SCALABLE SIMULATION FRAMEWORK API REFERENCE MANUAL

VERSION 1.0

DOCUMENTATION DRAFT - REVISION MARCH 25, 1999

James H. Cowie, Editor

1. Introduction

This reference manual specifies the detailed syntax and semantics of the Java and C++ bindings of version 1.0 of the Scalable Simulation Framework Application Programming Interface (SSF API), and is intended as a resource for SSF implementors and expert modelers. Modelers seeking an introduction to the use of SSF for constructing discrete-event simulations may prefer to start with the companion document, the `SSF User Manual`.

SSF provides a single, unified interface for discrete-event simulation (the SSF API). Object-oriented models that utilize and extend the framework can be portable across SSF-compliant simulation environments. This maximizes the potential for direct reuse of model code, while minimizing dependencies on a particular simulator kernel implementation. In addition to its concrete modeling applications, the API also functions as an abstract target for compilation of models specified in higher-level modeling languages or graphical modeling environments.

The framework's primary design goal was to support *high performance simulation.* SSF makes it possible to build models that are efficient and predictable in their use of space, able to transparently utilize parallel processor resources, and scalable to very large collections of simulated entities.

2. Framework Summary

The SSF syntax comprises five base class interfaces, `Event, Entity, inChannel, outChannel,` and `process`. The following sections describe the common semantics of compliant implementations of these classes. Appendix A supplies some additional observations and interpretations that may be useful in understanding the specification. Java and C++ class bindings are specified in Appendix B. Finally, Appendix C gives a summary of SSF implementation dependencies.

3. Terminology

- *Implementation-dependent* means that the behavior of the framework may vary among SSF implementations.
- An *SSF-compliant implementation* is one that provides appropriate semantics for each method of each SSF class as described in this document. An SSF-compliant implementation may not extend the SSF API with additional non-standard methods or classes, except through object-oriented derivation of new classes that extend the SSF base class interfaces without modification to their semantics, and without requiring particular implementation-dependent semantics of those base classes. Such a package of classes is said to constitute an *SSF extension.*

- An *SSF-compliant model* is one that extends or instantiates the framework's classes, without depending on any particular outcome of an implementation-dependent behavior.
- A *pure SSF model* is an SSF-compliant model that does not depend on any SSF extensions for correct operation that it does not itself provide.

4. Simulation Model

The SSF host language is the language in which the user writes SSF model code: for example, Java or C++. An SSF model is a conformant program in the host language, which at some point instantiates one or more entities, processes, and channel endpoints (cf. sections 6, 7, 8). The modeler defines the behavior of entities by associating with them processes that are to be executed on their behalf.

4.1. Starting the Simulation A simulation starts when any entity's `startAll()` method is called, commonly from the `main()` routine, or the equivalent in the local environment. The results of any subsequent calls to the `startAll()` of any entity are implementation-dependent. The caller of `startAll()` specifies two timestamps: the simulation's *start time*, which defaults to 0, and the *end time*. The simulation will execute over this interval of simulation time, inclusive of the endpoints.

4.2. Initialization After `startAll()` has been called, but before any process has started executing, the framework calls the `init()` routines of all processes and entities. The exact order of initialization is implementation-dependent, and may be concurrent, with the exception that no process will be initialized until the entity which owns it has completed initialization. The Entity method `now()` shall return the simulation start time for the duration of initialization.

4.3. Process Execution After all processes have been initialized, they become eligible to run at the start time; their actual execution is controlled by the framework, which executes the process's `action()` callback method. Every time the `action()` method returns to the caller, the framework executes it again immediately, as many times are are necessary to reach the end of the simulation.

Within the `action()` method, or within code called by `action()`, the process may elect to `wait` for a condition to be met; it will be suspended by the framework, and resumed when the condition has been met.

4.4. Framework Inner Loop The SSF framework provides nonpreemptive scheduling of processes. Once resumed by the framework, in order to allow the simulation clock to advance, processes must eventually suspend themselves by issuing `waitOn()` or `waitFor()` calls. All computation performed during a single resumption of a given process takes place "simultaneously" — that is, at the same simulation time.

Within the framework, simulation time advances at a (possibly variable) rate determined by the arrival of events on channels, and the duration of `waitFor()` statements. When simulation time exceeds the end time specified in the original `startAll()` call, the simulation ends.

See section 7 for a more extensive discussion of processes.

4.5. Start, Pause, Resume, and Join The `startAll()`, `pauseAll()`, `resumeAll()`, and `joinAll()` methods may not be called from within process code.

The `pauseAll()` method allows the simulation to be paused gracefully after some implementation-dependent delay. A call to `pauseAll()` blocks until the simulation has paused, and then returns the current simulation time. After `pauseAll()`, a call to `resumeAll()` resumes the simulation's forward progress. At any point, a call to `joinAll()` (e.g., from `main()`) will block without possibility of pause or resumption until simulation execution is complete.

4.6. Framework concurrency Each entity is said to be *aligned* to some object (cf. 6.2) This alignment target, whose exact type is implementation-dependent, represents a common scheduling authority (in parallel discrete-event simulation parlance, a *logical timeline*) for the processes of coaligned entities; each set of coaligned entities is said to constitute an *alignment group.*

Processes that are eligible to execute at each instant of simulation time will be scheduled by the framework: sequentially within an alignment group (but in implementation-dependent order), and concurrently across alignment groups. Put another way, the framework transparently manages any required synchronization to support in-order event exchange among the entities aligned to a set of potentially concurrent logical timelines.

4.7. Simulation time. The behavior of an SSF model is defined by the collective actions of its processes on its state. Every process resumption takes place at a particular simulation time, and each modification of model state by that resumption of the process takes place at that time.

The SSF framework guarantees that from the perspective of any single process, event-based interaction with other processes takes place in monotone nondecreasing timestamp order. That is, a process executing at time T will never receive an event whose scheduled delivery time is less than time T.

The SSF framework cannot make similar guarantees for direct modification of model state by processes. A process may not directly access model state that may be modified by non-coaligned processes (that is, processes belonging to entities in other alignment groups). An SSF model in which a process accesses model state that may be modified by non-coaligned entities has the potential for out-of-logical-order interaction, and is therefore undefined.

5. Events

Modelers may create arbitrary derived classes from the event base class provided by the SSF framework. All derived event classes must provide their own public copy constructor.

5.1. Event References: Save and Release Event storage management is the responsibility of the framework, which may release the storage of an event at any point after its last recipient process has suspended, unless the modeler has explicitly acquired a reference to it using the `save()` method and not released the reference using a matching call to `release()`.

The `save()` method takes as its single argument a pointer to a written or received event. Save() may be called more than once on the same event; the framework may not release the event's storage until a matching number of `release()` calls have been called on the event. Save() returns a pointer to the saved event; this event is guaranteed to be value-identical to the original event (as defined by the modeler's copy constructor), but may occupy different storage.

5.2. Event Aliasing An entity's `aliased()` method shall return true if it may be accessible to processes of multiple, noncoaligned entities. This aliasing condition may arise, for example, when an event has been written between a noncoaligned sender and receiver, or written to multiple noncoaligned recipients. The results of modifying the state of an event whose `aliased()` method returns true is undefined.

6. Entities

An Entity is an Object that can **own** processes and channel endpoints, and can be **aligned** with other Entities.

6.1. Ownership Each entity is said to **own** zero or more processes, inChannels, and outChannels. The base class constructors for the process, inChannel, and outChannel classes each take one argument, of type Entity, specifying the owner of the process, inChannel, or outChannel instance. This ownership is immutable. The lists of processes, inChannels, and outChannels owned by an entity are returned by the `processes()`, `inChannels()`, and `outChannels()` methods of Entity.

6.2. Alignment Each entity is aligned to some implementation-dependent object that represents an alignment group; the default alignment of an entity for which no alignment is specified is implementation-dependent, as is the maximum number of distinct alignment groups that can be created.

6.3. Realignment An entity receives an overriding alignment when its `alignto()` or `makeIndependent()` methods are called.

- The `alignto()` method takes one argument, a reference to the entity whose alignment serves as the new alignment target, and realigns this entity to the argument entity, effective at the return-value timestamp.
- The `makeIndependent()` method takes no arguments. If the creation of a new alignment group would exceed the maximum number of distinct alignment groups supported by the framework, `makeIndependent()` has no effect and returns `now()`; otherwise, it realigns this entity to a new, unique alignment group, effective at the return-value timestamp.

Both forms of realignment are subject to the following rules:

- Realignments may not be issued during entity or process construction.
- Realignments issued during the initialization (via `init()`) of a process or entity are always immediately effective, and shall return the simulation start time.
- Realignments issued during the execution of a process (via `action()` or code called by `action()`) shall return a timestamp greater than or equal to `now()`; the exact value of the timestamp returned is implementation-dependent. The modeler must wait until the specified time has arrived to take any action whose correctness depends on the new alignment.

Finally, it is an error to realign an entity if the realignment would result in the creation of channel mappings with zero minimal delay and zero mapping delay between noncoaligned entities.

7. Processes

7.1. Construction and Initialization A process may not be created by another process unless both processes are owned by coaligned entities. The process constructor takes as an argument a reference to the owner entity; the framework will invoke the process's `init()` method for post-construction initialization at some point after the owner entity's `init()` method has been called, and before the new process has run for the first time. The new process becomes eligible to run at the current simulation time.

Each process specifies a callback method (`action()`) that the framework executes one or more times during the simulation. This code consists of arbitrary computation that takes up zero elapsed simulation time, interspersed with one or more wait statements that take up non-negative amounts of simulation time.

In the C++ API, a process may "borrow" a callback method from its owner entity, specified as the second constructor argument, and the default `action()` will be to call this method, passing it a pointer to the process. Such a process may not override `action()`.

7.2. Wait Statements A process's wait methods may not be called from any context other than the `action()` or `init()` methods of the process itself, or in code called by those methods.

7.2.1. waitFor(time_t T) Suspends the process until T ticks of the logical clock have elapsed. The interval T must be of greater than zero duration.

7.2.2. waitForever() Suspends the process without the possibility of resumption.

7.2.3. waitOn

- **waitOn(inChannel I)** Suspends the process until an event arrives on channel I.
- **waitOn(inChannel[] Iset)** Suspends the process until an event arrives on some channel in the provided channel set.
- **waitOn()** Suspends the process until an event arrives on some channel in the default channel set, if one has been specified during initialization using `waitsOn()`. If no default channel set has been specified, suspend the process without the potential for resumption. If the `waitsOn()` declaration is executed more than once per process, the most recent call determines the default channel set.

7.2.4. boolean waitOnFor(inChannel[] Iset, time_t T) Suspend the process until an event arrives on some channel in the provided channel set, or until T ticks of the logical clock have elapsed, whichever comes first, returning true if the wait timed out, and false otherwise. The interval T must be of greater than zero duration.

7.3. Simple Processes A `simple function` is defined as one that returns control to the caller immediately following each `waitOn(), waitFor(),` or `waitOnFor()` statement, and which calls only simple functions.

By default, a process's `isSimple()` method returns false. It may be overridden by the modeler in derived classes to return true if and only if the `action()` method of the process is a simple function.

7.4. Process Scheduling Processes owned by entities belonging to different alignment groups may execute concurrently. Within an alignment group, the framework shall implement appropriate process scheduling policies and write buffering strategies to guarantee fairness. Fairness is defined by the requirement that no process may execute twice at simulation time T, or affect any other process by means of an event written with zero delay, before all other eligible processes have executed once. The process scheduling policy shall further ensure that if multiple processes are eligible to run at simulation time T, all processes that suspended via `waitFor()` shall execute before any process that suspended via `waitOn()` or `waitOnFor()`.

8. inChannels, outChannels

Like a process, both inChannel and outChannel take a reference to an owner entity as their first constructor argument.

8.1. Mapping Channels Associated with each outChannel is a list of zero or more inChannels to which the outChannel is currently mapped. Associated with each inChannel is a list of zero or more outChannels that are currently mapped to the inChannel. The maximum size of these mapping lists is implementation-dependent.

Executing `O.mapto(I)` creates a new mapping between outChannel O and inChannel I. The `mapto()` method returns a simulation time T at which the mapping becomes effective. If `mapto()` would cause the source or target list of mappings to exceed the maximum length defined by the implementation, the mapping will fail, returning an effective mapping time that is greater than the end of simulation time.

It is an error to create a channel mapping whose minimal channel delay and minimal mapping delay are both zero between noncoaligned entities.

8.2. Writing Events A process sends an event E on outChannel O with optional per-write delay D by issuing a write statement:

```
O.write(E,D);
```

A process may write to any outChannel that it can access via normal program scope, as long as the owner entity of the process and channel are either identical or coaligned.

When an event is written on a channel, its receive time (the time at which it will be made available to the recipient) is greater than or equal to its send time (the time at which it was written). Receive time is greater than send time when delays are present; there are three ways of specifying these delays.

- An outChannel may have an inherent `minimal channel delay` associated with it when it is constructed, by supplying a nonzero delay as the second constructor argument.
- A particular mapping between an outChannel and an inChannel may have an additional `minimal mapping delay`, specified using the optional second argument to `mapto()`.
- Finally, a particular event transmission may be given an additional `per-write delay` on each write to the channel, specified using the optional second argument to `write()`.

These three sources of delay are independent, additive, immutable, and equal to zero by default. The event will be delivered at the time it was sent, plus the minimal channel delay (if any), plus the minimal mapping delay (if any), plus the per-write delay (if any).

8.3. Receiving Events The event that is delivered is guaranteed to be equal to, but not storage-identical with, the event that was written. For these purposes, "equal" is defined by the modeler in the form of a mandatory copy constructor, which must appear in every Event derived class. See section 5.1 for a detailed discussion of event storage management.

A process may receive events from an inChannel if and only if the owner of the receiving process and the inChannel are identical or coaligned. To receive events from an inChannel I, a process may issue an $activeEvents()$ statement:

```
Event[] E_set = I.activeEvents();   // Java
Event** E_set = I->activeEvents();  // C++
```

In C++, this set of events will be null-terminated. In Java, the built-in array `size` operator returns the received event count. In either case, the storage for both the array and the delivered events themselves belongs to the framework and may be reclaimed by the framework after last event receipt, unless the modeler has specified otherwise (via `save()/release()`, section 5.1).

Event receipt is nondestructive. One, some or all of the processes of an entity are eligible to receive each event received on each inChannel of that entity. Each process shall be given the opportunity to receive, exactly once, each of the events scheduled for delivery on all coaligned inChannels in the current instant of simulation time. A process that explicitly advances in simulation time (via `waitFor()`) forfeits the right to receive any events still pending on coaligned inChannels.

If no process receives an event at its time of delivery, the event vanishes; it is **not** buffered by the framework for later redelivery.

8.4. Unmapping channels The `unmap()` operation allows a channel mapping to be broken, returning a simulation timestamp equal to or greater than `now()` at which the unmapping will be effective. Delivery of pending events on a newly unmapped channel is implementation-dependent.

It is not an error to write an event to an unmapped channel, or to wait on input from an unmapped channel; in the first case, the event simply disappears, and in the second, the process blocks as if the channel were mapped but no events were available.

Appendix A. Commentary

3. Terminology

SSF-compliant models should be portable between same-host-language implementations of SSF without modification, or across different-host-language implementations after making language-specific modifications (e.g., to replace Java references with C++ pointers). Where possible, modelers should avoid specific host-language features that restrict cross-language portability, such as multiple inheritance, pointer arithmetic, or dynamic type inspection.

4. Simulation Model

4.5. Start, Pause, Resume, and Join Calling these from within a process body would cause the simulation to deadlock. These are methods for external control of a simulator; e.g., from a tool such as a console. If a simulation component wishes to signal the need for a pause (e.g., because an error count has exceeded a significant theshold), it can set a flag or execute a callback to external code to make the condition visible. These are instance rather than static methods in order to allow multiple simultaneous simulations to exist within the same address space.

4.7. Simulation time A process may freely modify and/or access its own state, and the state of the entity that owns it, and the state of all entities which are known to be coaligned with that entity, and the state of all processes, inChannels, and outChannels that are owned by those coaligned entities. The modeler is solely responsible for writing processes that test coalignment and access shared state safely. SSF implementations need not protect against, or even detect, unsafe access to shared state. If in doubt, a process must send and receive query events, rather than accessing entity state directly.

Event exchange is the modeler's only available mechanism for coordinated interaction among non-coaligned processes. Simple wallclock synchronization/exclusion (e.g., via the `synchronized` keyword in Java) is **not** sufficient to provide logical-time synchronization and avoid logical-time causality errors.

Note that though access to state modified by a non-coaligned process is restricted, non-coaligned processes may share access to read-only data; for example, initialization data held in common. Similarly, noncoaligned processes may interact with systems outside the model altogether (such as debuggers and visualizers) as long as the external system is aware of the potential for out-of-order updates and doesn't propagate that information back into the model.

5. Events

5.1. Event References: Save and Release Of the following three variants of event-buffering code, only the last is a legal use of reference-saving.

```
Event** eset = I->activeEvents();

for (n=0; eset && eset[n]; n++)
   buffer->addElement(eset[n]);    // ILLEGAL: Event not saved.
waitOn(I);
```

```
for (n=0; eset && eset[n]; n++){
   eset[n]->save();               // Oops .. ignored return value
   buffer->addElement(eset[n]);   // ILLEGAL: may be different storage
}
waitOn(I);

for (n=0; eset && eset[n]; n++)
   buffer->addElement(eset[n]->save());  // LEGAL.  Save a reference.
waitOn(I);
```

The fact that `save()` may return a different pointer than the original saved event enables better storage management by the framework. For example, an implementation might call the copy constructor, and return a pointer to the copy, electing to free the storage of the original event. Another implementation might do a bitwise copy of the saved event to a new location (to minimize heap fragmentation) and return a pointer to the new location of the "same" event.

5.2. Event Aliasing Note that the `aliased()` method only detects aliasing caused by the framework's implementation-dependent copy strategy in the presence of events deliverable within multiple timelines.

It need not (indeed, cannot in general) detect storage-level aliasing within an alignment group. The intent is to avoid causality violations among noncoaligned processes, rather than simpler conflicts among coaligned processes; read-after-write hazards among non-coaligned entities can easily result in information travelling backwards through simulation time.

6. Entities

6.1. Ownership. In SSF, entities, rather than processes, are the quantum of mobility. Because ownership is immutable, the processes and channel endpoints owned by the entity travel with it when it is explicitly realigned.

6.2. Alignment Coalignment of entities affects model correctness in several significant ways. First, only coaligned entities may access or modify each others' state directly, rather than indirectly by means of event exchange (4.7). Second, processes may only wait for events to arrive from inChannels associated with coaligned entities (8.3), and may only create new processes associated with coaligned entities (7.1). Finally, channel mappings whose endpoints are associated with non-coaligned entities must have positive minimal delay (8.1, 6.3).

One common implementation of default alignment will create a pool of P timeline objects, where P is the maximum sensible degree of available concurrency, such as the physical processor count. The default alignment for new entities may then be chosen round-robin from this pool of alignments, overridden by the modeler if desired. In a sequential implementation, there may be only one default alignment common to all entities, thus making explicit `alignto()` and `makeIndependent()` methods purely annotative.

6.3. Realignment `MakeIndependent()` allows the modeler to "fork" a single logical timeline into two, allowing the modeler to partition a set of coaligned entities into two sets using `makeIndependent()` followed by suitable use of `alignto()`.

Note also that `alignto()` is not dynamically transitive:

```
A.alignto(B);
B.alignto(C);  // A not necessarily aligned with C

E.alignto(F);
D.alignto(E);  // D is now aligned with F

E.alignto(G);  // D is still aligned with F
```

Think of `alignto()` as a tool for adjusting alignment set memberships, rather than creating a durable relationship between two entities. If your parents are Republicans, and you choose to align yourself with your parents' ideology, you're a Republican. If they later choose to become Communists, this doesn't automatically change your ideology; you're presumably still a Republican until you explicitly change party alignments. There's no way to convert an entire family of registered voters to Communism without explicitly re-registering each member of the family.

Stated another way, when creating compositional entity trees, modelers should take pains to align the parent (container entity) before constructing and then aligning its children (contained entities). A common way to facilitate this ordering is to pass a reference to the parent into the constructor of the children, so that they can align themselves to it accordingly before creating their own children. It's the modeler's responsibility.

Static alignments will always return `now()`; that is, they become effective immediately. Dynamic alignments may return `now()`, but more frequently will return an implementation-dependent future timestamp (as with `mapto();` see section 8.1). Modelers must take this into account (e.g., by waiting for the time to arrive) before making use of the new alignment.

7. Processes

7.1. Construction and Initialization Allowing a C++ process to borrow an Entity method for its body makes it easier to express process behavior cleanly. Typically, a process refers to and acts upon the state of the owner entity to carry out its behavior. In Java, processes constructed as anonymous inner classes of entities can refer transparently to the members and methods of the enclosing class. In C++, borrowing an Entity method as a process body gives the same effect without requiring the modeler to write `owner()->foo` everywhere access to the owner's state is required.

7.3. Simple Processes The `isSimple()` method is an advisory mechanism that allows the author of a process to assert that `action()` returns control to the framework immediately after any wait; significant performance improvements can result, as the body of such a process may be executed on the simulator stack rather than in its own dedicated thread. Not all implementations will provide this optimization.

Asserting simplicity incorrectly will result in undefined behavior, including simulation deadlock. It can be detected and flagged as an error (two successive wait-statements executed sequentially in the same execution of `action()` signals the incorrect assertion) but only after an arbitrarily long interval of time has elapsed. This error is therefore nonrecoverable; SSF should crash. Users are encouraged to debug code without making this assertion, and then use it selectively to improve performance.

8. inChannels, outChannels

8.1. Mapping Channels As with entities' `alignto()` method, mappings created before the simulation starts will always return `now()`; that is, they become effective immediately. Dynamic mappings will usually return a future timestamp, forcing the modeler to wait for that time to arrive before making use of the new mapping.

8.3. Receiving Events An SSF implementation may wish to warn the modeler when events disappear for want of an interested receiving process. This is sometimes done by design, but is more often unintended and can be hard to debug.

8.4. Unmapping Channels SSF implementations may wish to warn modelers about writes to and reads from unmapped channels. Once again, this is unintended more often than not, and may signal a hard-to-diagnose error condition; for example, in a protocol implementation.

Appendix B. SSF API

A.1. Java Binding

```
public interface Event {
  public Event save();
  public void release();
  public boolean aliased();
}

public interface Entity {
  public ltime_t now();
  public void startAll(ltime_t t1);
  public void startAll(ltime_t t0, ltime_t t1);
  public ltime_t pauseAll();
  public void resumeAll();
  public void joinAll();
  public ltime_t alignto(Entity s);
  public Object alignment();
  public java.util.Vector coalignedEntities();
  public void init();
  public java.util.Vector processes();
  public java.util.Vector inChannels();
  public java.util.Vector outChannels();
}

public interface process {
  public Entity owner();
  public void action();
  public void init();
  public void waitOn(inChannel[] waitchannels);
  public void waitOn(inChannel waitchannel);
  public void waitForever();
  public void waitFor(ltime_t waitinterval);
  public boolean waitOnFor(inChannel[] waitchannels, ltime_t timeout);
  public boolean isSimple();
}

public interface outChannel {
 public Entity owner();
 public inChannel[] mappedto();
 public void write(Event evt, ltime_t delay);
 public void write(Event evt);
 public ltime_t mapto(inChannel tgt);
 public ltime_t mapto(inChannel tgt, ltime_t mapping_delay);
 public ltime_t unmap(inChannel tgt);
}
```

```
public interface inChannel {
  public Entity owner();
  public Event[] activeEvents();
  public outChannel[] mappedto();
}
```

A.2. C++ Binding

```
#define Object void

class Event {
public:
  Event* save();
  void release();
  boolean aliased();
};

class Entity {
public:
  virtual void init() =0;
  ltime_t now();
  void startAll(ltime_t t1);
  void startAll(ltime_t t0, ltime_t t1);
  ltime_t pauseAll();
  void resumeAll();
  void joinAll();
  Object* alignment();
  ltime_t alignto(Entity* s);
  Entity** coalignedEntities();           // null-terminated
  process** processes();                  // null-terminated
  inChannel** inChannels();               // null-terminated
  outChannel** outChannels();             // null-terminated
};

class process {
public:
  process(Entity* theowner, int simple=0);
  process(Entity* theowner, void(Entity::*body)(process*), int simple=0);
  virtual void action();
  virtual void init();
  virtual boolean isSimple();
  Entity* owner();
  void waitOn(inChannel** waitchannels);
  void waitOn(inChannel* waitchannel);
  void waitForever();
  void waitFor(ltime_t waitinterval);
```

```
  boolean waitOnFor(inChannel** waitchannels, ltime_t timeout);
};
class outChannel {
public:
  Entity* owner();
  inChannel** mappedto();                   // null-terminated
  void write(Event* evt, ltime_t delay =0);
  ltime_t mapto(inChannel* tgt, ltime_t mapping_delay =0);
  ltime_t unmap(inChannel* tgt);
};
class inChannel {
public:
  Entity* owner();
  Event** activeEvents();                   // null-terminated
  outChannel** mappedto();                  // null-terminated
};
```

Appendix C. Sequential and Simple Sequential SSF

The Scalable Simulation Framework API was designed with high performance implementation in mind; that is, scalability to very large problem sizes by utilization of parallel processing resources. At the same time, the SSF directives that control model concurrency (`alignment(), alignto(), makeIndependent()`) can be treated as pure annotations in a single-processor environment. That is, an SSF implementation can ignore these statements altogether while maintaining compliance. Such an implementation is said to implement *Sequential SSF*.

This appendix reinterprets the general SSF specification in light of the simpler requirements of a sequential SSF implementation, for the convenience of implementors and modelers.

C.1. Sequential SSF

In sequential SSF, a model contains exactly one alignment group, which serves as the default alignment target for all entities. The `alignto()` and `makeIndependent()` methods of Entity have no effect and return a timestamp equal to `now()`. Similarly, the `mapto()` and `unmap()` operations will always take effect immediately and return a timestamp equal to `now()`.

C.2. Simple Sequential SSF

For highest sequential performance, sequential SSF implementations may additionally mandate the exclusive use of simple processes (7.3). Such an implementation is said to implement *Simple Sequential SSF*.

C.3. Anticipating Parallelism

Any process in a sequential SSF model may freely access and modify any accessible part of the model's state. However, modelers must recognize that such direct access can make future parallelization significantly more difficult.

Sending and receiving events on channels is a higher-overhead form of interaction among processes, but lends itself better to subsequent parallelization of the model. Avoiding the use of channels with zero inherent minimal delay is another strategy that anticipates future parallelism.

Finally, sequential SSF models shouldn't ignore the timestamps returned by `alignto(), makeIndependent(), mapto(),` and `unmap()`. Correct execution of the model in the presence of future parallelism will require that the modeler wait until the effective timestamp to make use of the resultant alignment or channel mapping. As a low-overhead rule of thumb, have model code verify that the return timestamp is equal to `now()` — the failed assertion will flag a component that was designed for sequential operation but used in a parallel context.

Appendix D. Summary of Implementation Dependencies

1. Results of more than one call to `startAll()` (4.1)
2. Exact order and concurrency of entity/process initialization (4.2)
3. Timestamp returned by pauseAll() (4.5)
4. Order of resumption of simultaneously eligible processes (4.6)
5. Default entity alignment strategy (6.2)
6. Exact type of the object returned by alignment (6.2)
7. Maximum number of distinct alignments (6.2)
8. Timestamp returned by dynamic realignment (6.3)
9. Maximum multicast channel mapping count (8.1)
10. Fate of undelivered events on an unmapped channel (8.4)