs of the referenced galaxy.

7.6 An overview of DDF demos

The demos with icons shown in figure 7-7 illustrate dynamic dataflow principles.

`eratosthenes` The sieve of Eratosthenes is a recursive algorithm for computing prime numbers. This demo illustrates the implementation of recursion in the DDF domain. This is a concept demonstration only.

`errorDemo` An example of an inconsistent DDF system. An inconsistent DDF program is one where the long term average number of particles produced on an arc is not the same as the average long term number of particles consumed. This error is detected by bounding the buffer sizes and detecting overflow.

`ifThenElse` This demo illustrates the use of an SDF wormhole to implement a dynamically scheduled construct using the DDF domain. An if-then-else is such a dynamically scheduled construct. The top level schematic represents an SDF system, while the inside schematic represents a DDF system (implementing an if-then-else).

`fibonnacci` Generate the Fibonnacci sequence using a rather inefficient recursive algorithm that is nonetheless a good example of how to realize recursion.

`loop` This demo illustrates data-dependent iteration. Input integers are repeatedly multiplied by 0.5 until the product is less than 0.5. Turn on animation to see the iteration.

`picture` Construct a two-dimensional random walk using a hierarchy of nested wormholes. The outermost SDF domain has a wormhole called "drawline" which internally uses the DDF domain. That wormhole, in turn, has a wormhole called "display" which internally uses the SDF domain.

`repeat` This simple demo shows the effect of running a DDF scheduler on an SDF system. The *firingsPerIteration* pragma is used to control the meaning of an iteration.

`repeater` This is a simple illustration of the `Repeater` star, used in an SDF wormhole (DDF inside SDF).

`router` This is a simple illustration of the `EndCase` star.

`SDFinDDF` This rather trivial demo illustrates the use of a DDF wormhole whose inside domain is SDF. The top-level system (in the DDF domain) has an if-then-else overall structure, implemented of a matching pair of `Case` and `EndCase` stars. The inside system

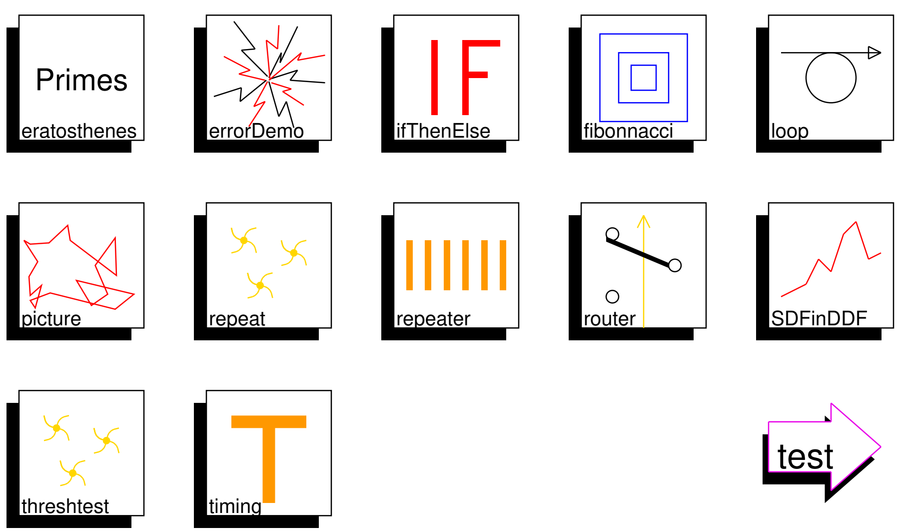

FIGURE 7-7: The DDF demos.

(in the SDF domain) multiplies the data value by a ramp.

`threshtest` This demo shows that Karp & Miller style thresholds are supported in DDF. The `Thresh` star is a dummy that implements a settable threshold.

`timing` This demo illustrates the use of the DDF domain to implement asynchronous signal processing systems. In this case, the system performs baud-rate timing recovery using an approximate minimum mean-square-error (MMSE) technique.

7.7 Mixing DDF with other domains

The mixture of the DDF domain with other domains requires a conversion between different computational models. In particular, some domains associate a *time stamp* with each particle, and other domains do not. Thus, a common function at the `EventHorizon` is the addition of time stamps, the stripping off of time stamps, interpolation between time stamps, or removal of redundant repetitions of identical particles. In this section, DDF-specific features on the domain interface will be discussed.

A galaxy or universe implemented using DDF may have a wormhole which contains a subsystem implemented in another domain. The DDF wormhole looks exactly like a DDF star from the outside. However, there are certain technical restrictions. In particular, it cannot have dynamic input portholes, meaning the number of particles consumed by the wormhole inputs is a compile-time constant. The wormhole is therefore fired when all input ports have new particles. When it is fired, it consumes the input data, invokes the scheduler of the inner domain, and retrieves the output particles. Thus, in all respects except one, the DDF wormhole behaves like an SDF wormhole (see "Wormholes" on page 12-4 for more information). The one exception is that the DDF wormhole need not consistently produce outputs.

When a DDF system is embedded within another domain, you may need to explicitly control what constitutes a firing of the subsystem. Specifically, by setting the *firingsPerIteration* pragma of a star in the DDF subsystem, you control how many firings of that star are required to complete an iteration. Zero means "don't care."

Note that some work has been done with a CGDDF target which recognizes and implements certain commonly used programming constructs [Sha92]. See "CG Domain" on page 13-1 for more information.