S3F – The Next Generation of SSF

David M. Nicol

January 22, 2011

1 Introduction

The Scalable Simulation Framework (SSF) was proposed over 10 years ago as an API for developing discrete-event simulations. A key motivation for SSF was to lay out an API such that models developed against it could be parallelized.

In its lifetime a number of implementations, variations, and extensions of SSF were developed, the latest and most advanced of these being PRIME [**?**], developed by Jason Liu, who worked also on earlier versions of SSF as a graduate student. In that time we've learned a lot about using the SSF API, things that work, things that are rarely used, and learned a lot about maintaining and using a simulator in an academic context.

A revisiting of SSF was long overdue. Motivated in part by the need for students and staff to be able to extend and modify a simulator for which there was expertise in-house, I took it upon myself to redesign the SSF core classes and interfaces, and develop version v.0 upon which we can build out to connect to the various electrical simulations and virtual systems that will make up the core of the Smart Grid testbed at Illinois. This document describes some of the thinking of that API, and how the system is intended to work.

A problem of course, what name? On reflection I see that any number of adjectives might be given to the the new system, adjectives beginning with the letter 'S'. Simple (perhaps). Second-generation. Standards-compliant. Safe, perhaps. And most probably, Slower. That would make SSSF, which gives one a headache to look at. Let's call it S3F.

2 Interface

The SSF view is that the user program create the simulation model by calling constructors for various SSF objects, initialized them, then pass control to the simulation engine. Control is returned only after the simulation has completed.

Any thing that interacts with the simulation state has to become part of the simulation. We could never get a GPU accelerated wireless channel computation integrated with PRIME because the structure used by the GPU framework had to be embedded within PRIME, and it was not obvious how to do that.

S3F takes a slightly different approach. From the thread of control that starts at `main` the code eventually creates a S3F object called an `Interface`. This creates a number of pthreads for the simulation. Thereafter, the control thread (now also a pthread) and the simulation threads coordinate through barrier synchronizations. In particular, after the simulation model has been built and initialized, the control thread will tell the simulation threads to advance simulation time up to some time it specifies. It frees them to advance the simulation that far, and waits until they have completed. Once they have, the simulation state lays dormant while the control thread may do any one of a number of things, e.g., acquire captured traffic from virtual machines running applications, traffic whose routing is to be simulated by the simulator. When the control thread has accomplished what it needs it may direct the simulation to advance again to a new target in simulation time. Figure 1 illustrates the architecture of this approach.

A thumbnail sketch of the approach is as follows.

1. the control thread creates an instance of a derived class from `Interface`.
2. The derived class constructor passes its parameters to the base `Interface` class constructor. This creates a number of pthreads whose code bodies are the `Timeline::thread_function`. These all immediately synchronize on a barrier that includes them all, and the control function.

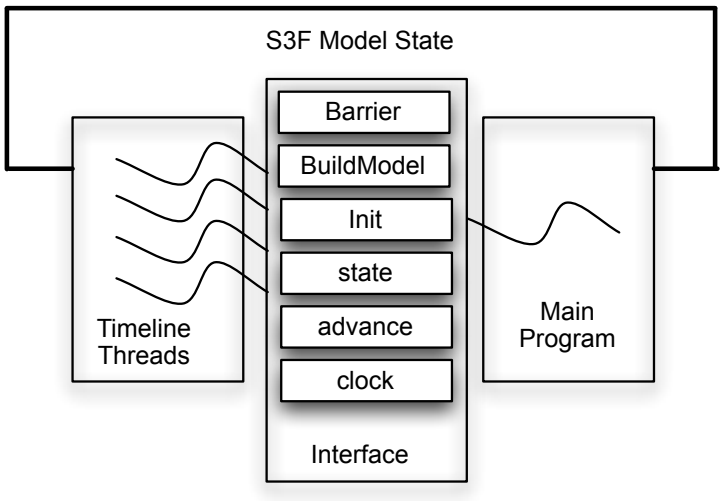

Figure 1: Logical Organization of S3F Simulation Program

3. The control thread continues, and calls `Interface::BuildModel`. This actually runs code written by a modeler or simulation framework builder, and does whatever it likes to create the structure of a simulation model by calling constructors of various S3F objects.

4. After `Interface::BuildModel` returns, the control thread calls `Interface::Init`. This method just calls the `init` method on every S3F "Entity" class instance, methods written by the modeler.

5. The control now calls `Interface::advance` with an argument specifying the length of the next epoch to simulate. The control thread suspends, the Timeline threads are released to advance simulation time as requested, and when they are done the control thread is released while the Timeline threads are suspended.

3 S3F Objects

The S3F modeler has access to a relatively small number of base classes for S3F objects. They are for the most part similar to those of SSF. There are some differences though, which we will highlight.

3.1 Time Types

SSF gave time the type `ltime_t`, and S3F follows suit. Unlike SSF though, S3F insists that `ltime_t` be some sort of integer, preferably a long integer. The time-scale of the clock counter is specified at the time the `Interface` instance is created.

Double precision numbers are particularly useful for generating continuous random variables, as well as other calculations that ultimately have to be interpreted in units of simulation time. S3F provides routines to convert `ltime_t` to double, and back. The units of the doubles involved may be identical to those of the simulation clock, or may be specified separately and be rescaled as part of the conversion.

3.2 Entities and Processes

One ought to think of and SSF or S3F simulation as interactions among a number of `Entity` objects. The simulation state is distributed among entities, entity methods provide the logic of the simulation as they modify the simulation state, the entities communicate with each other. The high level S3F view of `Entity` is very much like the SSF view. Each entity is *aligned* to some `Timeline` object (discussed below), and is "owner" of instances of other simulation objects, `OutChannel`, `InChannel`, and `Process`.

Alignment of `Entities` was in SSF a dynamic sort of thing. The things on which they were aligned—timelines—were not user visible objects in the sense of being able to name them as a class or call their methods. When an `Entity` was created it was by default aligned to its very own timeline, but you could then re-align it onto the timeline of another `Entity`. S3F takes a rather more static view of alignment. The `Timeline` class is visible to the modeler (although creation of a `Timeline` is reserved for the `Interface` class). When an `Entity` is created one specifies the `Timeline` on which it is aligned in the constructor. There is at present no mechanisms built in for dynamically re-aligning entities. This might be considered at a future date, it would not be hard, I've just not thought through the implications.

Under SSF the only code bodies that acted on simulation state the `action` method of the `Process` class. Under S3F any `Entity` method that returns `void *` and takes an `Activation` argument (described later) may serve. Under S3F the `Process` class still exists, but most of its functionality has been moved to the `Entity` class.

The SSF view of a `Process` was that of a logical thread expressed in method `action`. To remain ever-present, the main body of `action` would be written inside of an eternal "while(1) ..." loop. When `action` executed it could change state data, create and send "Events", and ultimately suspend until re-activated. The two basic means of suspension were `waitFor` and `waitOn`. The former took an `ltime_t t` argument and suspended execution of the code body at that point until the simulation clock had advanced exactly `t` units of simulation, at which point execution resumed—again, at that point in the code. `waitOn` identified an `InChannel`; `action` was suspended at the `waitOn` call until such simulation time as an "Event" is delivered to the `InChannel`. Execution resumed at that point, and continued until `action` executed another call that suspended it.

One of the biggest differences between SSF and S3F is in the semantics of executing Processes. In our C++ implementations of SSF we created our own lightweight threading, via source-to-source translation and dependence on the simulation modeler using the right pragmas in all the right places. To say that building our own threads was tricky is an understatement. To say that it is a PITA to maintain our own threads as compilers change is a tautology. I resolved that S3F would not go that route.

Another PITA with respect to code maintenance was the C++ implementation's reliance on third party packages, e.g., for fast memory allocation. So while there are various projects around doing threads for C++, S3F would use whatever was part and parcel of the GNU compiler g++. That leaves us with pthreads, which are a bit heavy-weight for code snippets in simulations.

One of the work-arounds to high thread overheads that SSF allowed for was "simple processes". A process was simple, basically, if any call that would suspend the process was placed in the code so that when the suspended thread was re-animated the code body would go immediately to the top of the encompassing "while(1)" loop. In a simple process the states of local variables did not have to be preserved across suspending calls, and so their implementations did not have to use threading of any kinds. Still, it was sometimes a challenge to express the logic of a process to make it a simple one.

Because of these factors S3F takes a different tack. It has a `Process` class, but it serves mostly as a container for a pointer to a `Entity` that owns it, and a method of that entity which serves as its code body. Every execution of the `Process` body is effectively a subroutine call to that method, made from within the simulation scheduler. Therefore, every execution of the `Process` body begins at the first line of that method. To be sure, one can save state in the Entity that owns the `Process` or the `Process` itself (if the modeler creates class derived from the `Process` base class) that directs the control flow from that point differently depending on the input presented to the `Process` and the saved state. The main point is that ordinary C++ run-time semantics apply whenever a `Process` is called to execute.

Like SSF, S3F provides `waitFor` and `waitOn` calls. The `waitFor` call is different in that it is an `Entity` method (`waitOn` is an `InChannel` method), but is different in a variety of other ways.

- Optionally, the call may specify *a different process* to execute after the elapsed simulation time. This is possible as we have decoupled the notion of suspension from the `waitFor` call altogether. One executing process can schedule the execution of another as a timer, for example. There is a restriction though that the entity that owns the process scheduled by `waitFor` has to be aligned to the same timeline as the owner of the process making the call.

- Optionally, the call may provide an `Activation` to be provided as input to the scheduled process call. We will say more about `Activations` later. For now it is enough to know that it is a smart pointer wrapper around a message.

- Optionally, the call may provide an unsigned integer *priority* to the call. All executions of `Process` bodies are ordered by simulation time, but priorities may be included to order the execution of processes scheduled to execute at precisely the same time. As with simulation time, the smaller the value, the higher the priority. S3F allows 16 bits of priority to be expressed by the user (and reserves some bits for itself).

While not part of the S3F standard API, experience with SSF highlighted the utility of allowing a user to create a sort of global name space where entities and inchannels may be identified with text strings. The original motivation was construction of models across machines that do not share memory. Eventually (soon even) S3F will have to wrestle with that bear, and so the functionality has been built in. When an Entity is constructed one may optionally provide a string name. The name needs to be unique among all Entities created in the model. The string is entered into a static `Entity` class directory, with a method provided for looking up the address of an entity, given its string name.

3.3 Timeline

A `Timeline` hosts an event list, and is responsible for advancing the simulation related to all entities aligned to it, using that event list. All of the underlying logic of event execution is contained in the Timeline, but is hidden from the user. The pubic interface to the `Timeline` class is very limited. The user can obtain its current simulation time (basically the time stamp of the current event being processed, or a time before which no future event will ever execute), and the time-scale of the clock ticks. It also reports its `event horizon`, which is a technical notion we'll save for the discussion on synchronization.

3.4 InChannel

The S3F notion of an `InChannel` is very much like SSF's, although the mechanics of using them is a little different. In both systems it serves as an endpoint for a communication channel. In both systems one can use `waitOn` to tell the simulation to awaken some process when a message is delivered to the `InChannel`. S3F continues the SSF notion that delivery is taken of a message that arrives to an inchannel at simulation time t only by those processes that have attached themselves to that inchannel. There is no buffering. A message may arrive to an inchannel with no processes listening, and if it does it is silently discarded.

S3F changes the way processes are attached to inchannels. Under SSF `waitOn` was a `Process` method, and the call identified one or more inchannels the process would then suspend on. Delivery of a message to any one of those inchannels would unsuspend the process.

Under S3F `waitOn` is an `InChannel` method. It has one argument (which may be implicit), the process that should be activated when a delivery is made to the inchannel. The owner of the process so identified must be on the same timeline as the owner of the InChannel.

In the semantics of SSF, the attachment of a process to an inchannel disappeared just as soon as the `waitOn` call returns. S3F allows this interpretation, but also allows the binding to persist past the execution of the process body resulting from an attachment. To wit, if a process is attached to an inchannel by a `waitOn` call and as a result of an arrival the process is executed, the process will **not** be called on the next arrival **unless** it is passed to the inchannel again by another `waitOn` call. However, one can make the attachment persist across process executions by calling `InChannel` method `bind`. Like `waitFor` the argument to `bind` is a process (either explicit, or implicitly the process executing when the call is made). Like `waitOn`, the owner of the process being bound must be on the same timeline as the owner of the inchannel.

Decoupling `waitOn` (or its sibling `bind`) from suspension allows for an odd corner case. If a process body is executing owing to an arrival on inchannel I, and then calls `waitOn` *again* to ensure that the process will be activated with the next arrival, the actual association will not take place until all processes activated by that arrival have executed. So it is possible for a process that has done a `waitOn` on I to be activated, do another `waitOn` on I (which does not suspend the code), then use an InChannel method to ask whether it is waiting on I and have the answer returned be "no".

In addition to `waitFor` and `bind`, S3F provides methods for canceling prior calls to these methods, and for querying whether a given process is either bound or waiting on the channel. It provides a means for ordering the execution of all processes that are activate by a common arrival, about which more, anon.

Like entities, inchannels may have string names declared for them at time of construction. Unlike entities though, the inchannel name has to be unique only among inchannels on the same entity. This makes it easy, for example, to associate an inchannel with some well known interface, e.g., port80. Coupled with textual entity names, we have a

global name space for identifying an inchannel—the entity name paired with the inchannel name. The utility of such a name space is better appreciated in the context of the `OutChannel::mapto` function, discussed below.

3.5 Message and Activation

SSF provided the `Event` base class as the basis for communication. Methods involved with the sending and receiving of Events passed pointers to the events. This design decision was fine for Java where garbage collection is done automatically, but was a nightmare for the C++ implementation. As we will see, a sent message may be silently duplicated for multiple destinations. It is expensive to have the system make unique copies of each message, but if the message is shared, complicated to delete it at the proper time. Our C++ implementations created their own schemes to do reference counting, but these still required active modeler specification of when a reference to an event was no longer needed.

Now ten years later a great deal of thought and effort has gone on in the C++ community about smart pointers. The current standard—embedded already in the GNU g++ compiler—is the `shared_ptr` smart pointer in the `std::tr1` namespace. Problem solved.

I never liked the use of the term "Event" in SSF for what is actually a message, particularly in the context of a discrete-event simulator! So the name in S3F is different, the base class is `Message`. The base class constructor allows one to pass along (and carry along) an integer type for the message, the utility of this will be noted shortly.

S3F contains a typedef using the term `Activation` to refer to a `shared_ptr<Message>`. For a communication to have any significant content at all, the user needs to create a class derived from `Message` to carry what he wants. A user that wants to send a message of type `MyMessage` (derived from `Message`) creates an instance of it, e.g. suppose `mmsg` is a pointer to a `MyMessage`. He creates an Activation such as `Activation act(mmsg)`, and passes `act` *by value* in the communication. Likewise on receipt an activation is passed by value. The passing by value is the magic part needed for reference counting.

When a process is activated owing to a prior `waitOn` (or even a **waitFor** its code body is passed one argument, an activation. C++ does not have introspection, all the code body knows is that the activation wraps a pointer to a `Message`. Downcasting is possible, but downcast to what subclass? If the message is from a derived class of `Message` there is no native way of inferring that. S3F helps in part by having `Message` carry a "type" code, offered at creation. If the recipient of an activation does not know what the derived type is by context, it can get the base class type, so that the modeler can create a code mapping integers to subclasses.

The beauty of the thing is that activation wrapped messages can be placed in STL containers, passed from one routine to another, have multiple wrappers of a common message be delivered to multiple inchannels—even multiple timelines—and the protected message will be automatically deleted when—and only when—the last activation protecting it has past out of scope.

3.6 OutChannel

An instance of the `OutChannel` class is the "send" point for an Activation. The constructor has a manditory argument—a pointer to the Entity that "owns" it, and an optional argument that will make sense later. The main two methods are `write`, and `mapto`. Both have many variations that work through all combinations of optional arguments.

`write` initiates the transfer of an activation to one or more inchannels (which we will see are specified by `mapto` calls). The one required argument is an instance of an activation. Optionally one may include a "per-write" time increment. The optional second argument of the `OutChannel` constructor is a lower bound on what value may be given as a "per-write" increment. Optionally one may include a scheduling priority in the `write` call. We will comment more deeply on the time increment later. Here we just accept that if the per-write increment is not given, it automatically takes value 0. The scheduling priority is like that allowed for a `waitFor` call, used in tie-breaking. Section S4 goes over these completely.

A `mapto` call has two parameters, a pointer to an inchannel, and a time delay. It is a declaration that that every write made to the outchannel will be delivered to the the named inchannel. The time delay is known as the *transfer time*, and is involved in computing the time at which the activation presented by a write will be delivered to the inchannel.

If an inchannel I is targeted by a `mapto` with transfer delay d and at time t acall is made to the outchannel's `write` method with per-write delay w, then the arrival time of the activation at an inchannel is $t + d + w$. The

motivation for this approach lies in S3F's support for parallelism. w is a lower bound on the delay between the sending of any activation from the outchannel to the inchannel. Indeed, if the outchannel was constructed with an optional lower bound b on the per-write delay, then $b + w$ is a lower bound. The synchronization mechanism uses this kind of information to establish synchronization windows.

While the factoring of the delay into two parts may seem odd, it has its uses beyond support of parallel simulation. If the outchannel to inchannel mapping represents a communication line between two switches, one component of the the latency is due to that line's bandwidth, a component that is fixed. A second component is due to queueing delay. When the queue is FCFS, the arrival time of the activation can be computed at the time the message joins the queue, e.g., is "written". Here the per-write delay would be computed as the time in queue before the message is sent.

3.6.1 return values

`write` returns a `bool` to indicate whether the activation will be delivered all to inchannels. Why *wouldn't* an inchannel receive the activation? A write will always be successful if its per-write delay is no smaller than the minimum promised when the outchannel was constructed (or, as it happens, a later change to that promise). However, we don't automatically reject a write if its per-write delay is smaller than promised. Depending on a number of things it might still be possible to deliver the activation within the constraints of the synchronization system. So the `write` returns `true` if all the mapto'ed inchannels will get their activations, and returns `false` if one or more of them will not. All of the inchannels that *can* get the activation within the constraints of the synchronization system can. If `false` is returned S3F provides enough handles on state information so that a model could calculate which inchannels did not receive the delivery.

On the other hand `mapto` returns a simulation time. The interpretation is that the time is an upper bound on the simulation time by which the `mapto` will have been implemented. This approach supports dynamic creation of `mapto` statements. A `mapto` call that is made not at initialization, but when the simulation is running may create a linkage that affects synchronization. Those calls are identified, the `mapto` request is buffered until it can be performed safely, and the earliest time the `mapto` can take place is returned.

3.6.2 dynamic changes

The `OutChannel` class supports many ways of dynamically changing the connections between the outchannel and the rest of the world. We have seen already that a `mapto` can be dynamically introduced. One can also dynamic `unmapto` an inchannel and remove the connection. Any activations that are "in flight" still will still be delivered at the scheduled time. The transfer time on a mapped outchannel-¿inchannel connection can be dynamically changed, but like the `mapto` this call returns a simulation time at which the change will be implemented, possibly in the future. Similarly one can dynamically change the outchannel's per-write minimum delay, and similarly that call returns a simulation time at which the change will be implemented.

Support for dynamic connections is motivated by our experience in using SSF to model mobile ad-hoc networks where the connections *do* change, and for allowing sophisticated models to offer larger lower bounds on future write deliveries, lower bounds that are used to establish synchronization among timelines.

4 Priorities

Our implementation of S3F gives each timeline a single STL priority queue for an event list. All events of all types go into this priority queue. The event records are ordered first by simulation time, and, among events with the same simulation type, a priority—an unsigned integer between 0 and $2^{16} - 1$. As with simulation time, a lower priority value implies that the action should take place earlier than one with a higher priority value. The priorities are used in two places, the event-list, and the list of processes to activate when an activate is delivered to a particular inchannel.

Deep in the bowels of S3F an "event" is a record that contains an event type, a simulation time, a priority, and other information, depending on the event type. A `waitFor` call creates a timeout event that includes any activation provided with the call, and the process (implicit or explicit) to be activated when the time arrives. A priority may be explicitly included in the `waitFor` call. If it is not, the default priority of $2^{16} - 1$ is put into the event record.

Another kind of event is for the arrival of an activation to an inchannel. This kind of event is the result of a `write` to an outchannel; for every inchannel mapped to that outchannel, a different event is inserted in the event list of the timeline on which the inchannel's owning entity is aligned. So it is with the `write` method that we have the

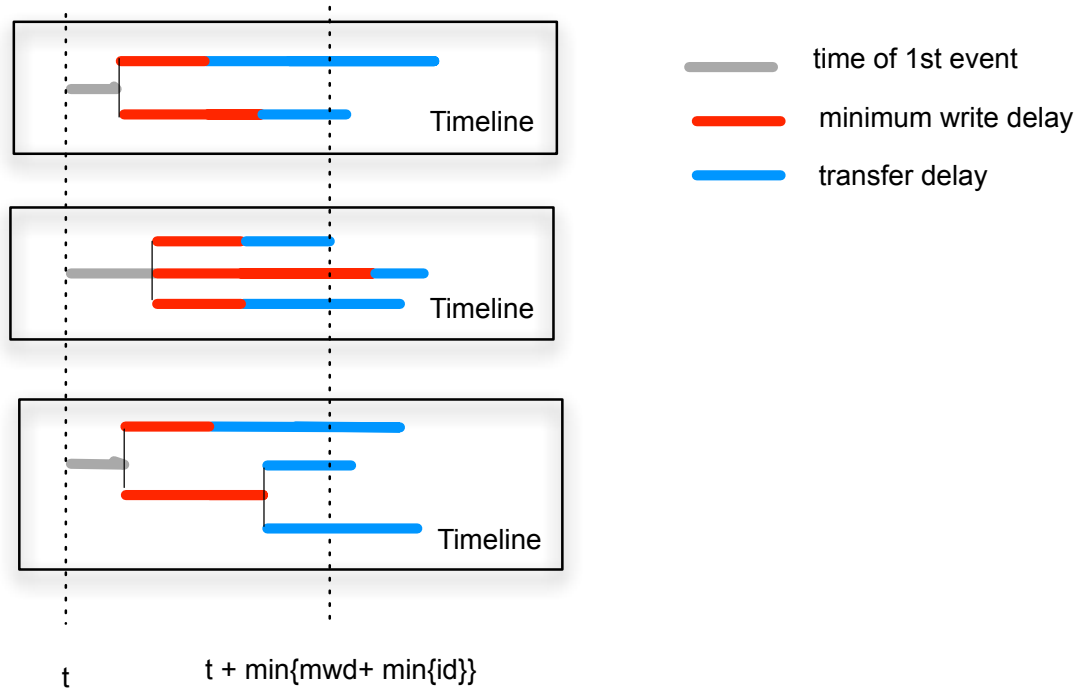

Figure 2: Synchronization Window

opportunity to include a priority to be carried to the event record(s) that result. As before, if no priority is included with the `write` the default of $2^{16} - 1$ is used.

Yet another place where priorities are used to order concurrent activations is in the list of processes to activate when an delivery is made to an inchannel. Processes join an inchannel's list by dint of `waitOn` and `bind` calls. However there are no priority arguments to these. Instead, S3F associates a priority state value with each `Process` instance and provides methods for reading and writing these. When a process is attached to an inchannel's list, it is inserted into that list ordered by increasing process priority code. Then, when an event that delivers an activation to an inchannel is executed, the processes are activated one at a time, in the order they appear in the list.

5 Synchronization

S3H supports parallel execution, which requires synchronization among the Timelines. Much of the motivation and design of S3F is to support synchronization more-or-less transparently, yet provide hooks to the sophisticated modeler to transfer modeling information to the synchronization engine that is improve performance.

The basic idea behind synchronization is simple. Use information about latencies across communication paths established between outchannels and inchannels to establish a window of simulation time with the property that no activation written to an outchannel at a time within that window will be received by an inchannel *on a different timeline* at a time also within that window. The windows are implemented with barrier synchronizations. The upshot is that writes whose activations cross between timelines do not have to be immediately delivered—their receipt lies on the other side of a barrier synchronization, so they can be buffered, and later integrated into their target timeline event lists.

Figure 2 illustrates the concept. Suppose the Timelines have all advanced to simulation t, and have in their event lists all events known (at that time) to be executed. Execution of some of these events may of course introduce other events into the list, but every future event known at time t is in the event list of the timeline that will execute it. The timelines are working to coordinate how far ahead in simulation time they can safely advance. Each timeline identifies the time of the first event it has in its list, which is a lower bound on when next the timeline will execute a process

that performs a write that crosses timeline boundaries. Each timeline also identifies the minimum over all mapped cross-timeline outchannel/inchannel pairs of the outchannel's minimum per-write delay, and the transfer delay to the inchannel. That is, each timeline i identifies a lower bound on the arrival time of any future cross-timeline write as

$$L_i = n_i + B_i$$

where we separate the bound into time of next event n_i, and

$$B_i = \min_{\textit{outchannel } c} \left\{ w_c + \min_{\textit{X timeline mapped inchannels } k} t_{c,k} \right\}$$

where w_c is the minimum per-write delay declared for outchannel c, and $t_{c,k}$ is the transfer time between c and inchannel k owned by an Entity aligned to a different timeline than c's owner. To establish the synchronization window the timelines offer their respective L_i values to a global min-reduction. On being released from the associated barrier, each timeline can read the minimum among all offered values—this is the upper edge of the window. Simulation time may advance to one clock tick less than this value.

The value of n_i may change from window to window, but B_i need not, at least so long as there are no changes in the inchannels mapped, the transfer delays, or the minimum per-write delays, the result of a prior computation can be used. However one cannot always expect the set of mapped outchannels/inchannels to remain constant, and changes in model state may allow (or require) the model to change an outchannel's per-write minimum delay, or some transfer delay. As we have seen already, S3F allows dynamic `unmap` and `mapto` calls, and dynamic changes to minimum per-write and transfer delays. S3F tries to minimize the impact of those changes by being smart about recomputing B_i. Once computed, S3F also counts the number of cross timeline outchannel/input pairs that actually achieve the computed value of B_i. Any change in mapped channels or their delays that decrease that count need not trigger a recomputation so long as the change leaves at least one outchannel/input pair with B_i's value. Likewise any change that cannot possibly lower B_i (e.g., *increasing* either the per-write minimum delay or a transfer delay) will not trigger recomputation of B_i. However, requests for changes that do affect B_i raise a flag that later triggers re-computation of B_i.

Calls to `OutChannel::mapto`, `OutChannel::new_transfer_delay` and `OutChannel::new_min_write_delay` may request changes that cannot be safely implemented immediately. These requests are queued to be processed in FCFS order after a timeline has executed all events in the current event window. FCFS processing implies that within a window a number of changes to a channel might be requested and buffered, but for both minimum per-write delay and transfer delay, only the last change requested "takes".

Processing of queued delay change requests may raise the flag that B_i must be recomputed. After the change requests are processed, that flag is tested, and if true, S3F does a complete analysis of the timeline's delays, and recomputes B_i.

After a timeline has processed all the window's events, implemented any buffered delay changes, and recomputed B_i (if needed), it sets its clock to the time of the window edge minus one clock tick, and enters another barrier synchronization to wait for all other timelines to do so also. When they are released, they transfer buffered events that resulted from cross timeline writes into their event lists (thereby ensuring that each time of next event n_i is what it needs to be), and compute the upper edge of the next synchronization window.