Chapter 3 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)," Technical Memorandum UCB/ERL M04/17, University of California, Berkeley, CA USA 94720, June 24, 2004.

3

SDF Domain

Author: *Steve Neuendorffer*
Contributor: *Brian Vogel*

3.1 Purpose of the Domain

The synchronous dataflow (SDF) domain is useful for modeling simple dataflow systems without complicated flow of control, such as signal processing systems. Under the SDF domain, the execution order of actors is statically determined prior to execution. This results in execution with minimal overhead, as well as bounded memory usage and a guarantee that deadlock will never occur. This domain is specialized, and may not always be suitable. Applications that require dynamic scheduling could use the process networks (PN) domain instead, for example.

3.2 Using SDF

There are four main issues that must be addressed when using the SDF domain:

- Deadlock
- Consistency of data rates
- The value of the iterations parameter
- The granularity of execution

This section will present a short description of these issues. For a more complete description, see section 3.3.

3.2.1 Deadlock

Consider the SDF model shown in figure 3.1. This actor has a feedback loop from the output of the AddSubtract actor back to its own input. Attempting to run the model results in the exception shown at the right in the figure. The director is unable to schedule the model because the input of the AddSubtract actor depends on data from its own output. In general, feedback loops can result in such condi-

tions.

The fix for such deadlock conditions is to use the SampleDelay actor, shown highlighted in figure 3.2. This actor injects into the feedback loop an initial token, the value of which is given by the *initialOutputs* parameter of the actor. In the figure, this parameter has the value {0}. This is an array with a single token, an integer with value 0. A double delay with initial values 0 and 1 can be specified using a two element array, such as {0, 1}.

It is important to note that it is occasionally necessary to add a delay that is not in a feedback loop to match the delay of an input with the delay around a feedback loop. It can sometimes be tricky to see exactly where such delays should be placed without fully considering the flow of the initial tokens described above.

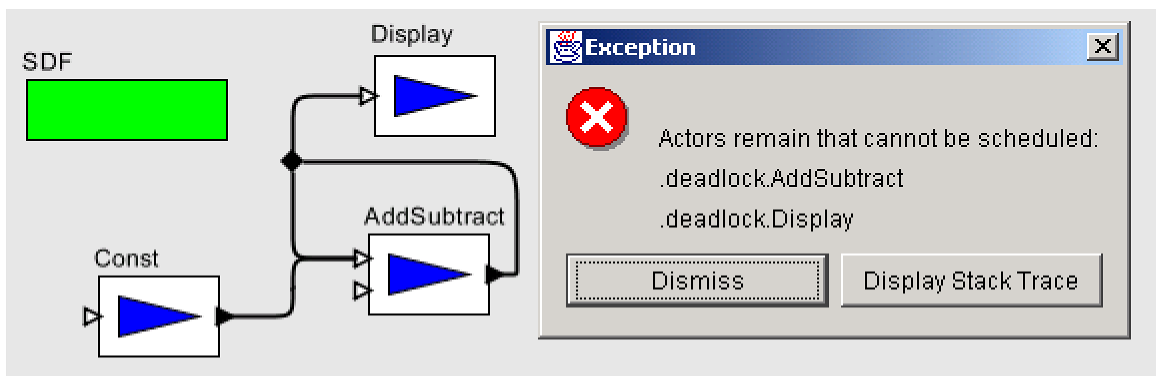

FIGURE 3.1. An SDF model that deadlocks.

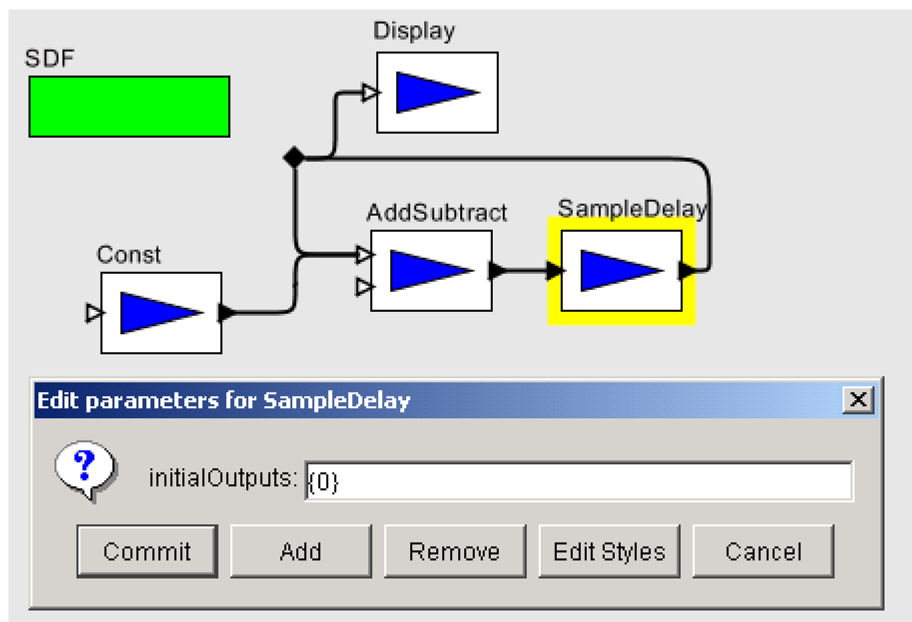

FIGURE 3.2. The model of figure 3.1 corrected with an instance of SampleDelay in the feedback loop.

3.2.2 Consistency of data rates

Consider the SDF model shown in figure 3.3. The model is attempting to plot a sinewave and its downsampled counterpart. However, there is an error because the number of tokens on each channel of the input port of the plotter can never be made the same. The DownSample actor declares that it consumes 2 tokens using the *tokenConsumptionRate* parameter of its input port. Its output port similarly declares that it produces only one token, so there will only be half as many tokens being plotted from the DownSample actor as from the Sinewave.

The fixed model is shown in figure 3.4, which uses two separate plotters. When the model is executed, the plotter on the bottom will fire twice as often as the plotter on the top, since it must consume twice as many tokens. Notice that the problem appears because one of the actors (in this case, the DownSample actor) produces or consumes more than one token on one of its ports. One easy way to ensure rate consistency is to use actors that only produce and consume one token at a time. This special case is known as *homogeneous* SDF. Note that actors like the SequencePlotter which do not specify rate parameters are assumed to be homogeneous. For more specific information about the rate parame-

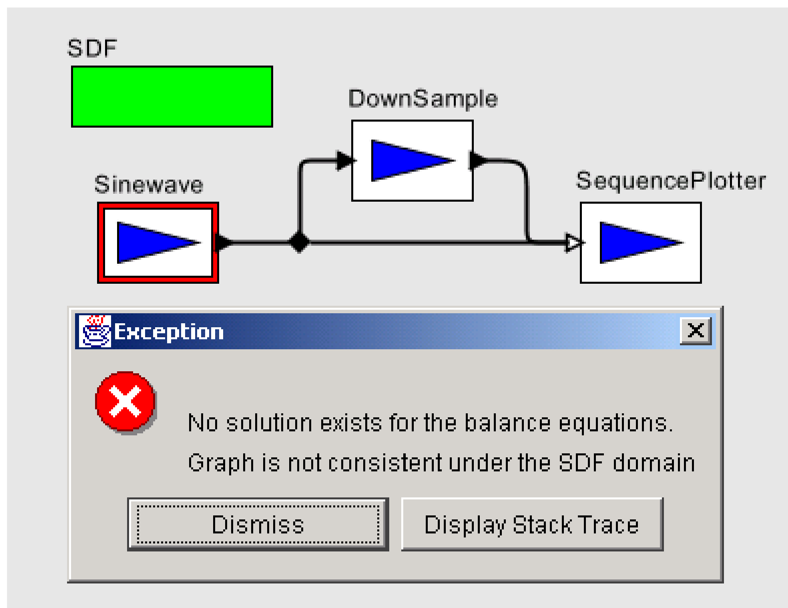

FIGURE 3.3. An SDF model with inconsistent rates.

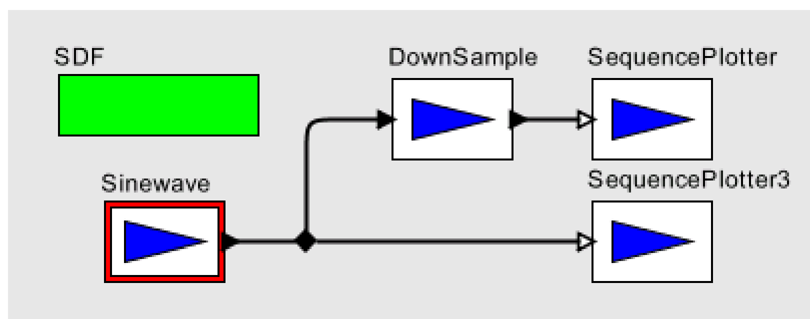

FIGURE 3.4. Figure 3.3 modified to have consistent rates.

ters and how they are used for scheduling, see section 3.3.1.

3.2.3 How many iterations?

Another issue when using the SDF domain concerns the value of the *iterations* parameter of the SDF director. In homogeneous models one token is usually produced for every iteration. However, when token rates other than one are used, more than one interesting output value may be created for each iteration. For example, consider figure 3.5 which contains a model that plots the Fast Fourier Transform of the input signal. The important thing to realize about this model is that the FFT actor declares that it consumes 256 tokens from its input port and produces 256 tokens from its output port, corresponding to an order 8 FFT. This means that only one iteration is necessary to produce all 256 values of the FFT.

Contrast this with the model in figure 3.6. This model plots the individual values of the signal. Here 256 iterations are necessary to see the entire input signal, since only one output value is plotted in each iteration.

3.2.4 Granularity

The granularity of execution of an SDF model is determined by solving a set of equations determined by declared data rates of actors, and the connections between actors. As mentioned in the previous section, this schedule may involve a small or large number of firings of each actor, depending on the relative data rates of the actors. Generally, the smallest possible valid schedule, corresponding to the smallest granularity of execution, is the most interesting. However, there are some instances when

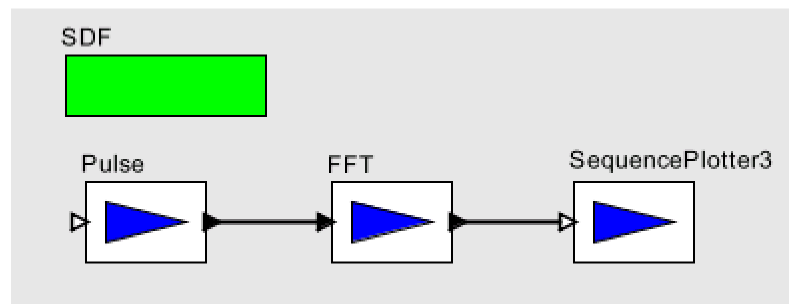

FIGURE 3.5. A model that plots the Fast Fourier Transform of a signal. Only one iteration must be executed to plot all 256 values of the FFT, since the FFT actor produces and consumes 256 tokens each firing.

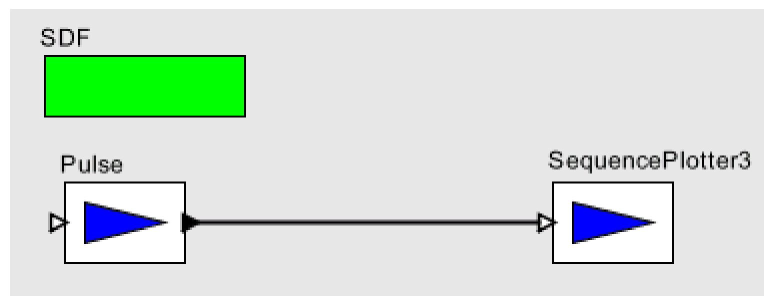

FIGURE 3.6. A model that plots the values of a signal. 256 iterations must be executed to plot the entire signal.

this is not the case. In such cases the *vectorizationFactor* parameter of the SDF Directory can be used to scale up the granularity of the schedule. A *vectorizationFactor* of 2 implies that each actor is fired twice as many times as in the normal schedule.

One example when this might be useful is when modeling block data processing. For instance, we might want to build a model of a signal processing system that filters blocks of 40 samples at a time using an FIR filter. Such an actor could be written in Java, or it could be built as a hierarchical SDF model, using a single sample FIR filter, as shown in Figure 3.7. The *vectorizationFactor* parameter of the Director is set to 40. Here, each firing of the SDF model corresponds to 40 firings of the single sample FIR filter.

Another useful time to increase the level of granularity is to allow vectorized execution of actors. Some actors override the iterate() method to allow optimized execution of several consecutive firings. Increasing the granularity of an SDF model can provide more opportunities for the SDF Director to perform this optimization, especially in models that do not have fine-grained feedback.

3.3 Properties of the SDF domain

SDF is an untimed model of computation. All actors under SDF consume input tokens, perform their computation and produce outputs in one atomic operation. If an SDF model is embedded within a timed model, then the SDF model will behave as a zero-delay actor.

In addition, SDF is a statically scheduled domain. The firing of a composite actor corresponds to a single iteration of the contained(3.3.1) model. An SDF iteration consists of one execution of the pre-calculated SDF schedule. The schedule is calculated so that the number of tokens on each relation is the same at the end of an iteration as at the beginning. Thus, an infinite number of iterations can be executed, without deadlock or infinite accumulation of tokens on each relation.

Execution in SDF is extremely efficient because of the scheduled execution. However, in order to execute so efficiently, some extra information must be given to the scheduler. Most importantly, the data rates on each port must be declared prior to execution. The data rate represents the number of tokens produced or consumed on a port during every firing[1]. In addition, explicit data delays must be added to feedback loops to prevent deadlock. At the beginning of execution, and any time these data rates change, the schedule must be recomputed. If this happens often, then the advantages of scheduled execution can quickly be lost.

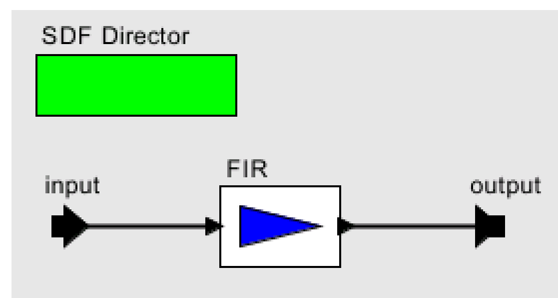

FIGURE 3.7. A model that implements a block FIR filter. The vectorizationFactor parameter of the director is set to the size of the block.

1. This is known as *multirate* SDF, where arbitrary rates are allowed. Not to be confused with *homogeneous* SDF, where the data rates are fixed to be one.

3.3.1 Scheduling

The first step in constructing the schedule is to solve the *balance equations* [84]. These equations determine the number of times each actor will fire during an iteration. For example, consider the model in figure 3.8. This model implies the following system of equations, where *ProductionRate* and *ConsumptionRate* are declared properties of each port, and *Firings* is a property of each actor that will be solved for:

$$\mathit{Firings}(\mathrm{A}) \times \mathit{ProductionRate}(\mathrm{A1}) = \mathit{Firings}(\mathrm{B}) \times \mathit{ConsumptionRate}(\mathrm{B1})$$

$$\mathit{Firings}(\mathrm{A}) \times \mathit{ProductionRate}(\mathrm{A2}) = \mathit{Firings}(\mathrm{C}) \times \mathit{ConsumptionRate}(\mathrm{C1})$$

$$\mathit{Firings}(\mathrm{C}) \times \mathit{ProductionRate}(\mathrm{C2}) = \mathit{Firings}(\mathrm{B}) \times \mathit{ConsumptionRate}(\mathrm{B2})$$

These equations express constraints that the number of tokens created on a relation during an iteration is equal to the number of tokens consumed. These equations usually have an infinite number of linearly dependent solutions, and the least positive integer solution for *Firings* is chosen as the *firing vector*, or the repetitions vector.

The second step in constructing an SDF schedule is dataflow analysis. Dataflow analysis orders the firing of actors, based on the relations between them. Since each relation represents the flow of data, the actor producing data must fire before the consuming actor. Converting these data dependencies to a sequential list of properly scheduled actors is equivalent to topologically sorting the SDF graph, if the graph is acyclic[1]. Dataflow graphs with cycles cause somewhat of a problem, since such graphs cannot be topologically sorted. In order to determine which actor of the loop to fire first, a *data delay* must be explicitly inserted somewhere in the cycle. This delay is represented by an initial token created by one of the output ports in the cycle during initialization of the model. The presence of the delay allows the scheduler to break the dependency cycle and determine which actor in the cycle to fire first. In Ptolemy II, the initial token (or tokens) can be sent from any port, as long as the port declares an *tokenInitProduction* property. However, because this is such a common operation in SDF, the Delay actor (see section 3.5) is provided that can be inserted in a feedback loop to break the cycle. Cyclic graphs not properly annotated with delays cannot be executed under SDF. An example of a cyclic graph properly annotated with a delay is shown in figure 3.9.

In some cases, a non-zero solution to the balance equations does not exist. Such models are said to

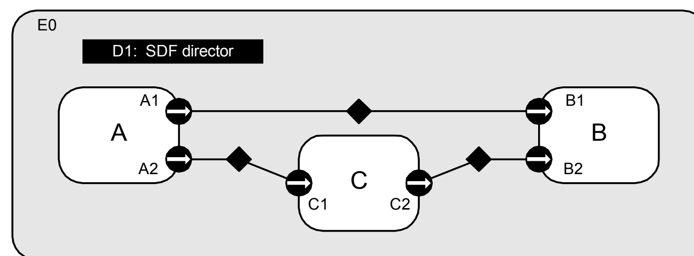

FIGURE 3.8. An example SDF model.

1. Note that the topological sort does not correspond to a unique total ordering over the actors. Furthermore, especially in multirate models it may be possible to interleave the firings of actors that fire more than once. This can result in many possible schedules that represent different performance trade-offs. We anticipate that future schedulers will be implemented to take advantage of these trade-offs. For more information about these trade-offs, see [47].

be *inconsistent*, and cannot be executed under SDF. Inconsistent graphs inevitably result in either deadlock or unbounded memory usage for any schedule. As such, inconsistent graphs are usually bugs in the design of a model. However, inconsistent graphs can still be executed using the PN domain, if the behavior is truly necessary. Examples of consistent and inconsistent graphs are shown in figure 3.10.

3.3.2 Hierarchical Scheduling

So far, we have assumed that the SDF graph is not hierarchical. The simplest way to schedule a hierarchical SDF model is flatten the model to remove the hierarchy, and then schedule the model as

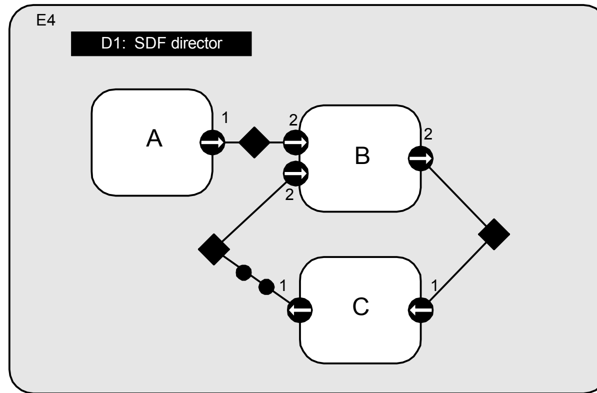

FIGURE 3.9. A consistent cyclic graph, properly annotated with delays. A one token delay is represented by a black circle. Actor C is responsible for setting the *tokenInitProduction* parameter on its output port, and creating the two tokens during initialization. This graph can be executed using the schedule A, A, B, C, C.

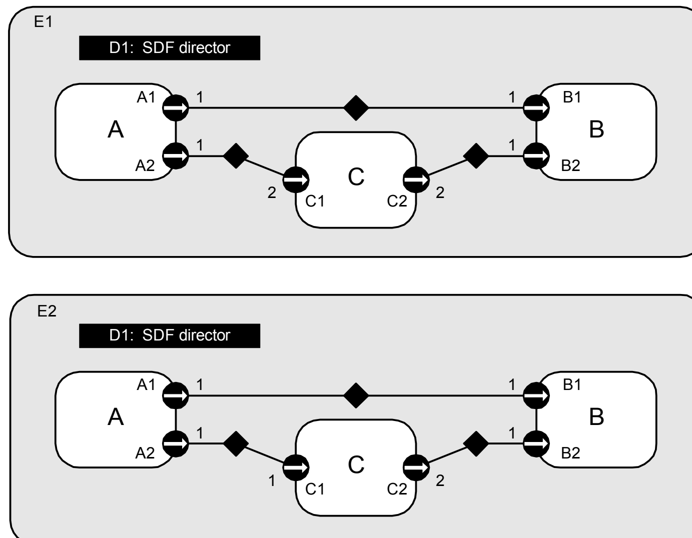

FIGURE 3.10. Two models, with each port annotated with the appropriate rate properties. The model on the top is consistent, and can be executed using the schedule A, A, C, B, B. The model on the bottom is inconsistent because tokens will accumulate between ports C2 and B2.

usual. This technique allows the most efficient schedule to be constructed for a model, and avoids certain composability problems when creating hierarchical models. In Ptolemy II, a model created using a transparent composite actor to define the hierarchy is scheduled in exactly this way.

Ptolemy II also supports a stronger version of hierarchy, in the form of opaque composite actors. In this case, the hierarchical actor appears to be no different from the outside than an atomic actor with no hierarchy. The SDF domain does not have any information about the contained model, other than the rate parameters that may be specified on the ports of the composite actor. The SDF domain is designed so that it automatically sets the rates of external ports when the schedule is computed. Most other domains are designed (conveniently enough) so that their models are compatible with default rate properties assumed by the SDF domain. For a complete description of these defaults, see the description of the SDFScheduler class in section 3.4.2.

3.3.3 Hierarchically Heterogeneous Models

An SDF model can generally be embedded in any other domain. However, SDF models are unlike most other hierarchical models in that they often require multiple inputs to be present. When building one SDF model inside another SDF model, this is ensured by the containing SDF model because of the way the data rate parameters are set as described in the previous section. For most other domains, the SDF director will check how many tokens are available on its input ports and will refuse firing (by returning false in prefire()) until enough data is present for an entire iteration to complete.

3.4 Software Architecture

The SDF kernel package implements the SDF model of computation. The structure of the classes in this package is shown in figure 3.11.

3.4.1 SDF Director

The SDFDirector class extends the StaticSchedulingDirector class. When an SDF director is created, it is automatically associated with an instance of the default scheduler class, SDFScheduler. This scheduler is intended to be relatively fast, but not designed to optimize for any particular performance goal. The SDF director does not currently restrict the schedulers that may be used with it. For more information about SDF schedulers, see section 3.4.2.

The director has a parameter, *iterations*, which determines a limit on the number of times the director wishes to be fired[1]. After the director has been fired the given number of times, it will always return false in its postfire() method, indicating that it does not wish to be fired again. The *iterations* parameter must contain a non-negative integer value. The default value is an IntToken with value 0, indicating that there is no preset limit for the number of times the director will fire. Users will likely specify a non-zero value in the director of the toplevel composite actor as the number of toplevel iterations of the model.

The SDF director also has a *vectorizationFactor* parameter that can be used to request vectorized execution of a model. This parameter increases the granularity of the executed schedule so that the director fires each actor *vectorizationFactor* times more than would be normal. The *vectorizationFactor* parameter must contain a positive integer value. The default value is an IntToken with value one, indicating that no vectorization should be done. Changing this parameter changes the meaning of an

1. This parameter acts similarly to the Time-to-Stop parameter in Ptolemy Classic.

embedded SDF model and may cause deadlock in a model that uses it. On the other hand, increasing the *vectorizationFactor* may increase the efficiency of a model, both by reducing the number of times the SDF model needs to be executed, and by allowing the SDF model to combine multiple firings of contained actors using the iterate() method.

The newReceiver() method in SDF directors is overloaded to return instances of the SDFReceiver class. This receiver contains optimized method for reading and writing blocks of tokens. For more information about SDF receivers, see section 3.4.3.

3.4.2 SDF Scheduler

The basic SDFScheduler derives directly from the Scheduler class. This scheduler provides unlooped, sequential schedules suitable for use on a single processor. No attempt is made to optimize

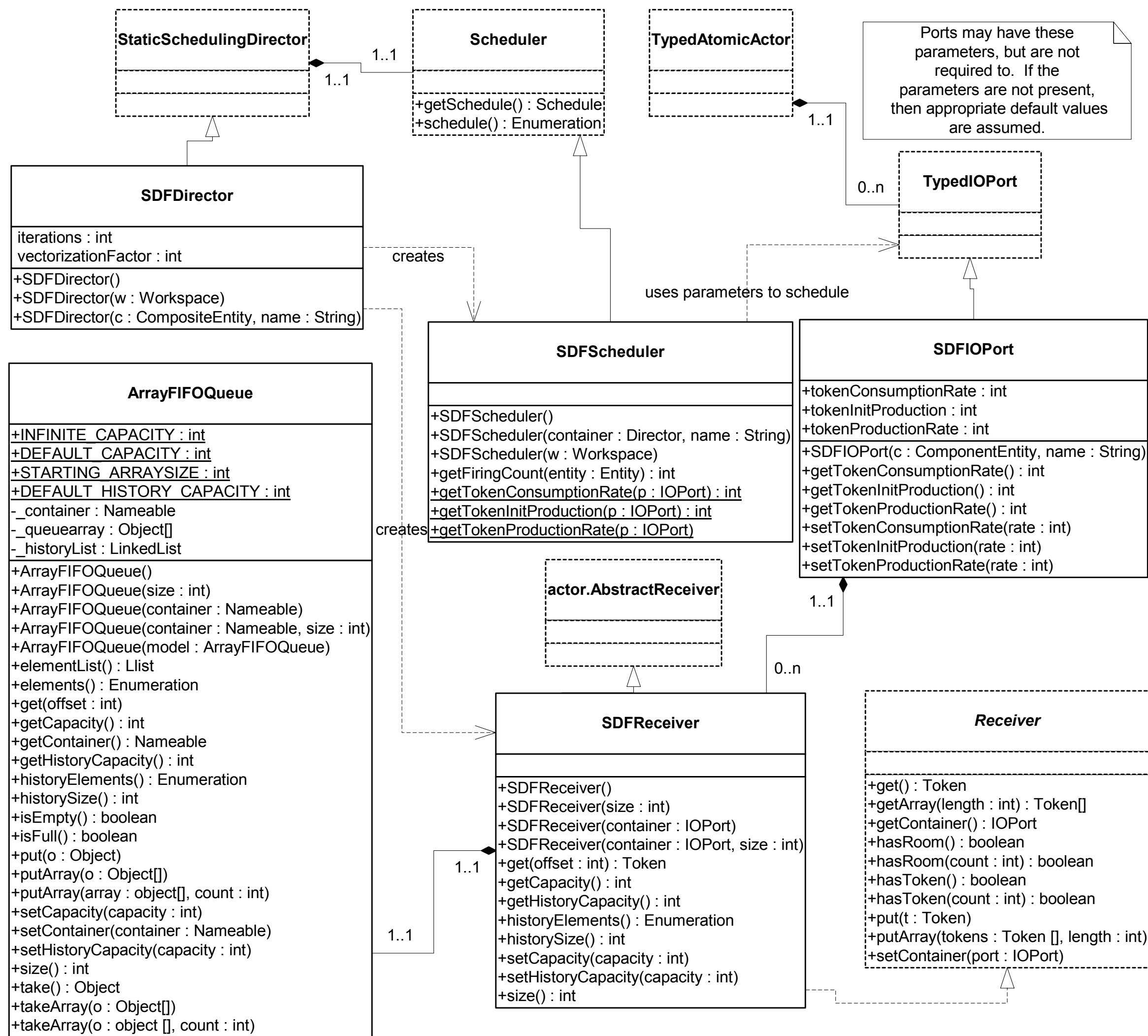

FIGURE 3.11. The static structure of the SDF kernel classes.

the schedule by minimizing data buffer sizes, minimizing the size of the schedule, or detecting parallelism to allow execution on multiple processors. We anticipate that more elaborate schedulers capable of these optimizations will be added in the future.

The scheduling algorithm is based on the simple multirate algorithm in [84]. Currently, only single processor schedules are supported. The multirate scheduling algorithm relies on the actors in the system to declare the data rates of each port. The data rates of ports are specified using three parameters on each port named *tokenConsumptionRate*, *tokenProductionRate*, and *tokenInitProduction*. The production parameters are valid only for output ports, while the consumption parameter is valid only for input ports. If a parameter exists that is not valid for a given port, then the value of the parameter must be zero, or the scheduler will throw an exception. If a valid parameter is not specified when the scheduler runs, then default values of the parameters will be assumed, however the parameters are not then created[1].

After scheduling, the SDF scheduler will set the rate parameters on any external ports of the composite actor. This allows a containing actor, which may represent an SDF model, to properly schedule the contained model, as long as the contained model is scheduled first. To ensure this, the SDF director forces the creation of the schedule after initializing all the actors in the model. The SDF scheduler also sets attributes on each relation that give the maximum buffer size of the relation. This can be useful feedback for analyzing deadlocks, or for visualization.This mechanism is illustrated in the sequence diagram in figure 3.12.

SDF graphs should generally be connected. If an SDF graph is not connected, then there is some concurrency between the disconnected parts that is not captured by the SDF rate parameters. In such cases, another model of computation (such as process networks) should be used to explicitly specify the concurrency. As such, the SDF scheduler by default disallows disconnected graphs, and will throw an exception if you attempt to schedule such a graph. However, sometimes it is useful to avoid introducing another model of computation, or to allow dynamic modifications to an executing model. By setting the *allowDisconnectedGraphs* parameter of the SDF director to true, the scheduler will assume a default notion of concurrency between different parts of a model. Each disconnected 'island' will be scheduled independently, and an overall schedule created that includes one execution of the schedule for each island.

Multiports. Notice that it is impossible to set a rate parameter on individual channels of a port. This is intentional, and all the channels of an actor are assumed to have the same rate. For example, when the AddSubtract actor fires under SDF, it will consume exactly one token from each channel of its input *plus* port, consume one token from each channel of its *minus* port, and produce one token the single channel of its *output* port. Notice that although the domain-polymorphic adder is written to be more general than this (it will consume *up to* one token on each channel of the input port), the SDF scheduler will ensure that there is always at least one token on each input port before the actor fires.

Dangling ports. All channels of a port are required to be connected to a remote port under the SDF domain. A regular port that is not connected will always result in an exception being thrown by the scheduler. However, the SDF scheduler detects multiports that are not connected to anything (and thus have zero width). Such ports are interpreted to have no channels, and will be ignored by the SDF scheduler.

1. The assumed values correspond to a homogeneous actor with no data delay. Input ports are assumed to have a consumption rate of one, output ports are assumed to have a production rate of one, and no tokens are produced during initialization.

3.4.3 SDF ports and receivers

Unlike most domains, multirate SDF systems tend to produce and consume large blocks of tokens during each firing. Since there can be significant overhead in data transport for these large blocks, SDF receivers are optimized for sending and receiving a block of tokens *en masse*.

The SDFReceiver class implements the Receiver interface. Instead of using the FIFOQueue class to store data, which is based on a linked list structure, SDF receivers use the ArrayFIFOQueue class, which is based on a circular buffer. This choice is much more appropriate for SDF, since the size of the buffer is bounded, and can be determined statically[1].

The SDFIOPort class extends the TypedIOPort class. It exists mainly for convenience when creating actors in the SDF domain. It provides convenience methods for setting and accessing the rate parameters used by the SDF scheduler.

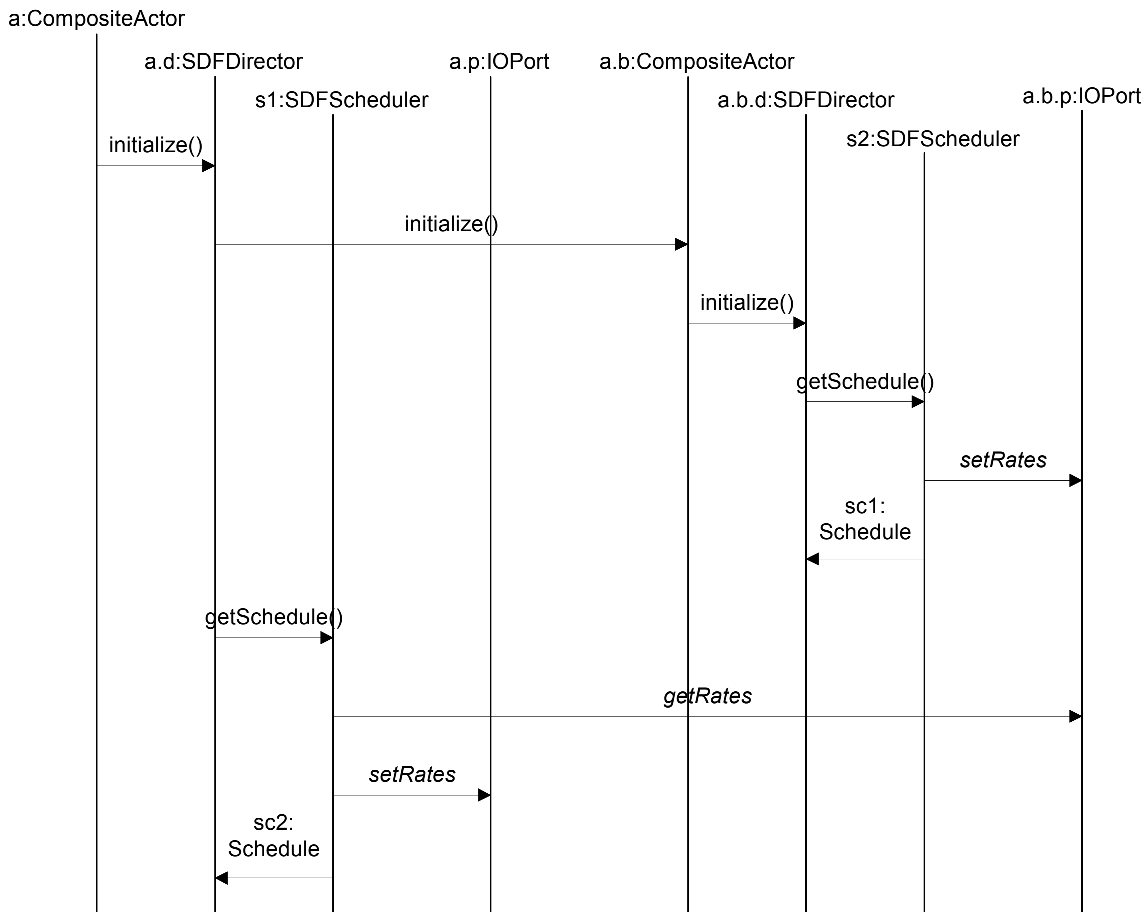

FIGURE 3.12. The sequence of method calls during scheduling of a hierarchical model.

1. Although the buffer sizes can be statically determined, the current mechanism for creating receivers does not easily support it. The SDF domain currently relies on the buffer expanding algorithm that the ArrayFIFOQueue uses to implement circular buffers of unbounded size. Although there is some overhead during the first iteration, the overhead is minimal during subsequent iterations (since the buffer is guaranteed never to grow larger).

3.4.4 ArrayFIFOQueue

The ArrayFIFOQueue class implements a first in, first out (FIFO) queue by means of a circular array buffer[1]. Functionally it is very similar to the FIFOQueue class, although with different enqueue and dequeue performance. It provides a token history and an adjustable, possibly unspecified, bound on the number of tokens it contains.

If the bound on the size is specified, then the array is exactly the size of the bound. In other words, the queue is full when the array becomes full. However, if the bound is unspecified, then the circular buffer is given a small starting size and allowed to grow. Whenever the circular buffer fills up, it is copied into a new buffer that is twice the original size.

3.5 Actors

Most domain-polymorphic actors can be used under the SDF domain. However, actors that depend on a notion of time may not work as expected. For example, in the case of a TimedPlotter actor, all data will be plotted at time zero when used in SDF. In general, domain-polymorphic actors (such as AddSubtract) are written to consume at most one token from each input port and produce exactly one token on each output port during each firing. Under SDF, such an actor will be assumed to have a rate of one on each port, and the actor will consume exactly one token from each input port during each firing. There is one actor that is normally only used in SDF: the SampleDelay actor. This actor is provided to make it simple to build models with feedback, by automatically handling the *tokenInitProduction* parameter and providing a way to specify the tokens that are created.

SampleDelay

Ports: *input* (Token), *output* (Token).
Parameters: *initialOutputs* (ArrayToken).

During initialization, create a token on the output for each token in the *initialOutputs* array. During each firing, consume one token on the input and produce the same token on the output.

1. Adding an array of objects to an ArrayFIFOQueue is implemented using the java.lang.system.arraycopy method. This method is capable of safely removing certain checks required by the Java language. On most Java implementations, this is significantly faster than a hand coded loop for large arrays. However, depending on the Java implementation it could actually be slower for small arrays. The cost is usually negligible, but can be avoided when the size of the array is small and known when the actor is written.