Parallel Real-time Immersive Modeling Environment (PRIME)

Scalable Simulation Framework (SSF)

VERSION 1.0

USER'S MANUAL

JASON LIU

Colorado School of Mines
Department of Mathematical and Computer Sciences
Golden, CO 80401

June 9, 2006

DISCLAIMER

ABOUT THIS MANUAL

This manual is distributed along with the PRIME SSF software. An online version of this manual can be obtained from the PRIME web site

`http://www.mines.edu/Academic/macs/research/prime/`

This manual is modified from the DaSSF user’s manual. We rearranged the content for better exposition and we added descriptions of new functionalities that do not exist in DaSSF. This user’s manual explains the use of PRIME SSF version 1.0. Additional information about this software can also be found at the above web site. A reference manual accompanying this document contains detailed descriptions of the PRIME SSF classes, type definitions, and function prototypes. The reference manual is also distributed with the software and can also be obtained from the PRIME web site.

ACKNOWLEDGMENT

We wish to thank Professor David M. Nicol for his vision on the DaSSF Project since 1998 when he was at the Dartmouth College. The project evolved over the years and later renamed to iSSF when Professor Nicol moved to the University of Illinois at Urbana-Champaign. We aspire to continue this line of research and providing high-quality high-performance simulation software to the research community.

Contents

1 Introduction

Parallel Real-time Immersive Modeling Environment (PRIME) is an ongoing research project at Colorado School of Mines. The goal of this project is to investigate high-performance simulation techniques—including parallel and distributed discrete-event simulation, real-time simulation and emulation, and alternative efficient multi-resolution modeling techniques—to achieve simulations of large-scale complex systems, in particular, the global computer network. The Scalable Simulation Framework (SSF) is a standard object-oriented application programming interface (API) for parallel simulation of potentially large and complex systems. SSF specifies a concise yet powerful API that is compact, portable, and transparently parallelizable on multiple computer architectures. PRIME SSF is an C++ implementation of the SSF API. PRIME SSF is developed based on our previous DaSSF and iSSF implementations, which feature a highly efficient, conservatively synchronized parallel simulation kernel supporting high-performance simulation of large-scale systems.

1.1 Scalable Simulation Framework

Scalable Simulation Framework is the brainchild of researchers in the parallel simulation and networking research fields aiming to create a scalable self-organizing simulation infrastructure capable of simulating the global Internet. Simulating the global Internet is hindered by many problems. The Internet is described by Paxson and Floyd as "an immense moving target", primarily due to its large size, heterogeneity, and rapid change. To overcome these difficulties, researchers have identified some important simulation strategies:

- Using parallel simulation to improve the scalability of the simulation and overcome limitations imposed by traditional sequential simulation in both memory space and computational complexity.
- Providing a lean and self-configurable simulation API to ease the task of maintaining large and complex simulation models.
- Making the simulator portable across different programming languages and machine platforms in order to maximize the potential for model reuse by minimizing the dependencies on a particular simulator implementation or a particular hardware platform.

The design of SSF followed these guidelines. The SSF API contains only five main base classes that can be used for building a complex simulation model. Their functions can be described briefly as follows:

- `Entity` is the base class that represents a simulation entity. A simulation entity is a container for simulation state variables. For example, in a network simulation, hosts and routers are typically modeled as entities. Each entity contains variables that describe the state of a router or a host, such as the size of the output queue at the network interface or the remaining time before a packet retransmission. In SSF, an entity object *owns* instances of the `Process`, `inChannel`, and `outChannel` classes, which we describe below. They are part of the entity's state. It is expected that user creates a class derived from the `Entity` base class containing user-defined state variables, including processes and channels.
- `Process` is considered as a part of an entity that specifies the evolution of the entity's state. SSF adopts a process-oriented simulation world-view. Each simulation process in SSF is represented by an instance of a `Process` class or a class derived from the `Process` base class. One can think of these processes as traditional Unix processes: they are independent threads of control. A process can be blocked waiting for a message to arrive at one or any one of a set of specified input channels of the entity to which the process belongs. A process can also be blocked for a period of simulation time to elapse.

- `inChannel` represents the receiving end of a directed communication link between entities. In SSF, communication between entities is achieved by message passing. An entity can only receive messages from another entity through an input channel. An `inChannel` object belongs to an entity and the ownership is immutable. That is, one cannot change the owner entity of an input channel once it has been created.

- `outChannel` is the starting point of a communication link between entities. Similar to the `inChannel` object, an `outChannel` object must belong to an entity. An output channel of an entity can be mapped to multiple input channels that belong to this or other entities. A message sent to the output channel will delivered by the simulation runtime system to all corresponding input channels that are mapped to the output channel.

- `Event` is the base class that represents messages sent between entities through the communication channels. The user is expected to define his or her own event classes that are derived from this `Event` base class. In this case, the user is able to include specific information that needs to be passed between entities.

SSF API is truly generic for systems that can be modeled as a collection of objects that communicate via message passing. This type of simulation model can be automatically mapped to multiple processors for parallel processing. For example, SSF has been proven to be very effective in modeling telecommunication networks.

1.2 PRIME SSF

PRIME SSF is a high-performance C++ implementation of the Scalable Simulation Framework. PRIME SSF is a direct descendent of the DaSSF implementation, which is designed for large-scale models on parallel platforms. The runtime system uses efficient parallel conservative simulation synchronization algorithms to achieve good parallel performance. Throughout the design and implementation, we strive to conserve memory usage in the simulation model, since memory consumption is one of the biggest problems we need to address when dealing with large-scale models. The standard SSF API is also augmented with functionalities, such as support for the event-orient simulation world-view in addition to the process-oriented approach for better simulation efficiency, and the real-time simulation support for interactions with real applications. In principle PRIME SSF can run on almost all UNIX-based machines. Parallelization is fully supported on most of these platforms. PRIME SSF is capable of running simulation on either shared-memory multiprocessors, or distributed-memory machine clusters, or a combination of both.

Following DaSSF's design, PRIME SSF uses a handcrafted multithreading mechanism to support process-oriented simulation in order to maintain good memory efficiency. We chose to avoid using a general multithreading library (such as pthreads) primarily for two reasons. First, the context switching cost of a general multithreading library is often too high to be used in a simulation environment. To maintain good performance, the overhead for scheduling simulation processes must be kept very low, since process context switches happen frequently—presumably once for each event that is processed. The cost of switching between system threads would become prohibitive in this scenario. Second, the memory footprint of a thread in simulation must also be kept as small as possible. This is because a large-scale simulation may consist of millions of simulation processes. In a general multithreading library, the system preallocates memory for the stack of each thread, and typically the size of the stack is set to be large enough for the worst-case scenario. This creates a serious problem in term of memory consumption, even for a moderate sized model that has merely hundreds of threads.

To support the handcrafted multithreading, PRIME SSF (as in DaSSF) introduces source-code translation. The original model written in C++ is embedded with annotations—special comments that are used

to describe state variables and function invocations in a process. A source-to-source translator is used to automatically transform these annotations into embedded C++ instructions that carry out thread scheduling. These handcrafted threads, as opposed to system threads, are created in the program heap rather than the stack. A context switch between these threads is therefore implemented as function calls without the involvement of the operating system, which can be executed much more efficiently. PRIME SSF multithreading is highly efficient. But the efficiency comes at a price. We feel that the user needs to be informed of the complications introduced by this and other design decisions of PRIME SSF as well as their effects on the model development. For example,

- A C++ SSF implementation is definitely faster than a Java SSF implementation. However, Java provides an automatic garbage collection mechanism, which will be especially helpful for a naive programmer. A C++ SSF implementation, like PRIME SSF, imposes strict and meticulous rules in the use of memory in the model. Violations to these rules may result in unfriendly program crashes that require time and expertise to debug the model. On the other hand, this can be also regarded as an advantage for a C++ SSF implementation. C++ is still much faster than Java, even with its garbage collection turned off. A judicious use of memory results in high efficiency, which is especially critical for large-scale simulations.

- PRIME SSF, following the design of its predecessors, relies on user's annotations for source-to-source translation. C++ language is a very complex and, to some degree, not well standardized. One would have to take a tremendous effort to develop a fully-fledged C++ source-code parser that works on various machine architectures and operating systems. This is a path we did not take. Instead, we developed a simple source-to-source translator in Perl. In this case, the programmer must carry the burden of providing information to the translator for it to instrument the original C++ source code. In particular, the user must identify those places in the source code that may cause a potential context switch. With some extra care, we believe that in most circumstances the added complication is insignificant. In practice, the extra care is paid off quite well with a significant performance gain.

2 Installation

2.1 Supported Platforms

PRIME SSF is capable of running on a variety of architectures. It can be configured to run either as a standalone program, in which case parallelization is supported using shared memory on multiprocessor platforms (including sequential simulation if the machine only has a single processor), or as a distributed program, where communication and synchronization are supported via message passing. Note that the distributed simulation in SSF can be regarded as a collection of SSF simulators each running on multiprocessor machine in a cluster of machines that communicate using message passing. The following is a list of machine platforms we have run PRIME SSF:

- Intel/AMD x86 running Linux.
- PowerPC running OS X (Darwin).

We expect PRIME SSF able to run on most UNIX-based platforms. For those platforms not listed here, there is a good chance that, with a minimum effort, we can make PRIME SSF run on those machines.

2.2 System Requirements

The following tools and utilities are either required or recommended in order to enable full functionalities of PRIME SSF:

- GNU C/C++ compiler (`gcc` and `g++`), although other compilers (such as Intel C/C++ compilers) are also possible candidates.
- Practical Extraction and Report Language (`perl`).
- GNU `make` is recommended.
- GNU `flex` (the lexical analyzer) and `bison` (the parser) are recommended.
- ar and ranlib (archiving tools).
- The Message Passing Interface (MPI) is recommended for distributed simulation.
- GNU `automake` and `autoconf` are required for developers of PRIME SSF.

2.3 Configuring SSF Runtime System

PRIME SSF is distributed with open source. The distribution comes with a compressed tar file (with a `.tgz` extension). You need to first uncompress and extract source code from this compressed tar file:

```
% tar xzvf prime-1.0.0.tgz
```

This command will create a new directory called `prime` (which we hereby refer to as the PRIME root directory) and put all extracted files into this directory. The files are organized into sub-directories according to their functions. A brief description of the files included in the distribution is provided in the README file at the PRIME root directory.

PRIME SSF uses GNU `automake` and `autoconf` to configure the runtime system automatically. The user only needs to invoke the `configure` script produced by `autoconf` to configure the runtime system and then use the `Makefile` produced by `automake` and `autoconf` to build the system. For default configuration, the user only needs to run the following command at the PRIME root directory:

```
% ./configure
```

This script first checks the availability of the system tools required to run PRIME SSF on this platform. Errors will be printed out if the script detects that the system requirement is not met. If no error is encountered, the `configure` script attempts to find out correct values for various system-dependent variables that will be used by compilation. Customized configuration can be performed by supplying parameters to the configuration script. For example, if the user wants to enable SSF debugging information and runtime checking to make sure the simulation is running correctly, he or she can run the script as follows:

```
% ./configure --enable-ssf-debug --enable-ssf-scrutiny
```

It is important to know that if one decides to reconfigure the system with a new set of options, the PRIME SSF runtime system must be cleaned up first (using `make clean`), reconfigured (using the `configure` script), and then rebuilt (using `make`). There is a list of options that the user can use to configure the PRIME SSF runtime system. We now describe these options in more detail.

- `--enable-metis/--disable-metis`: indicates whether graph partitioning is supported. METIS is a graph partitioning tool written by George Karypis at the University of Minnesota. PRIME SSF uses METIS to partition simulation models described in the SSF DML model specification (see section 6). If the option is enabled, the system will build the METIS library in the `metis` sub-directory under the PRIME root directory. If omitted, the default is `--enable-metis`.

- `--enable-serial/disable-serial`: indicates whether the Boost serialization support is included. PRIME SSF does not use the Boost serialization library, but the user is recommended to use the library if serialization is seriously in need when writing simulation models. The library is written by Robert Ramey and is distributed as part of the Boost library. If the option is enabled, the system will build the serialization library in the `serial` sub-directory under the PRIME root directory. The default is `--disable-serial`.

- `--enable-dml-debug/--disable-dml-debug`: indicates whether debugging information is needed for the DML library. DML stands for Domain Modeling Language and it has been used extensively in SSF to configure simulation models. The DML library is part of the SSF runtime system. This option, if enabled, will allow the library to print out debugging information. Thus, the option is most useful for someone debugging the DML library. The default is `--disable-dml-debug`.

- `--enable-dml-locinfo/--disable-dml-locinfo`: indicates whether to keep track of the textual locations of the DML symbols when parsing a DML file. The location information, including the file name and the line number of each DML symbol, is useful to a user who wants to locate syntactic errors in a DML file. This option is useful for someone in the process of developing a model using DML. However, the location information may consume significant memory space. In cases where one has already settled with the DML model and simply wants to use the model for big simulations, the location information can be excluded for the sake of conserving memory. The default is `--enable-dml-locinfo`.

- `--enable-rng-debug/--disable-rng-debug`: indicates whether debugging information is needed for the RNG library. The RNG library contains a long list of random number generators, ported from the original DaSSF implementation, from the Boost random library, and from the SPRNG library (developed at the Florida State University and distributed by the National Center for Supercomputing Applications). The RNG library is part of the SSF runtime system. This option, if enabled, will allow the library to print out necessary debugging information. The option is most useful for someone debugging the RNG library. The default is `--disable-rng-debug`.

- `--enable-ssf-debug/--disable-ssf-debug`: indicates whether debugging information is needed for the SSF runtime system. At the time when a model is first built, the user may want to enable this option so that the user will be able to trace the program using a debugger like `gdb`. Also, if the debug switch is enabled, it is possible to let the SSF runtime system to print out useful information during execution using the `-debug` command-line option. This command-line option sets a mask to filter out unwanted print-out messages, which could be otherwise quite verbose. When the model has been fully tested, the user may want to disable this option and thereby improve the performance of the simulator. The default option is `--disable-ssf-debug`.

- `--enable-ssf-scrutiny/--disable-ssf-scrutiny`: specifies whether the runtime system should check the correctness of the parameters provided by the user when the SSF interface functions are invoked by the user. That is, if this option is enabled, the system must make sure, if possible, that the arguments used to invoke the SSF API methods are as expected according to the SSF specification. This runtime checking could be very time consuming. At the time when a model is first built, the user may want to turn this switch on so that errors can be caught quickly when testing the correctness of the model. When the model has been tested free of these obvious problems, the user can disable this option for better simulation performance. The default is `--disable-ssf-scrutiny`.

- `--with-ssf-ltime=[float|double|long|longlong]`: specifies the type of the simulation time used by PRIME SSF. In SSF, simulation time is defined as type `ltime_t`. This option spec-

ifies what will be the actual type used in simulation, which can be single-precision float-point numbers (`float`), double-precision floating-point numbers (`double`), long integers (`long`), or long long integers (of type `long long` in C and C++). The default of is `--with-ssf-ltime=longlong`.

- `--with-ssf-sync=[standalone|mpi|libsynk]`: specifies the synchronization method used by PRIME SSF for distributed simulation. If the `standalone` option is chosen, the system does not support distributed simulation, although the simulation can still run on multiple processors via shared memory. If the `mpi` option is chosen, PRIME SSF uses the Message Passing Interface (MPI) for communication between remote processors. In this case, the user needs to specify the MPI compilers when configuring the system (using `--with-mpicc` and `--with-mpicxx` options), unless the MPI library uses the default names for C and C++ compilers (i.e., `mpicc` and `mpicxx`). If the user chooses the `libsynk` option, PRIME SSF uses the libSynk library, developed by Kalyan Perumalla at Georgia Tech, to communicate and synchronize SSF instances running on distributed-memory machines. The default option is `--with-ssf-sync=standalone`.

- `--with-ssf-threads/--without-ssf-threads`: specifies whether PRIME SSF should be forced to use threads for shared-memory parallelism. This option only applies to certain platforms. In PRIME SSF, each processor within a shared-memory multiprocessor machine is represented by either a Unix process or a thread. This option specifies which one to use. Unless the user is a seasoned developer of PRIME SSF, it is recommended that the user uses the default, which is `--without-ssf-threads`.

- `--enable-ssf-quickmem/disable-ssf-quickmem`: specifies whether to enable the SSF quick-memory service. Quick memory is a memory management layer above shared memory, which is used for communicating simulation processes on a local machine. Quick memory provides fast memory allocation and deallocation services at each processor at the cost of additional memory overhead due to fragmentation. If you are not sure whether to use quick memory, choose the default, which is `--enable-ssf-quickmem`.

- `--enable-ssf-hoard/--disable-ssf-hoard`: specifies the shared-memory management layer. Hoard, developed by Emery Burger at the University of Texas at Austin, is an efficient memory allocator for shared-memory multiprocessors. If you are not sure whether to use quick memory, choose the default, which is `--enable-ssf-hoard`.

- `--with-mpicc=`*`mpi-c-compiler`* and `--with-mpicxx=`*`mpi-c++-compiler`*: specify the names of the C and C++ compilers for the MPI library. This option is applicable only when `--with-ssf-sync=mpi` is used. By default, the configuration script will look for standard names of the MPI compilers, such as `mpicc` and `mpicxx`. However, on some systems, the names of the MPI compilers are difficult to guess. In this case, the user must provide the names to the configuration script using these options.

- `--with-metis-prefix=`*`libmetis-dir`*, `--with-serial-prefix=`*`libserial-dir`*, `--with-dml-prefix=`*`libdml-dir`*, and `--with-rng-prefix=`*`librng-dir`*: specify the location of the associated libraries. These options are used only when the SSF runtime system is detached from these libraries. They are rarely used.

2.4 Building SSF Runtime System

After configuration, the runtime system can be built easily with the `make` command at the PRIME root directory. A library (`libssf.a`) will be create as the result. After building the runtime system, you can also compile all the examples bundled with this distribution. The steps are shown as follows:

```
% make > log 2>&1
% cd examples
% make
```

Whenever the user decides to change the configuration of the system, the runtime system must be rebuilt. For example, if the user decides to turn off the debugging information and runtime checking for better simulation performance once the model has been tested, the following steps ought to be taken (in the PRIME root directory). Note that the arguments to the configuration script is actually optional in this case since they are the default behavior.

```
% make clean
% ./configure --disable-ssf-debug --disable-ssf-scrutiny
% make
```

2.5 Command-Line Options

Once the executable file of a simulation model is created successfully, the model is ready to run. There are standard command-line arguments that can be used to change the behavior of the simulation runtime system. Some of these command-line options are universal for all SSF configurations, while others only apply to a certain type of configuration. Parsing the command-line arguments happens before the user's main function in the model is invoked. These command-line arguments will be removed from the argument list before the main function is called. To prevent the SSF runtime system to parse certain command-line options that are used by a simulation model, the users can put a double dash (`--`) to stop the runtime system from parsing the rest of the command-line arguments. All possible command-line arguments that can be recognized by the SSF runtime system are listed here:

- `-submodel` <*filename*>: the user can use this option to provide the name of the DML file that contains the preprocessed SSF DML model description (see section 6). This command-line argument is used when the user wants to use the simulation kernel to partition and build the model from SSF DML model description rather than using the main function. A shorthand of this option is `-s`.

- `-nmachs` $< m$ `[`n_0`,]`p_0`:[`n_1`,]`p_1`:`$\cdots$`:[`n_{m-1}`,]`$p_{m-1} >$: this argument is used to provide the hardware profile for the simulation run. It specifies the number of distributed-memory machines (i.e., m) that the user wants to use for the simulation run. If the user chooses to run simulation using MPI, this number must match the number of nodes used for starting the MPI program (i.e., the parameter provided for the `mpirun -np` option). If the user runs the simulation in a stand-alone mode, this number must be one. This option also specifies the number of processors on every distributed node: p_i is the number of processors on node i. If the user chooses to use the libSynk library for global communication and synchronization, the names of the distributed machines must also be specified: n_i is the name of node i. The name of a node can either be the name of the machine or an IP address. A shorthand of this option is `-m`.

- `-nprocs` <*nprocs*>: this argument is used to specify the number of processors on the current machine—the only machine used for this simulation run. The option is the same as `-m 1` *`nprocs`*. That is, the option only applies to the stand-alone mode or when the number of machines used by distributed simulation is one. A shorthand of this option is `-n`.

- `-rank` <*rank*>: this argument specifies the index of the current machine that starts the simulation. This option is only applicable if the user chooses to use the libSynk library for global communication and synchronization. The rank must be between 0 and $m - 1$, where m is the total number of distributed machines that are used by simulation. A shorthand of this option is `-r`.

- `-seed` <*random-seed*>: this option sets the system-wide random seed of all random streams used in SSF, both by the runtime system and by the user's simulation model. It is guaranteed that simulation will be repeatable if using the same random seed, provided that the random variables are correctly seeded (with non-zero seed) in the model. Issues about simulation repeatability are discussed in detail in section 7.1.

- `-heap` <*heapsize*>: by default, the runtime system will pre-allocate memory to be used as the program heap *on each processor*. The program heap is used for dynamic memory allocation and deallocation. In models that require a large amount of memory space, the size of the pre-allocated memory may not be enough. This argument specifies the size of the program heap per processor (in MB) needed by the application.

- `-backstore` <*dirname*>: the shared data segments are mapped into files (using the `mmap` system call). This option specifies the directory where these files are created. This option is particularly useful on some systems where certain directories are mounted as in-memory file systems. The default is `/tmp`.

- `-silent`: this argument will prevent the runtime system from printing out simulation runtime information, which could be sometimes quite distracting.

- `-outfile` <*filename*>: this option tells the runtime system to pipe the output to the designated file. By default, messages are printed to the standard output. A shorthand of this option is `-o`.

- `-debug` <*mask*>: this option applies only when system debugging information is enabled at configuration. The debug mask is used to filter out messages which are not considered interesting by the user. This option is used internally by PRIME SSF developers. A shorthand of this option is `-d`.

- `-delta` <*delta*>: this option is used to impose flow control for scheduling logical processes. The argument specifies the simulation time interval at which a logical process must reschedule itself. This flow control mechanism is experimental. The default value is ∞, which is equivalent to turning this mechanism off.

- `-flowmark` <*memsize*>: this is another flow control mechanism for scheduling logical processes. The argument specifies the maximum number of bytes that can be used by new kernel events in a single execution session of a logical process before it must be rescheduled. This flow control mechanism is experimental. The default value is ∞, which is equivalent to turning this mechanism off.

- `-thresh` <*threshold*>: this option is used by the composite synchronization protocol to partition channels between logical processes as either synchronous or asynchronous. This argument is not recommended to normal SSF users. If the option is omitted, the kernel will automatically classify channels using runtime measurements. A shorthand of this option is `-t`.

- `-progress` <*interval*>: this option is used to tell the runtime system to show the progress of the simulation run. A message will be printed by the runtime system every *interval* amount of simulation time to indicate the simulation is running normally. The option is useful for a simulation that takes a long time to run. A shorthand of this option is `-p`.

3 Getting Started

In this section, we use an example to show how to develop a simulation model in PRIME SSF. We use this example to describe different stages involved in developing and running a simulation model.

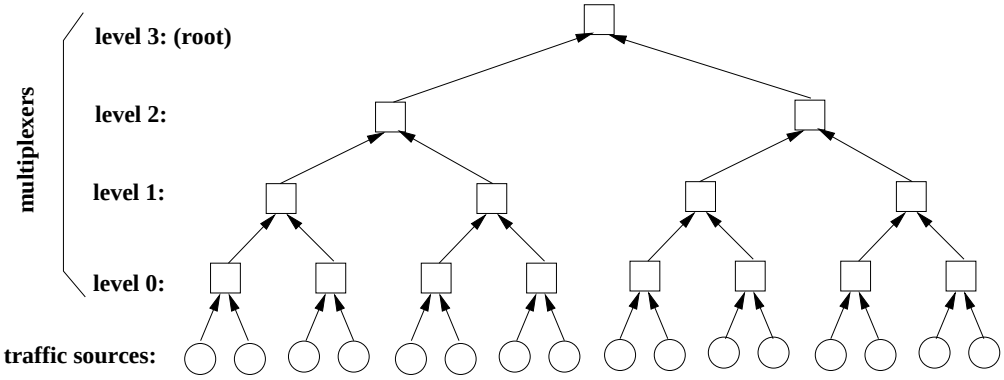

Figure 1: Topology of the *muxtree* model.

3.1 The *muxtree* Example

The *muxtree* model is defined in a header file `muxtree.h` and a source file `muxtree.cc`, both located at the `examples/muxtree` directory under PRIME root directory. We use a `Makefile` to create the final executable file.

The *muxtree* model represents a simplistic network of multiplexer switches. The network topology is a tree, where the leaves of the tree are traffic sources. Packets are generated from these traffic sources. They travel through the switches and flow to the root of the tree. At each multiplexer, the packets are buffered in a FIFO queue before entering service. Each packet in service waits for some constant service time before it is sent to the multiplexer at the next level until it reaches the root. Packets may be dropped if a buffer overflow occurs.

An example topology of the model is shown in figure 1. Traffic is flowing upwards. There are two parameters that control the topology of the tree: the number of input links to each multiplexer switch (`fan-in`) and the height of the multiplexer tree (`levels`). In the example shown in the figure, the network consists of 4 levels of multiplexers, each having a fan-in of 2. These parameters are compile-time constants.

3.2 The Model

In SSF, we use two types of entities to represent the traffic sources and the multiplexers, respectively. An entity is a container for state variables. For example, the entity that represents a multiplexer will contain variables that describe the state of the FIFO queue. Once the entities are defined, we need to set up the connections between these entities so that these entities can communicate with one another. This is achieved by mapping an output channel of an entity to an input channel of another entity. Packets generated by the traffic source entities are represented as events in SSF. Events are sent from the traffic source entities to the multiplexer entities until they reach the root of the tree.

3.2.1 Source Entities

A traffic source is modeled by the `SourceEntity` class, which is derived from the `Entity` base class. A traffic source generates packets with an inter-arrival time according to an exponential distribution and sends the packets out from its output channel. The output channel is mapped to the input channel of another entity that represents a multiplexer at the next tree level. The definition of the `SourceEntity` class is as follows:

```
class SourceEntity : public Entity {
 public:
  int id;     // identity of the source
```

```
    long nsent; // (statistics) number of packets generated
    Random rng; // a random number generator

    outChannel* oc; // output channel connecting to a multiplexer

    SourceEntity(int myid);
  };
```

All state variables related to the traffic source entity must be defined in the class. In the `SourceEntity` class, the variables include an integer to identify the source entity (`id`), a long integer to count the number of packets that has been generated by this source entity (`nsent`), a pseudo random number generator for sampling packet inter-arrival times (`rng`), and an output channel owned by this source entity from which packets are sent (`oc`). Pseudo random number generators in SSF are represented as instances of the `Random` class, which contains the state information for it to generate a sequence of numbers that is expected to be random and reproducible if given the same random seed. Output channels, as well as input channels, are considered as part of the state of the entity. An output channel is the starting point of a communication link between entities. An output channel can map to an arbitrary number of input channels (including zero). An event sent to an output channel will be delivered to all mapped input channels.

The `SourceEntity` class defines only one method—the constructor of this class, the source code of which is listed as follows:

```
SourceEntity::SourceEntity(int myid) :
  id(myid), nsent(0),
  rng(Random::USE_RNG_LEHMER, myid+1)
{
  oc = new outChannel(this, TRANSMISSION_DELAY);

  // map output channel to input channel of a multiplexer
  char icname[128];
  sprintf(icname, "IN_0_%d", id/MUXTREE_FANIN);
  oc->mapto(icname);

  // create a process to generate packets
  new EmitProcess(this);
}
```

The constructor initializes the entity's state variables by setting the identifier of this new source object, resetting the count used for statistics, and initializing the pseudo random number generator. The constructor for the pseudo random number generator takes two arguments. The first argument is the type of the random number generator. The RNG library used by PRIME SSF consists of more than 40 different random number generators with different randomness properties and resource requirements. The example uses the Lehmer random number generator. The second argument is the initial seed for the random number generator. To make each traffic source behaves different from one another, the seed should be different. We use the source entity id (plus one) as the seed in this example.

The constructor also creates an output channel and map it to another input channel of a given name. The output channel is created with a delay (using the constant `TRANSMISSION_DELAY`), which means that all events sent from this output channel will not be delivered until after the specified delay. The name of the input channel that this output channel is mapped to must be unique in the entire model and is chosen carefully so that the source entities connects to the right multiplexer entities (described later) in the tree.

An `EmitProcess` instance is created at the end of the constructor. This will be the process used by the source entity to generate packets. The `EmitProcess` class is defined as follows:

```
class EmitProcess : public Process {
 public:
  // The constructor.
  EmitProcess(SourceEntity* owner) : Process(owner) {}

  // The owner entity of this process is a traffic source entity.
  SourceEntity* owner() { return (SourceEntity*)Process::owner(); }

  // This method is the process's starting procedure.
  virtual void action(); //! SSF PROCEDURE
};
```

The `EmitProcess` class is derived from the `Process` base class. A process in SSF is regarded as part of an entity that specifies the evolution of the entity's state. SSF uses a process-oriented simulation world-view. What it means is that the user should write simulation processes to change the state of the entities. One can think of the processes in a traditional operating system concept. A process is simply a thread of control. In SSF, a process can be blocked either by waiting on an input channel for an event to arrive or by waiting for a specified period of simulation time to pass. Each simulation process is represented as an instance of the `Process` class or a class derived from the `Process` base class.

The `EmitProcess` class defines three methods. The constructor of this class does nothing more than simply invoking the constructor of the `Process` base class. The `EmitProcess` class also overrides the `owner` method, which returns the owner entity of this process and then casts it into the correct source entity type. The `action` method is the starting procedure of this process. The method will be invoked by the runtime system as soon as the process is created.

A special comment `//! SSF PROCEDURE` is placed after the declaration of the `action` method *at the same line*. This special comment among a few others is the kind of annotations required by the SSF source-to-source translator, whose function is to embed the handcrafted multithreading mechanism. Using the handcrafted multithreading mechanism is a design decision for the PRIME SSF to deliver high-performance process-oriented simulation capability. In SSF, we need to mark those methods that are invoked by a simulation process. We call them *procedures*. Inside a procedure, a process may suspend by either waiting on the input channels or waiting for a given simulation time to pass. Process suspension may happen if the procedure invokes a wait statement (which we describe later). The `action` method is by definition the starting procedure of the process and therefore should be marked accordingly.

The `action` method of the `EmitProcess` class is listed as follows:

```
//! SSF PROCEDURE
void EmitProcess::action() {
  ltime_t tm; //! SSF STATE
  for(;;) {
    tm = (ltime_t)owner()->rng.exponential(1.0/SRC_ENTITY_IAT);
    waitFor(tm);

    // create and send out a newly generated packet
    MyMessage* msg = new MyMessage(now(), owner()->id);
    owner()->oc->write(msg);
```

```
      owner()->nsent++;
    }
  }
```

The `action` method will be invoked implicitly by the system once the process starts running after an `EmitProcess` object is created. Since the `action` method is a procedure, SSF requires that the definition of this method must be mark by `//! SSF PROCEDURE` at the previous line. Within the `action` method, the control enters a loop, in which the process waits for an exponentially distributed amount of simulation time before generating a packet event and then sending it out from the output channel. The `waitFor` method is defined in the `Process` base class as a wait statement: the process will be blocked for the specified amount of simulation time. The delay time is obtained from an exponential distribution (using the pseudo random number generator `rng`, which is defined as a state variable in the source entity) with a mean inter-arrival time of `SRC_ENTITY_IAT`, a constant defined somewhere else. This sampled simulation time is stored in a local variable `tm`. In order for the SSF source-to-source translator to embed code that handles process suspension within a procedure, it is required that local variables that span across wait statements are declared as a state variable of this procedure, including those variables that are passed as arguments to the wait statements. That is why the variable is marked with `//! SSF STATE`. It is required that the declaration of state variables must be done at the *beginning* of a procedure and the variables must not be initialized at the time of declaration.

A `MyMessage` object is created that represents the packet being generated. As described later, the `MyMessage` class is derived from the `Event` base class. The constructor of the class takes two arguments: the current simulation time (returned from the `now` method defined by the `Process` base class) and the source entity's id. The packet is sent out from the output channel by calling the write method of the `outChannel` class. Note that the loop is actually unnecessary. In SSF, if the control reaches the end of a starting procedure, the process will restart from the beginning of the procedure.

3.2.2 Packet Events

The packets generated from the source entities at the leaves of the tree will be sent to the multiplexer entities and will finally reach the root of the tree if not dropped at an intermediate node. The packets are represented as an SSF event called `MyMessage` derived from the `Event` base class:

```
class MyMessage : public Event {
 public:
  ltime_t time; // the time when this event is generated
  int srcid;    // id of the source entity that generates the event

  // The constructor.
  MyMessage(ltime_t tm, int sid) : time(tm), srcid(sid) {}

  // The copy constructor and the clone method.
  MyMessage(const MyMessage& msg) :
    time(msg.time), srcid(msg.srcid) {}
  virtual Event* clone() { return new MyMessage(time, srcid); }

  // Pack the event into a compact byte stream.
  virtual ssf_compact* pack() {
    ssf_compact* dp = new ssf_compact;
```

```
    dp->add_ltime(time); // serialize the time
    dp->add_int(srcid);  // serialize the source entity id
    return dp;
  }

  // The factory method of this event object.
  static Event* create_my_message(ssf_compact* dp) {
    ltime_t tm; int sid;
    if(!dp || !dp->get_ltime(&tm) || !dp->get_int(&sid))
      return 0; // return null if an error is encountered
    return new MyMessage(tm, sid);
  }

  // Declare this event class.
  SSF_DECLARE_EVENT(MyMessage);
};
```

Each packet contains two variables: one records the simulation time when this packet is generated and the other is the source entity id. The constructor for the `MyMessage` class sets these variables. In order for the simulation runtime system to automatically ship this event to another processor on shared memory, it is required that the event class must include the copy constructor and the `clone` method. The copy constructor is responsible for copying the member variables of this event class. The `clone` method is invoked by the simulation runtime system to create a new instance of this event class that is identical to the original object.

In addition, in order to ship this event to a remote processor in a distributed-memory environment, two additional methods must be provided. The `pack` method is invoked by the runtime system to convert this event into a byte stream (i.e., serialization). The byte stream is represented in PRIME SSF as an `ssf_compact` class. Methods like `add_ltime` and `add_int` are used to serialize and insert a simulation time and an integer into the byte stream. When the byte stream (or byte array) is delivered to the remote processor, the runtime system will invoke the factory method of this event class to *unpack* this byte stream and create an exact copy of the original event. The factory method can be any static method that takes a pointer to an `ssf_compact` object as the argument and returns a pointer to a generic `Event` object. The factory method uses methods like `get_ltime` and `get_int` to retrieve the corresponding data from the byte stream (i.e., deserialization). Note that the order of serialization must match the order of deserialization. A new `MyMessage` object is then created and returned.

It is important to remember that all event classes (those derived from the `Event` base class) must be declared to the runtime system using the `SSF_DECLARE_EVENT` macro. In addition, each event class must also register its factory method in the source file using the `SSF_REGISTER_EVENT` macro:

```
// Register the event with the associated factory method.
SSF_REGISTER_EVENT(MyMessage, MyMessage::create_my_message);
```

3.2.3 Multiplexer Entities

The multiplexers in this model is represented by the `MultiplexerEntity` class, which is defined as follows:

```
class MultiplexerEntity : public Entity {
 public:
  int level;  // tree level of the multiplexer
```

```
    int id;     // id of the multiplexer (within each tree level)

    long nrcvd; // statistics: total received packets
    long nlost; // statistics: total packets dropped due to overflow
    long nsent; // statistics: total packets passed on

    MyMessage** buf; // buffered messages as a circular list
    int head;   // the head of the buffer
    int tail;   // the tail of the buffer
    int qlen;   // number of packets already in the buffer

    inChannel* ic;  // input channel to receive packets
    outChannel* oc; // output channel to send packets

    inChannel* int_ic;  // internal input channel
    outChannel* int_oc; // internal output channel

    // The constructor and the destructor.
    MultiplexerEntity(int mylevel, int myid);
    virtual ~MultiplexerEntity();

    virtual void init();
    virtual void wrapup();

    // Starting procedure of the arrival process.
    void arrive(Process*); //! SSF PROCEDURE SIMPLE

    // Starting procedure of the service process.
    void serve(Process*);  //! SSF PROCEDURE
  };
```

Same as the `SourceEntity` class, the `MultiplexerEntity` class is also derived from the `Entity` base class. This class contains two integers to identify the position of the multiplexer in the tree: the level and the index id. The class also contains three long integers used for collecting statistics during the simulation. The multiplexer contains a FIFO queue. Packets arrived at the multiplexer will first enter the FIFO queue if the queue is not full. When the previous packet is sent out from the multiplexer, the packet at the head of the queue will enter the server and will be serviced for some constant time before leaving the multiplexer. The state of the queue, including the packets in the queue, is therefore part of the state of the entity. The model uses a circular list to represent the queue (`buf`), with an integer (`head`) storing the index to the packet at the head of the queue and another integer storing the index to the packet at the tail of the queue. The variable `qlen` keeps the number of packets currently in the queue.

Each multiplexer entity has two communication channels: an input channel, which is used to receive packets from either a source entity or a multiplexer entity at the previous tree level, and an output channel, which is used to send packets to the next tree level. Each multiplexer entity also defines two additional (internal) channels to communicate between the arrival process and the service process of this entity, which we describe below. The class has one constructor and one destructor defined. Also, the `MultiplexerEntity` class overrides two methods at the `Entity` base class: the `init` method, which we use to initialize this entity (including creating the source entities if the multiplexer is at level 0) at the start of simulation, and

the `wrapup` method, which we use to wrap up this entity (including writing out the statistics) before the simulation finishes.

There are two processes defined for each multiplexer entity. The arrival process waits for incoming packets from the input channel, enqueues them, and notifies the service process if the queue was empty before the arrival. The service process dequeues the packet at the front of the queue, waits for a fixed amount of simulation time to model the packet being serviced, then sends the packet out from the output channel. The starting procedure for the arrival process is the `arrive` method, which is marked by `//! SSF PROCEDURE SIMPLE`, indicating that the process is a simple process. A simple process is a process containing only one wait statement and the wait statement must be placed at the end of the procedure. This special arrangement allows efficient process context switching using continuation. The starting procedure for the service process is the `serve` method, which is marked by `//! SSF PROCEDURE`.

The constructor of the `MultiplexerEntity` class initializes the state variables, creates the channels, and starts the arrival and service processes:

```
MultiplexerEntity::MultiplexerEntity(int mylevel, int myid) :
  level(mylevel), id(myid),
  nrcvd(-1), // nrcvd marks the first time arrive() is called
  nlost(0), nsent(0),
  tail(0), head(0), qlen(0)
{
  buf = new MyMessage*[MUX_ENTITY_BUFSIZ];

  // create input channel and name it appropriately
  char icname[128];
  sprintf(icname, "IN_%d_%d", level, id);
  ic = new inChannel(icname, this);

  // create output channel and map it accordingly
  oc = new outChannel(this, TRANSMISSION_DELAY);
  if(level < MUXTREE_LEVELS-1) { // except tree root
    sprintf(icname, "IN_%d_%d", level+1, id/MUXTREE_FANIN);
    oc->mapto(icname);
  }

  // internal input channel and output channel, map them together
  int_ic = new inChannel(this);
  int_oc = new outChannel(this);
  int_oc->mapto(int_ic);

  // create the arrival process, which is a simple process
  Process* arrive_proc = new Process
    (this, (void (Entity::*)(Process*))
     &MultiplexerEntity::arrive, true);
  arrive_proc->waitsOn(ic); // set default channel to wait

  // create the service process
  Process* serve_proc = new Process
    (this, (void (Entity::*)(Process*))
```

```
      &MultiplexerEntity::serve);
  serve_proc->waitsOn(int_ic); // set default channel to wait
}
```

The external input channel (`ic`) must have a globally unique name, which is obtained using the tree level and the index id of the multiplexer entity. Output channels of other entities can therefore be mapped to the corresponding input channels of the multiplexer entities as a tree structure. Both the arrival process and the service process use the `Process` base class, rather than creating their own sub-class as in the case of the `EmitProcess` class. The arrival process is a simple process and it is created by passing `true` as the second argument. The arrival process is by default waiting on the external input channel `ic` and similarly the service process is by default waiting on the internal input channel `int_ic`. The default is set by calling the `waitsOn` method defined in the `Process` base class.

The destructor of the `MultiplexerEntity` class needs to reclaim the memory used by the queue. And if there are packets stored in the queue, they need to be reclaimed as well (using the `release` method defined by the `Event` base class). All entities are reclaimed by the runtime system at the end of the simulation. Although reclaiming the memory is actually no longer necessary (since the runtime system automatically takes care of reclaiming all memory blocks at the end of the simulation), it is nonetheless a good programming habit.

```
MultiplexerEntity::~MultiplexerEntity() {
  while(qlen > 0) {
    // get the message at the front of the queue
    MyMessage* msg = buf[(head++)%MUX_ENTITY_BUFSIZ];
    msg->release();
    qlen--;
  }
  delete[] buf;
}
```

The `init` method of an entity is called by the runtime system immediately after the entity is created. We use this opportunity to create the traffic source entities as the leave nodes if the multiplexer entity is at tree level 0 (see figure 1). Similarly, the entity's `wrapup` method is called by the runtime system before the entity is about to be destroyed (which happens only at the very end of the simulation). We use this opportunity to write out the statistics collected at this multiplexer entity. The `init` and the `wrapup` methods are defined as follows:

```
void MultiplexerEntity::init() {
  if(level == 0) {
    for(int i=0; i<MUXTREE_FANIN; i++) {
      SourceEntity* src = new SourceEntity(id*MUXTREE_FANIN+i);
      // align the source entity to this multiplexer entity
      src->alignto(this);
    }
  }
}

void MultiplexerEntity::wrapup() {
  ssf_compact* dp = new ssf_compact;
  dp->add_long(nrcvd);
```

```
    dp->add_long(nlost);
    dp->add_long(nsent);
    dumpData(1, dp); // we use type 1 to identify the dumped record
    delete dp; // don't forget to delete the record after use
  }
```

In the `init` method, the newly created `SourceEntity` objects are immediately aligned to the multiplexer entity that creates them. The `alignto` method is defined in the `Entity` base class and is used to change the current alignment of the entity. Entity alignment is an important concept in SSF. Each entity in SSF has a timeline. Entities sharing the same timeline (in which case we call these entities *coaligned*) are guaranteed to advance their simulation time synchronously. Therefore, these entities can access (i.e., read and write) each other's state variables during the simulation. This is important because, in a parallel simulation, entities in general are running on different processors and therefore processing events at different points in simulation time. Thus accessing a state variable of an entity that is not coaligned may cause a causality error. In our case, the multiplexer entities at tree level 0 and the source entities that connect to them are coaligned, which means the entities can directly access each other's state variables (although we actually choose not to do so in this example).

In the `wrapup` method, we pack the collected statistics into the data structure `ssf_compact`, which represents a platform-independent byte stream. The `dumpData` method is then invoked to write the byte stream out to the default output file generated by the simulator. (Actually, the runtime system has a number of output files, each created for a distributed-memory node). Typically, the user does not need to access the simulation output file. Rather, at the end of the simulation, one can retrieve the entity data records written to the output file and use the data to compute global statistics for the entire simulation run, which we will see in the next section.

```
//! SSF PROCEDURE SIMPLE
void MultiplexerEntity::arrive(Process* p) {
  if(nrcvd >= 0) { // if not the first time
    MyMessage** evts = (MyMessage**)ic->activeEvents();
    for(int i=0; evts[i]; i++) {
      if(qlen != MUX_ENTITY_BUFSIZ) { // if the queue is not full
        // save a new reference of this event into the queue
        buf[(tail++)%MUX_ENTITY_BUFSIZ] = (MyMessage*)evts[i]->save();
        qlen++;

        // unblock the service process upon the first message
        if(qlen == 1) int_oc->write(new Event);
      } else nlost++;
      nrcvd++;
    }
  } else nrcvd++;
  waitOn(); // wait on the default input channel
}
```

The `arrive` method is the starting procedure of the arrival process. All procedures in SSF must be marked by either `//! SSF PROCEDURE` or `//! SSF PROCEDURE SIMPLE` at the previous line where the procedures are defined. The arrival process waits for a packet to arrive at the input channel (`ic`)

and then puts the packet at the tail of the FIFO queue. And if the queue was empty before the newly arrived packet, the arrival process will send a message through the internal output channel (`int_oc`) to the service process, which is blocked on the internal input channel (`int_ic`), which is mapped to the internal output channel. The code in the `arrive` method is arranged in such a way that the `waitOn` method is invoked only at the end of the procedure—a requirement for a simple SSF process. As mentioned earlier, a simple process is a process containing only one wait statement and the wait statement must be placed at the end of the procedure. This special arrangement allows much faster process context switching using continuation.

The first time the `arrive` procedure is called, the execution bypasses the if-statement, since `nrecv` was initialized as -1. The process therefore is blocked immediately waiting on packets to arrive at the default input channel of this process (i.e., `ic`). The main logic of the arrival process is to determine whether there is space for the newly arrived packets in the buffer. Note that there could be more than one packet simultaneously arriving at the input channel. We use the `activeEvents` method to retrieve the packets, which are put into an null-terminated list. If the queue is full the packet is discarded. Otherwise, the packet is inserted into the queue at the tail. A new reference to the packet event is created in this case (by calling `save`). This is necessary so that the event event will not be reclaimed by the runtime system. Should it happen that the arrival is to an empty queue, the process notifies the service process to immediately start servicing the packet. This is done by writing an event out to the internal output channel, which is mapped to the internal input channel on which the service process should be waiting.

```
//! SSF PROCEDURE
void MultiplexerEntity::serve(Process* p) {
  MyMessage* inservice; //! SSF STATE

  waitOn();
  while(qlen > 0) {
    inservice = buf[(head++)%MUX_ENTITY_BUFSIZ];
    qlen--;

    waitFor(MUX_ENTITY_MST);

    // send the serviced message
    oc->write(inservice);

    // release the reference to the event just sent
    inservice->release();
    nsent++;
  }
}
```

The service process first waits on the internal input channel. In this way, whenever there are new packet arrivals, the arrival process will send an event through the internal channel and wake up the service process. For each packet in the queue, the service process removes the packet from the front of the queue, and then simulates the packet switching time by suspending itself for a period of time that equals `MUX_ENTITY_MST` (a constant for the mean service time) before sending the packet out from the output channel. The packet event needs to be dereferenced (using the `release` method) so that the system can reclaim it once it is not used any more.

3.2.4 The Main Function

The main function is used to create the *muxtree* model and starts the simulation. The code is listed in the following:

```
int main(int argc, char** argv) {
  ...
  int nnodes = 1;
  for(int level=MUXTREE_LEVELS-1; level>=0; level--) {
    for(int index=0; index<nnodes; index++) {
      if(index%ssf_num_machines() == ssf_machine_index()) {
        MultiplexerEntity* mux =
          new MultiplexerEntity(level, index);
        mux->makeIndependent(); // make a separate timeline
      }
    }
    nnodes *= MUXTREE_FANIN; // # nodes at the next tree level
  }

  Entity::startAll(start_time, end_time);
  Entity::joinAll();

  return 0;
}
```

The model is created one tree level at a time starting from the root of the tree (at level `MUXTREE_LEVELS-1`). In SSF, the main function is called at each distributed machine that participates the simulation run. SSF adopts the SPMD programming model, or single program multiple data. Accordingly, the model must be built in a distributed fashion. In this example, we only build a portion of the tree that is assigned to the machine that is currently running. Partitioning is achieved by assigning multiplexer entities at each tree level to different machines in a round-robin fashion. We only instantiate those multiplexer entities with indices equal to the index of the machine (returned from the `ssf_machine_index` method) modulus the total number of distributed machines (returned form the `ssf_num_machines` method). We also make each multiplexer entity an independent timeline using the `makeIndependent` method defined in the `Entity` base class.

The `startAll` method is defined as a static function in the `Entity` base class. We call this method to set the start time and the end time of the simulation run. The simulation will execute over this time interval, inclusive of both endpoints. The `start_time` and `end_time` are variables whose value is taken from the command-line argument to the main function. Here we do not show the code for parsing the command-line arguments for brevity. The simulation will run conceptually as an independent process separate from the main function. The main function is expected to wait for the simulation to finish by calling the `joinAll` method before the main function returns.

3.2.5 The Global Wrapup Function

At the end of the simulation, statistics should be gathered and printed out. This can be done either within the main function after `joinAll` returns or using a global wrapup function. A global wrapup function is a callback routine registered by the user (using the `SSF_SET_GLOBAL_WRAPUP`). Once registered, the global wrapup function will be called by the simulation runtime system after the simulation finishes. The function will be called once and only once in the entire distributed simulation environment only on machine 0.

```
static void my_global_wrapup() {
  int x = 0;
  long sum_nrcvd = 0;
  long sum_nlost = 0;
  long sum_nsent = 0;
  Enumeration* dd = Entity::retrieveDataDumped();
  while(dd->hasMoreElements()) { // enumerate each dumped data
    ssf_entity_data* dumprec = (ssf_entity_data*)dd->nextElement();
    if(dumprec->type() != 1) continue;
    ssf_compact* packed = dumprec->data();
    long nrcvd, nlost, nsent;
    if(packed->get_long(&nrcvd) != 1 ||
       packed->get_long(&nlost) != 1 ||
       packed->get_long(&nsent) != 1) {
      fprintf(stderr, "ERROR: unknown statistic record format\n");
      return;
    }
    x++;
    sum_nrcvd += nrcvd;
    sum_nlost += nlost;
    sum_nsent += nsent;
    // both dumprec and packed are not to be reclaimed by user!
  }
  delete dd; // but we should reclaim the enumeration object!

  printf("(%d records) nrcvd=%ld, nlost=%ld, nsent=%ld\n",
  x, sum_nrcvd, sum_nlost, sum_nsent);
}

SSF_SET_GLOBAL_WRAPUP(my_global_wrapup);
```

In this example, the global wrapup function (the `my_global_wrapup` method) is used to collect the statistical data dumped by all multiplexer entities (in their `wrapup` methods). The total number of packets received, lost, and sent for all multiplexer entities are computed and then printed out. We do this by enumerating through the records written out by the multiplexer entities. Each record is represented as an `ssf_entity_data` object, containing the machine-independent byte stream (of type `ssf_compact`), which the modeler used to pack the statistics. The data are retrieved from this byte stream one at a time, which are then summed over all records and finally printed out.

3.3 Building and Running the Model

3.3.1 The Makefile

PRIME SSF does source-to-source translation before the simulation program is compiled and linked. PRIME SSF programs are *almost* written in pure C++. Actually, they are C++ with some annotations embedded as special comments. The source-to-source translator must be used appropriately in this case to transform these annotations into equivalent C++ instructions that implement the multithreading mechanism. To hide the complexity of this procedure, it is recommended that the user organize the source code using a `Makefile`, in which the system can automatically determine what it can do to build a large simulation program from

the header files and the source files. The skeleton of the `Makefile` for the muxtree example is shown in the following:

```
include ../../ssf/config/Makefile.common

MUXTREE_HFILES = muxtree.h
MUXTREE_CFILES = muxtree.cc
MUXTREE_OFILES = $(MUXTREE_CFILES:.cc=.o)

CXXOPT = -g -Wall

MUXTREE_INCLUDES = $(SSF_INCLUDES) -I.
MUXTREE_CFLAGS   = $(SSF_CFLAGS) $(CXXOPT)
MUXTREE_LDFLAGS  = $(SSF_LDFLAGS)
MUXTREE_LDADD    = $(SSF_LDADD)

muxtree: $(MUXTREE_OFILES)
      $(CXX) $(MUXTREE_OFILES) -o $@ $(MUXTREE_LDFLAGS) $(MUXTREE_LDADD)

.cc.o:
      $(CXXCPP) $(MUXTREE_INCLUDES) $(MUXTREE_CFLAGS) -o _tmp0.cc $<
      $(CMTSTRIP) < _tmp0.cc > _tmp1.cc
      $(XLATER) _tmp1.cc > _tmp2.cc
      $(CXX) $(CXXOPT) -o $@ -c _tmp2.cc
      rm -f _tmp?.cc
```

One important thing to note is that the `Makefile` needs to include another file called `Makefile.common`, which actually contains definitions (i.e., macros) necessary to compile the source code. The `SSF_INCLUDES` macro lists all directories that the compiler needs to search for header files of PRIME SSF. The `SSF_CFLAGS` macro has all the compiler flags consistent with the SSF runtime system. Both macros must be used when compiling the source code. `SSF_LDFLAGS` and `SSF_LDADD` contain the flags for the linker as well as precompiled libraries necessary for building the executable file.

There are several steps one must take to compile a source file:

1. Run the preprocessor (`CXXCPP`) on the source code, which will generate a temporary file called `_tmp0.cc`,
2. Get rid of the comments in the `_tmp0` file using `CMTSTRIP`, the result of which will be put into another temporary file called `_tmp1.cc`,
3. Run the source-to-source translator (`XLATER`), which will generate the new source code in `_tmp2.cc`,
4. Compile `_tmp2.cc`, which will generate the object file,
5. Delete all temporary files generated by this process.

3.3.2 The Results

To compile the *muxtree* example, we need to first go to the directory containing the source files and the Makefile (i.e., the `examples/muxtree` directory under the PRIME root directory). We are then ready to build the model:

```
% make depend
% make
```

After that, an executable (`muxtree`) will be created. The `make depend` command is optional yet useful for creating the correct dependencies of the source code. Now, we can start running the muxtree model from time 0 to time 100, assuming we use two processors.

```
% ./muxtree -n 2 0 100
```

The result of the simulation run may look like the following:

```
------------------------------------------------------------
Parallel Real-time Immersive Simulation Environment (PRIME)
Scalable Simulation Framework (SSF) 1.0
Copyright (c) 2005-2006 Colorado School of Mines.
PRIME SSF is open source.
See COPYRIGHT for detailed licence information.
------------------------------------------------------------

Simulation time [0, 1e+08]
[0] setup time: user = 0.05, system = 0.01
[0] DECADE LENGTH = 4.61169e+18
[0]-------------- processor 0 --------------
[0]  CPU time: user = 1.67, system = 0.02
[0]  timelines = 36
[0]  entities = 292 (329 created here)
[0]  kernel events = 58221
[0]  messages = 3484 [xprc=1932, xmem=0]
[0]  timeline context switches = 218
[0]  process context switches = 58185
[0]-------------- processor 1 --------------
[0]  CPU time: user = 1.67, system = 0.02
[0]  timelines = 37
[0]  entities = 293 (256 created here)
[0]  kernel events = 59242
[0]  messages = 3484 [xprc=1552, xmem=0]
[0]  timeline context switches = 228
[0]  process context switches = 59205
======================== summary =========================
  execution time (max wall-clock time) = 1.87014
  CPU user time: total = 3.34, avg = 1.67
  total timelines = 73
  total entities = 585
  total kernel events = 117463 (rate = 62809.7/sec)
  total messages = 6968 (xprc=3484, xmem=0)
  total timeline context switches = 446 (rate = 238.485/sec)
  total process context switches = 117390 (rate = 62770.7/sec)

(73 records) nrcvd=57978, nlost=50333, nsent=7134
```

The system prints out information, which can be divided into three parts. First, the title and version of the particular PRIME SSF distribution are given, followed by the runtime parameters used by the kernel. It also shows that the simulation runs from time 0 to 10^8. This is because we choose to use microsecond precision in the simulation (defined as a macro in the program we did not show). 100 seconds translates to 10^8 microseconds or simulation ticks seen by the simulation runtime system. Second, each processor on every machine prints out a report about the simulation run. The report shows the number of timelines and entities assigned to the processor and the total number of kernel events processed. The number of messages, defined as cross-timeline events sent by each processor, is also shown. Some of these messages went across processor boundaries on the shared-memory multiprocessor (`xprc`); others traveled across memory space boundaries between distributed machines (`xmem`). The report also shows the number of timeline context switches and the number of process context switches on each processor. A timeline context switch happens when a timeline (which is internally represented as a logical process) is suspended and replaced by another timeline in the runtime system's timeline scheduler. A process context switch happens when a simulation process resumes execution as a result of the previous running process suspending its execution and yielding the CPU. Both these numbers are used as a measure of overhead due to the parallel synchronization algorithm. The report is prefixed by a machine index shown in squared brackets at beginning of a line. Last, there is summary report showing the totals for the entire simulation. The execution time is the maximum among the execution times on all machines we use for the simulation. Note that the execution time does not include the time during which the simulator builds the model and generates this report.

The user output is shown at the end of the report. In this example, the multiplexer entities received a total of $57,978$ packets, lost most of them ($50,333$), and allowed only $7,134$ packets to pass through. This result is largely due to the small buffer size and large fan-in factor at each multiplex.

In our example, we configured the runtime system as stand-alone (i.e., single shared-memory node), although our model is built to handle parallel situations. Assuming MPI has been installed properly, we can rebuild the simulation runtime system and the muxtree model:

```
% cd prime-root-directory
% make clean
% ./configure --with-ssf-sync=mpi
% make
% cd examples/muxtree
% make clean
% make depend
% make
```

The simulation program in this case can be considered as a regular MPI application. Suppose that you have MPICH installed and you want to run the simulation on two machines, you can start the simulation using the following command:

```
% mpirun -np 2 -machinefile mymachlist ./muxtree 0 100
```

The `mymachlist` file contains a list of machines that you can run your MPI applications. In this example, we do not need to specify the machine configurations using the `-nmach` option, in which case SSF assumes we use single-processor machines. If the user decides to use multiple processors on each of two dual-processor machines, for example, you can run the simulation using:

```
% mpirun -np 2 -machinefile mymachlist ./muxtree -nmach 2 2:2 0 100
```

The result is shown in the following. Note that the result from the simulation model is exactly the same as running in as stand-alone. The reason is that we seed the random number generators in the source entities

exactly the same. Here, repeatability is very important to simulation models. The user can change the seed of all random number generators using the command option `-seed`. In this way, the modeler can get different statistical samples for simulation output analysis.

```
------------------------------------------------------------
Parallel Real-time Immersive Simulation Environment (PRIME)
Scalable Simulation Framework (SSF) 1.0
Copyright (c) 2005-2006 Colorado School of Mines.
PRIME SSF is open source.
See COPYRIGHT for detailed licence information.
------------------------------------------------------------

Simulation time [0, 1e+08]
[0] setup time: user = 0.01, system = 0.01
[0] DECADE LENGTH = 1e+06
[1] setup time: user = 0.01, system = 0
[1] DECADE LENGTH = 1e+06
[1]-------------- processor 0 --------------
[1]  CPU time: user = 1.03, system = 0.07
[1]  timelines = 18
[1]  entities = 146 (164 created here)
[1]  kernel events = 29211
[1]  messages = 1742 [xprc=388, xmem=966]
[1]  timeline context switches = 1836
[1]  process context switches = 29193
[1]-------------- processor 1 --------------
[1]  CPU time: user = 1.03, system = 0.07
[1]  timelines = 18
[1]  entities = 146 (128 created here)
[1]  kernel events = 29257
[1]  messages = 1742 [xprc=388, xmem=966]
[1]  timeline context switches = 1836
[1]  process context switches = 29239
[0]-------------- processor 1 --------------
[0]  CPU time: user = 1.04, system = 0.06
[0]  timelines = 18
[0]  entities = 138 (120 created here)
[0]  kernel events = 28194
[0]  messages = 1645 [xprc=388, xmem=679]
[0]  timeline context switches = 1836
[0]  process context switches = 28176
[0]-------------- processor 0 --------------
[0]  CPU time: user = 1.04, system = 0.06
[0]  timelines = 19
[0]  entities = 155 (173 created here)
[0]  kernel events = 30801
[0]  messages = 1839 [xprc=578, xmem=873]
[0]  timeline context switches = 1938
```

```
[0]  process context switches = 30782
======================== summary ========================
  execution time (max wall-clock time) = 2.57122
  CPU user time: total = 4.14, avg = 1.035
  total timelines = 73
  total entities = 585
  total kernel events = 117463 (rate = 45683.8/sec)
  total messages = 6968 (xprc=1742, xmem=3484)
  total timeline context switches = 7446 (rate = 2895.9/sec)
  total process context switches = 117390 (rate = 45655.4/sec)

(73 records) nrcvd=57978, nlost=50333, nsent=7134
```

4 SSF Core API

In this section, we discuss in detail about the core Application Programming Interface—the five SSF classes and associated types in PRIME SSF. A user simulation program only needs to include the `ssf.h` header file for these class definitions and function prototypes.

4.1 Simulation Phases

The entire simulation run can be divided into four phases.

- `Preparation Phase`: At the beginning of a simulation run, if the user provides an SSF DML model description file, or if there is a file named "`.submodel.dml`" in the current working directory, the kernel will load the file and construct the model before the user main function is called. Entities will be created according to the DML description. Inside the entity's constructor or `init` method, processes, input channels, and output channels are created as well. If no such DML file is found, this preparation phase is skipped. See section 6 for more details on building a simulation model using the SSF DML model description file.

- `Initialization Phase`: After the preparation phase, if the main function is provided in the simulation program, the runtime system will invoke the main function, in which the user may construct the model (see the muxtree example in section 3). In the main function, entities can be created before the `Entity::startAll` method is called. Before `Entity::startAll` returns, the `init` method of these entities will be invoked by the runtime system and the user can further create more entities. If so, the `init` method of these entities will be called as well. During this time, processes, input channels, and output channels are created. Re-alignment of these entities and mapping of these channels can also be specified. The initialization phase ends when the `startAll` returns. If the user, however, does not provide the main function, the initialization phase is skipped.

- `Running Phase`: If the main function is present in the simulation program, the simulation starts from the specified start time as soon as the `Entity::startAll` method is called. The `startAll` method is similar to the `fork` system call which is used to spawn other processes in Unix. In SSF, the simulation processes are created as a result of the `startAll`. The user usually will call `Entity::joinAll` method right after the `startAll` method returns in order to freeze the main process until the simulation ends at the pre-specified end time (which is also a parameter to the `startAll` method). If the user main function is omitted, the simulation starts to run as soon as

the preparation phase is over. That is, the runtime system will call `startAll` and `joinAll` implicitly.

- `Wrapup Phase`: When the simulation time reaches the pre-specified end time, the kernel will terminate all running simulation processes and the `wrapup` method of these processes are called. After that, the `Entity::wrapup` method is called for each entity in simulation. If the main function is defined, the `Entity::joinAll` method will then return and the user will have the chance to print out statistics of this simulation run at the end. Also, if a global wrapup function is defined, the runtime system will invoke this function immediately after the main function returns.

4.2 Types and Classes

There are several types defined by the SSF API. The most important is the type of the simulation time used by the runtime system. In PRIME SSF, simulation time (defined as the `ltime_t` type) can assume one of the following primitive types:

- `float`: single precision floating point numbers;
- `double`: double precision floating point numbers;
- `long`: long integer type;
- `long long`: long long integer type.

The user must choose one of above types to represent the simulation time when configuring the PRIME SSF runtime system. See section 2.3 for details about configuring PRIME SSF. Once set, the simulation time type cannot be changed unless the runtime system is reconfigured and rebuilt. The SSF API also supports a `boolean` type with two predefined constants: `true` and `false`. Actually, the `boolean` type is type-defined as an integer.

The SSF API defines five core classes: `Entity`, `process`, `Event`, `inChannel`, and `outChannel`. In the following sections, we discuss these classes in more detail.

4.3 The `Entity` Class

The `Entity` class is the base class that represents a logical component in simulation. It serves as the container class for simulation state variables, including instances of the `Process` class, the `inChannel` class, and the `outChannel` class. A skeleton of the `Entity` class is shown in the following:

```
class Entity {
public:
  virtual void init();
  virtual void wrapup();

  ltime_t now();

  Object* alignment();
  ltime_t alignto(Entity*);
  ltime_t makeIndependent(int = -1);

  Entity** coalignedEntities();
  Process** processes();
```

```
    inChannel** inChannels();
    outChannel** outChannels();

    static void startAll(ltime_t);
    static void startAll(ltime_t, ltime_t);
    static void joinAll();

    virtual void config(Configuration*);
    static char* getenv(char* name);

    void dumpData(int, ssf_compact*);
    static Enumeration* retrieveDataDumped();

    long schedule(void (Entity::*)(Event*), Event*, ltime_t);
    long schedule(Process*, void (Process::*)(Event*),
                  Event*, ltime_t);
    boolean unschedule(long);
    void schedule_nonretractable(void (Entity::*)(Event*),
                                 Event*, ltime_t);
    void schedule_nonretractable(Process*,
      void (Process::*)(Event*), Event*, ltime_t);

    boolean isSimple();
    void waitOn(inChannel**);
    void waitOn(inChannel*);
    void waitForever();
    void suspendForever();
    void waitFor(ltime_t);
    void waitUntil(ltime_t);
    boolean waitOnFor(inChannel**, ltime_t);
    boolean waitOnFor(inChannel*, ltime_t);
    boolean waitOnUntil(inChannel**, ltime_t);
    boolean waitOnUntil(inChannel*, ltime_t);
    void waitOn();
    boolean waitOnFor(ltime_t);
    boolean waitOnUntil(ltime_t);
    void waitsOn(inChannel**);
    void waitsOn(inChannel*);
    inChannel** activeChannels();
  };
```

The public methods can be classified into three groups: basic SSF entity methods, methods for direct event scheduling, and aliasing methods for processes with entity methods as procedures. The next three sections describe these methods in detail.

4.3.1 Basic SSF Entity Methods

Basic SSF entity methods include methods that are either originally defined by the SSF Specification or PRIME SSF extensions to the standard SSF API. These methods are used for basic entity operations includ-

ing entity construction, initialization, inquiries for entity state, finalization, and destruction. There is a static method used to fetch runtime environment variables. Others include methods used for collecting simulation statistics.

- The `init` method is called by the runtime system after the entity has been created and before any process that belongs to the entity starts to run. The method is expected to be called at the same simulation time when the entity is created. The user may override this method in the derived entity class. Similarly, the `wrapup` method is called by the system just before the entity is reclaimed. The runtime system reclaims all entities when the simulation finishes. The user may also override the method in the derived entity class.

- The `now` method returns the current simulation time. It is left undefined if called before the simulation starts or after the simulation has finished. During simulation, all coaligned entities advance their simulation time synchronously. That is, all entities sharing the same timeline (see below) use the same simulation clock.

- The `alignment` method returns a generic pointer that points to a timeline object to which this entity is aligned. The timeline object is internal to the SSF implementation. That is why the user only gets a pointer of the `Object` type, which is actually defined as `void`. The user can only use this pointer to compare with that of another entity in order to draw conclusions whether the two entities share the same timeline. Entities having the same alignment pointer are considered to be coaligned and are therefore free to access each other's state variable directly.

- The `alignto` method changes the alignment of the current entity and align itself to another entity. A timestamp at which this alignment change takes into effect is returned. The alignment remains unchanged until the simulation clock gets to the returned timestamp. Note that realignment is not transitive—entities coaligned with this entity before calling this method will not change their timeline. The `makeIndependent` method changes the alignment of the current entity and creates a timeline for itself. A timestamp is also returned; the alignment remains unchanged until the simulation clock advances to the returned timestamp. The user can provide an index to a processor to which the new timeline is assigned. If omitted, the runtime system randomly assign the timeline to a processor. In PRIME SSF, realignment is allowed only at the preparation and initialization phase. Dynamic alignment of entities during simulation is not allowed.

- The `coalignedEntities` method returns a null-terminated array of pointers to all entities that are coaligned with this entity. At least, the array should contain a pointer to the current entity itself. Similarly, the `processes` method returns a list of processes owned by this entity, the `inChannels` method returns a list of all input channels owned by this entity, and the `outChannels` method returns a list of all output channels owned by this entity. The returned array belongs to the system and is only immediately accessible to the user. That is, the user must not reclaim the array and the user should not store the returned value (i.e., the pointer to the array) and later access its elements. The user must make an explicit copy if the array is used later. Otherwise, the runtime system may reclaim the array when the current process is blocked.

- The two `startAll` methods initiate the simulation. Both the start time (default to zero) and end time of the simulation run are provided. The simulation will run over this interval, inclusive of both endpoints. The method can be considered as a fork system call, in which separate simulation processes are spawned. And the main function will continue after calling this method until the `joinAll` method is called. After that, the main process will be suspended until the simulation finishes at

the specified simulation end time and the control is then returned to the next instruction after the `joinAll` method in the main function.

- The `config` method is called if and after the entity is created by the runtime system from the SSF DML model description file. To do that, the entity must be a public entity. That is, the entity must be registered with the runtime system (using the `SSF_REGISTER_ENTITY` macro) so that its name can be used in the SSF DML description file. If, in the SSF DML description, there is a `CONFIGURE` attribute within the `ENTITY` attribute, this method will be called with the DML attribute that represents the `CONFIGURE` attribute as an argument to the macro.

- The `getenv` method returns the value of an environment variable. This static method searches the list of all environment variables defined by the simulation for a string that matches the given name. If the model is created from the SSF DML model description file, the environment variables are defined using the ENVIRONMENT attribute in the SSF DML model description file (see section 6 for more details)..

- The `dumpData` method is used to write the statistical data of this entity out to a file, while the `retrieveDataDumped` method is used to collect the statistic data at the end of the simulation. See section 7.5 for an in-depth discussion of the statistics data collection in PRIME SSF.

4.3.2 Direct Event Scheduling

There is a nontrivial overhead associated with supporting the process-oriented simulation world-view. The overhead is mainly due to the process context switches. Although we use the handcrafted multithreading mechanism to reduce the context switching cost and memory consumption, process-oriented simulation is still more expensive than an event-driven approach. To avoid the overhead, in addition to supporting process-oriented simulation, PRIME SSF also provides mechanisms for the event-driven simulation paradigm. The user can directly schedule events without having to depend on processes. The family of the `schedule` methods in the `Entity` base class provides such capability.

The `schedule` methods can directly insert an event into the event-list. The event is expected to happen after an arbitrary amount of delay specified as one of the parameters (or zero if the delay is not specified). When the event happens, the callback function, which is provided as an argument to the `schedule` methods, will be invoked. The callback function can be a member function of the current entity or a process owned by this entity (or its coaligned entities). In the latter case, a pointer to a `Process` object must be provided as an argument. The `schedule` methods return a handle which can be used later to unschedule the event that has just been scheduled. Unscheduling an event that has happened at an early simulation time bears no effect. The `unschedule` method returns `true` if the event has really been cancelled. Otherwise, it returns `false`, meaning that the event has happened already. The `schedule_nonretractable` methods are similar to the corresponding `schedule` methods, except that the events thus scheduled cannot be cancelled. These methods are a little faster than their `schedule` counterparts, because they can avoid the overhead of bookkeeping in the runtime system.

Note that the callback function, whether it is a method of the `Entity` class or the `Process` class, should not contain any wait statements (such as `waitFor`). That is, the callback function must be a regular C++ function which, when called by the system, should proceed until the function returns without process suspension. It is only an event handler in the discrete-event simulation.

PRIME SSF imposes strict rules about referencing simulation events. It is an important design decision adopted by PRIME SSF since events are the most important asset in a discrete-event simulation. To reduce the memory space occupied by events and to avoid the overhead due to memory allocation and deallocation associated with events, PRIME SSF uses reference counters. According to the event referencing rules, an

event presented to the runtime system by calling methods such as `schedule` will be no longer owned by the user after the call. The event since then belongs to the runtime system and the system will be responsible for the rest of its life cycle. If the user wants to keep this event for later access, he or she must call the `Event::save` method to create another reference to the event. Details about using events are discussed in more detail in section 4.7.

4.3.3 Methods for Processes

The standard SSF API specifies that the `Process::action` method provides the only starting procedure of a simulation process. One disadvantage of this scheme is that for the process to access the state of the entity that owns it, the process must explicitly acquire a pointer to the entity and reference the state of the entity through that pointer. Thus access to the entity state variables must be treated in a syntactically different way from access to process state. PRIME SSF, following the design of DaSSF, allows any method of an `Entity` sub-class to be specified as the starting procedure of a process. In addition, PRIME SSF provides the same set of methods that are defined in the `Process` class so that an entity method that functions as a starting procedure can directly call these `Process` methods (such as `waitOn`) without having to go through the pointer to the `Process` object.

```
boolean isSimple();
void waitOn(inChannel**);
void waitOn(inChannel*);
void waitForever();
void suspendForever();
void waitFor(ltime_t);
void waitUntil(ltime_t);
boolean waitOnFor(inChannel**, ltime_t);
boolean waitOnFor(inChannel*, ltime_t);
boolean waitOnUntil(inChannel**, ltime_t);
boolean waitOnUntil(inChannel*, ltime_t);
void waitOn();
boolean waitOnFor(ltime_t);
boolean waitOnUntil(ltime_t);
void waitsOn(inChannel**);
void waitsOn(inChannel*);
inChannel** activeChannels();
```

Most of these method calls are variations of what is called *wait statements*, and therefore these methods can only be called within a procedure. These methods are aliases to the corresponding methods defined in the `Process` class. We discuss them in more detail in section 4.4.

4.3.4 Public Entities

The runtime system may assume the responsibility of partitioning a simulation model and building the model in a distributed fashion. That is, the entities are created by the runtime system on distributed computers according to the partitioning result. In this case, the user only needs to provide factory methods for these entities and map the names of these entities with the factory methods so that the runtime system will be able to create them when parsing the SSF DML model description file. The entities created using the above method are call *public* entities. In another word, public entities can be published as standard building blocks of a model. For instance, in a network model, the user can create a standard library of network

components—hosts, routers, protocols, etc. All these network components can be defined as public entities so that the library can be used to create different network models by selectively using these components with different parameters and connecting them in a specific way. We describe the format of the SSF DML model description file in section 6.

In PRIME SSF, entities are static objects. Every entity, public or private, belongs to the system. That is, the user does not have further jurisdiction over the life cycle of a created entity. In particular, the user cannot delete an entity. The entity is expected to be reclaimed by the system when the simulation ends.

Every public entity should be registered with the runtime system. This is achieved by providing a factory callback method (which is a *static* method of the derived entity class). The prototype of the callback function is the following:

```
static Entity* (*SSF_ENTITY_FACTORY)(const ssf_entity_dml_param**);
```

The function takes a null-terminated list of parameters (pointers to objects of the `ssf_entity_dml_param` type), which are decoded by the runtime system from parsing the DML model description file. The callback function must return a newly constructed entity object. A null returned from the callback function means that the callback function has trouble decoding the parameter list to create a new entity instance, which will cause the runtime system to generate an error. The `ssf_entity_dml_param` class is defined as follows:

```
class ssf_entity_dml_param {
 public:
  enum param_type_t {
    TYPE_INTEGER = 1,
    TYPE_FLOAT   = 2,
    TYPE_STRING  = 3
  };

  int get_type() const;
  boolean get_value(long& intval) const;
  boolean get_value(double& fltval) const;
  boolean get_value(const char*& strval) const;
};
```

Each `ssf_entity_dml_param` instance represents a parameter to the entity constructor. Each instance corresponds to a PARAMS attribute in SSF DML model description. Each parameter has a type: `TYPE_INTEGER`, `TYPE_FLOAT`, or `TYPE_STRING`. The type of the parameter can be determined by calling the `get_type` method. Each parameter also contains the value, which can be retrieved by one of the three methods, depending on the type of the parameter.

Also, the user should map the name of the entity class to its factory method. In this way, the runtime system will be able to instantiate the entity object when decoding the DML file. PRIME SSF provides a macro that establishes the mapping. The macro must be placed in the source file where the entity methods are defined:

```
SSF_REGISTER_ENTITY(myentity, myentity_factory_method);
```

The following is a code snippet one can use to register the multiplexer entity class in the muxtree example. It shows the callback function as the factory method:

```
// in muxtree.h
class MultiplexerEntity : public Entity {
```

```
  ...
  // The factory method is static.
  static Entity* create_instance(const EntityParam**);
};

// in muxtree.cc
Entity* MultiplexerEntity::create_instance
  (const ssf_entity_dml_param** params)
{
  // there must be two and only two parameters, or return error.
  if(!params || !params[0] || !params[1] || params[2]) return 0;

  // the parameters must be a string and an integer, or return error.
  if(params[0]->get_type() != ssf_entity_dml_param::TYPE_STRING ||
     params[1]->get_type() != ssf_entity_dml_param::TYPE_INTEGER)
    return 0;

  // get the parameters, create the instance, and return it
  char* name; get_value(name);
  int id; get_value(id);
  return new MultiplexerEntity(name, id);
}

SSF_REGISTER_ENTITY(MultiplexerEntity,
                    &MultiplexerEntity::create_instance);
```

4.3.5 Entity Alignment

Each entity has a timeline. Entities that share the same timeline are said to be coaligned. Entities that only coaligned with themselves are said to be independent. Timeline is an implementation-dependent data structure that is not disclosed to the user. The user can only get an alignment pointer returned from calling the `alignment` method; the pointer (which is cast to `void*`) can only be used to compare with the pointers obtained from other entities to determine whether they are coaligned or not.

When entity is created, it uses the timeline of the calling method, which we call *context*. For example, if the entity is created inside another entity's `init` method, these two entities will be coaligned. If the entity is constructed inside the main function, the entity will be coaligned with all other entities that are created by the main function or its subroutines. An entity alignment will not change until one of the realignment methods (i.e., `alignto` and `makeIndependent`) is called.

Dynamic realignment is not implemented in PRIME SSF. It is not to say that calls to the `alignto` and `makeIndependent` methods will create an error after the simulation has been initialized (i.e., after the `startAll` method is called). The runtime system simply returns a timestamp which is beyond the simulation end time.

4.4 The `Process` Class

A simulation process in SSF is represented by the `Process` class. A simulation process specifies the state evolution of the logical component represented by its owner entity. A simulation process starts as soon as the `Process` object is created. A process may be blocked by waiting for a message to arrive on a specified

input channel or a set of input channels. It may also be blocked waiting for a specified period of simulation time to pass. Process suspension happens as a result of the wait statement, which is defined as a function call to one of the wait methods in the `Process` base class. We start by showing the skeleton of the `Process` class in the following:

```
class Process {
 public:
  Process(Entity*, boolean = false);
  Process(Entity*, void (Entity::*)(Process*), boolean = false);
  virtual ~Process();

  virtual void init();
  virtual void wrapup();

  virtual void action();

  void waitOn(inChannel**);
  void waitOn(inChannel*);
  void waitForever();
  void suspendForever();
  void waitFor(ltime_t);
  void waitUntil(ltime_t);
  boolean waitOnFor(inChannel**, ltime_t);
  boolean waitOnFor(inChannel*, ltime_t);
  boolean waitOnUntil(inChannel**, ltime_t);
  boolean waitOnUntil(inChannel*, ltime_t);
  void waitOn();
  boolean waitOnFor(ltime_t);
  boolean waitOnUntil(ltime_t);

  void waitsOn(inChannel**);
  void waitsOn(inChannel*);

  boolean isSimple();
  Entity* owner();
  ltime_t now();
  inChannel** activeChannels();
};
```

4.4.1 Processes and Procedures

The constructor of the `Process` class specifies the starting procedure of the simulation process. A starting procedure is either a `Process` method or an `Entity` method. The standard SSF API specifies that each process should only start from the `Process::action` method. PRIME SSF extends the API to include methods of the owner entity to be the starting procedures of processes as well. A starting procedure is at the bottom of the calling stack allocated for the thread that implements the simulation process. The body of a starting procedure may consist of an arbitrary number of regular instructions that take up zero simulation time. That is, these regular instructions including function calls do not advance the simulation clock of

this process. The simulation clock can only be advanced if wait statements are encountered, in which case the process are suspended potentially to allow the simulation clock of this process to change. We discuss these wait statements in the next section. In a typical process, regular instructions are interspersed with wait statements. The starting procedure can call other methods that have wait statements, in which other methods with wait statements may be invoked too. The calling sequence is recorded in a stack maintain for each simulation process. We call all these methods that contain wait statements *procedures*.

A process starts to run as soon as the corresponding `Process` object is created. A process should not be created by another process unless both processes are owned by the same entity or coaligned entities. The starting simulation time of a newly created process is the same as the simulation time when its constructor is called. In SSF, when the control reaches the end of the starting procedure, the process continues its execution from the beginning of the procedure as if there is an infinite loop that wraps around the starting procedure. Semantically, this means a process will run forever, unless the process is terminated when the process is blocked beyond the the simulation end time.

Note that if the starting procedure is a method of the derived entity class, the procedure function must take one parameter—a pointer to the process object. This pointer can be used to access the process's state variables and member functions. If the modeler chooses to use the `action` method in the `Process` class as the starting procedure, there is no such parameter, since we know that the class instance (i.e., via the implicit `this` pointer) is the simulation process of the starting procedure.

In order to cut the cost due to context switching in scheduling simulation processes, the standard SSF API added a special type of processes called *simple* processes. A simple process is defined as a process in which the control returns to the thread scheduler immediately after a wait statement. That is, the execution path of a simple process ends with a wait statement. Conceptually, this means that the control of this process should encounter one and only one wait statement before it reaches the end of the starting procedure. It is important to know that there can be as many wait statements in the process as there are different execution paths. Also, a wait statement can be in any procedure (not necessarily the starting procedure) of this process as long as the control immediately returns from this procedure and then from the starting procedure after the wait statement is reached. The reason that we want to use simple processes is that the runtime system can use the process continuation technique to reduce the overhead needed for saving the state of the processes at process context switches. Using simple processes can therefore significantly accelerate a simulation run.

The `init` method is called by the runtime system after the process object has been created and before the process' starting procedure is called. The user may override this method in the derived process class. This method is expected to be called at the same simulation time as when the process is created. The `wrapup` method is called just before the process is reclaimed by the runtime system. The system reclaims all processes when simulation finishes. It is also possible that the runtime system deletes a process once it realizes that the process will remain to be blocked for the rest of the simulation.

4.4.2 Wait Statements

The wait statements are function calls to the specific `Process` methods. A wait statement can potentially block a process from execution until a certain condition is met. The condition could be an event has arrived at an input channel of the owner entity that a process is waiting on. It could also be a period of simulation time has elapsed. There are therefore in principle two types of wait statements. One waits for a message to arrive on the specified input channel(s). The other waits for a period of simulation time to pass. There are also wait statements that combine both. In any case, a process may suspend its execution until the event it is waiting for happens. The wait statements can only be called within the procedure of a process. It is an error to call a wait statement in a regular function.

- The `waitOn` methods are used to block a process until an event arrives at a specified input channel

or an array of input channels. If an array is given, the array is expected to be null-terminated. The input channels must be either owned by the owner entity of the calling process or owned by entities coaligned with the owner entity of the calling process. An empty list will cause the calling process to be suspended forever.

- The `waitFor` method is used to block a process until after the specified amount of simulation time has elapsed. Similarly, the `waitUntil` method is used to block a process until a specific point in simulation time. Semantically, the `waitUntil` method can be implemented using the `waitFor` method. It is, however, more than just a syntactic sugar. The `waitUntil` method can be used to prevent situations where adding simulation timestamps may create round-off errors.

- The `waitOnFor` methods are designed as a combination of the `waitOn` and `waitFor` methods. The calling process will be suspended until an event arrives at the specified input channel or channels, or until the specified amount of simulation time has elapsed, whichever happens first. The method returns `true` if it is timed out and `false` if an event arrives at an input channel. The `waitOnUntil` methods behave similarly except that the calling process is blocked until a specific point in simulation time.

- The `waitForever` method is used to suspend a process forever. Semantically, it is identical to terminating the process. The runtime system reclaims the process immediately. The `suspendForever` method also terminates a process. However, the process object will not be reclaimed until the simulation ends.

- Under the standard SSF API, if a process wishes to wait on a large array of input channels repeatedly, *every time* it calls a wait function it must specify the array as an argument. The SSF implementation must be prepared to deal with a different array on every invocation. Certain efficiencies are possible if we can indicate to the runtime system that the process can identify a static set of channels that it will repeatedly wait on. PRIME SSF extends the API by providing `waitsOn` methods to let the runtime system set the static channels to be used by default. It is understood, if the channels are not present in the wait statement's argument list, the process should wait on the list of channels last set by the `waitsOn` method.

4.4.3 Query Methods

There are methods in the `Process` class whose goal is to help the user determine the state of a process. The state of a process can only be queried by coaligned entities or other processes whose owner entities are coaligned with the owner entity of this process.

- The `isSimple` method returns `true` if the process is simple, or `false` otherwise. Whether the process is simple or not must be determined when the process is created. This property is immutable.

- The `owner` method returns a pointer to the owner entity of this process.

- The `now` method returns the current simulation time of the entity. This method is an alias to the `Entity::now` method.

- The `activeChannels` method can only be called within a procedure. The method returns a null-terminated array of input channels, which contain newly arrived events that unblock the process.

4.5 The `inChannel` Class

The `inChannel` class represents the end point of a communication link between entities. An input channel can be mapped from several output channels. Similarly, an output channel can be mapped to multiple input channels. An event sent from an output channel of an entity will be delivered by the runtime system to a mapped input channel of another entity. Therefore, information is exchanged in SSF through message passing. Processes may be blocked on a set of input channels for message arrivals. An event that arrives at an input channel on which no process is waiting will be discarded by the runtime system implicitly. The `inChannel` class provides necessary methods for retrieving the messages arrived at an input channel.

```
class inChannel {
 public:
  inChannel(Entity*);
  inChannel(char*, Entity*);
  virtual ~inChannel();

  Entity* owner();
  Event** activeEvents();
  void publish(char*);
};
```

PRIME SSF defines three types of input channels: internal, external local, and external global. Internal input channels are unnamed input channels. An unnamed input channel is unknown externally and therefore cannot be referenced outside the current address space. External local input channels are those that are created unnamed and then published using the macro:

```
SSF_PUBLISH_DML_INCHANNEL(inChannel*, char*);
```

The input channels thus defined can be referenced in the SSF DML model description file. It is important to know that the name provided here is not published globally and therefore cannot be directly referenced from another address space. The input channel does not have to have a globally unique name as well, as long as it is unique among all input channels of the owner entity. External global input channels are defined with globally unique string names and therefore can be referenced from another memory space. This type of input channels is either created using the constructor that requires the global name as the argument, or created as unnamed but then assigned with a name using the `publish` method.

The `owner` method returns a pointer to the owner entity of the input channel. All input channels are statically owned by an entity. The ownership is immutable. The `activeEvents` method should be called within a procedure. The method returns a null-terminated array of events which have just arrived at this input channel. The events listed in the array must be accessed according to the special event referencing rules. In particular, the events belong to the runtime system and the process is only entitled to access them immediately after the method call and before it becomes blocked again. The user needs to call the `Event::save` method to create a reference to the event if the event is expected to be used at a later time. In section 4.7, we discuss the use of events in more detail. Note that an event not retrieved by this method is going to be reclaimed by the system implicitly. This is frequently the case where the user only wants to use the event mechanism to wake up processes. Also, it is important to know that event retrieval is non-destructive. That is, if multiple processes are waiting on the input channel, all processes will be able to receive an event. Each process, however, is only given the opportunity to receive the event exactly once.

4.6 The `outChannel` Class

The `outChannel` class represents the starting point of a communication link between entities. An output channel can be mapped to several input channels. An event written to an output channel will appear at all corresponding input channels that are mapped from the output channel. Different from the `inChannel` class, there is no external global output channels. That is, there is no such constructor that associates a globally unique string name to an output channel. Therefore the output channel cannot be referenced from another address space. An external local output channel can be either created directly by providing the name to the output channel to the constructor, or first created unnamed but then published using the following macro:

```
SSF_PUBLISH_DML_OUTCHANNEL(outChannel*, char*);
```

The names of published output channels do not need to be globally unique as long as they are unique among the output channels of the owner entity. Similar to the `inChannel` class, once the output channel is published, it can be referenced in the SSF DML model description. The `outChannel` class is defined in the following:

```
class outChannel {
 public:
  outChannel(Entity*, ltime_t = 0);
  outChannel(char*, Entity* theowner, ltime_t = 0);
  virtual ~outChannel();

  Entity* owner();

  virtual void write(Event*, ltime_t = 0);

  ltime_t mapto(inChannel*, ltime_t = 0);
  ltime_t mapto(char*, ltime_t = 0);
  ltime_t unmap(inChannel* tgt);
  ltime_t unmap(char*);
};
```

The delay between the time an event is written to an output channel and the time the event shows up at a mapped input channel is divided into three components: per-write delay, channel delay, and mapping delay. *Per-write delay* is the delay specified when one invokes the `write` method. The event is expected to wait for the given per-write delay before it is really sent out to the output channel. *Channel delay* is associated with each output channel. It is set when the channel is constructed and its value is immutable during the life time of the output channel. `Mapping delay` is associated with each mapped input channel and output channel pair. The mapping of an output channels and input channels (as well as the mapping delays) can be either described in the SSF DML model description file (if both channels are defined as external local) or when the `mapto` method is called explicitly. PRIME SSF requires that the sum of the channel delay and the mapping delay be strictly larger than zero.

The `write` method is used to send an event out from the output channel. A per-write delay can be specified as an argument. The delay must not be negative. A process may call the `write` method of any output channel as long as the owner entity of the process and that of the output channel are either identical or coaligned. The written event is scheduled to arrive at each of the mapped input channels after a delay equal to the sum of the channel delay, the mapping delay, and the per-write delay. Access to the event after the method call follows the event referencing rules, which we describe in section sec:event.

The `mapto` methods are used to create a mapping between the output channel and the specified input channel (either by a pointer to the input channel or by its globally unique string name). A mapping delay can also be specified if it is other than zero. The methods return a timestamp which indicates the simulation time at which the mapping will become effective. The `unmap` methods do the opposite of the `mapto` methods. They tear down connections made previously between the output channel and the given input channel. In PRIME SSF, channel mappings are allowed only at the preparation and initialization phases. Dynamic mapping of channels is not supported.

4.7 The `Event` Class

The `Event` class is the base class for messages that are passed between entities through channels. The user may directly use the `Event` class as a simple mechanism to coordinate simulation processes. If additional information is needed to pass between entities, the user must create a class derived from the `Event` class. A skeleton of the `Event` class is shown in the following:

```
class Event {
 public:
  Event();
  virtual ~Event();

  Event(const Event&);
  virtual Event* clone();

  Event* save();
  void release();
  boolean aliased();

  virtual ssf_compact* pack();
};
```

4.7.1 Event Referencing Rules

For the sake of efficiency, PRIME SSF uses a reference counter scheme to manage the events. In order to let it work, PRIME SSF imposes a strict set of rules for accessing these events. In general, an event is either owned by the user or by the runtime system. A user-owned event shall be managed by the user and the user should have total jurisdiction over its existence. That is, the event can be deleted if needed. A system-owned event is under the jurisdiction of the runtime system. The user cannot delete such event explicitly aside from increasing (using the `save` method) or decreasing (using the `release` method) the reference counter of this event.

An event is first owned by the user when it is newly created. As soon as it is presented as an argument when calling an SSF API method, such as `outChannel::write` or `Entity::schedule`, the event becomes system-owned. The number of references maintained by the user, however, does not change when the ownership of the event is transfered to the runtime system. The user keeps the references to the event until a matching number of `release` calls are made to this event. When an event is provided to the user by the runtime system—that is, the event is either retrieved explicitly (using the `inChannel::activeEvents` method) or implicitly through a callback function (such as the one specified when the `Entity::schedule` method is called)—the event still belongs to the runtime system. In any case, the user shall not delete this event. In addition, the event is only “immediately” accessible to the

user: the runtime system may reclaim this event at any point after the process has been suspended. In order to keep the event for later use, the user must create another reference to the event (using the `save` method) or an explicit copy of this event is necessary.

Each event class should have a copy constructor, in which the member data is copied explicitly. The constructor is used by the runtime system to create a new event object when the event is sent to another processor. Creating a new event object is actually achieved by the runtime system calling the `clone` method of the event class. The `clone` method must be provided by all derived event classes. This *virtual* method is expected to return a pointer to a newly created event object. The following shows an example where the `clone` method is defined in the `MyEvent` class, which derives from the `Event` base class:

```
class MyEvent : public Event {
 public:
  ...
  Event* MyEvent::clone() { return new MyEvent(*this); }
  ...
};
```

The `save` method is used to create a new reference to the event by incrementing a user reference counter. It is used when the user wants to save the event which has just been created or received from an input channel. In particular, an event given to the user by the runtime system (e.g., via the `inChannel::activeEvents` method) is only immediately accessible to the process. The runtime system may reclaim its storage without further notice after the process has been suspended. The `save` method can be called to save the event across process suspensions. This method can be called more than once on the same event. The system will not reclaim the event until a matching number of `release` calls have been made to the event. The `release` method is used to dereference the event by decrementing a user reference counter. The method is called when the user decides not to keep the saved event any more. The number of `release` calls must be less than or equal to the calls to the `save` method. When the user reference counter reaches zero, that is, when there are matching number of saves and releases, the system may reclaim the event if the event is no longer used by the runtime system.

The event provided to the user by the runtime system often has many aliases: there may exist several references to the same event. It is obvious that an event with multiple aliases should be read-only. The user is able to query whether an event has more than one reference or not. The `aliased` method returns `true` if there are multiple aliases of the same event, in which case the user should not modify the content of this event as it is expected to be shared by multiple processes. If the `aliased` method returns false, the caller can be sure that accessing to this event is exclusive and it is therefore fine to modify the event without worrying about possible conflicts.

4.7.2 Event Registration, Packing and Unpacking

In PRIME SSF, the runtime system may partition the model among distributed-memory machines. Events have to travel across memory space boundaries by means of real messages (e.g., using MPI). A user event needs to be translated into a machine-independent byte stream before it is shipped to the destination processor where the runtime system reconstructs the user event from the byte stream. A user event may be an instance of a derived event class, in which case the user must specify how to translate the event object to and from the byte stream. The user also need to provide a way for the runtime system to recognize the event at the time of reconstruction.

The translation from an user event object to a byte stream is called *event packing*. The reverse translation from a byte stream to a newly created user event object is called *event unpacking*. For event packing, the system requires that the derived event class must have a virtual method named `pack`, which the runtime

system will invoke at the time when it needs to create the byte stream before the event is to be shipped to a remote machine. The `pack` method is responsible for packing necessary data fields of the event object so that the event object can be reconstructed (unpacked) on another machine. A (machine-independent) byte stream in SSF is represented by the `ssf_compact` class, which is described in more detail in section 7.4.

In order for the runtime system to recognize the event from a byte stream at the remote machine, the user must also register the derived event class with an event factory method, which is responsible for creating a new event object of the corresponding event class and unpacking the data from the byte stream. The event factory method should be a callback function defined as a static member function of the derived event class. The prototype of the event factory method is defined as follows:

```
typedef Event* (*SSF_EVENT_FACTORY)(ssf_compact*);
```

PRIME SSF provides two macros: one is used for registering the derived event class and it must be within the class declaration; the other is used to associate the class with the event factory method and it must be used along with the definitions of the class methods in the source file:

```
SSF_DECLARE_EVENT(event-class-name);
SSF_REGISTER_EVENT(event-class-name, address-of-factory-method);
```

5 Source Code Instrumentation

5.1 Source-to-source Translation

PRIME SSF does source-to-source translation. The goal is to embed handcrafted multithreading mechanism inside regular C++ code. Using handcrafted multithreading mechanism is the design decision to deliver high-performance process-oriented simulation and make efficient use of memory space. The important strategy adopted by PRIME SSF to support multithreading is to transform the SSF model written in C++ into similar C++ source code that implements the multithreading mechanism.

The source files of an SSF model are “almost” pure C++: the syntactic correctness of the code can be checked by ordinary means, before and after substitutions are made in the text. C++ is a very complex language, a fact that has placed limitations on what we can reasonably hope to achieve in a translator. To do it “right” requires parsing any legitimate C++ program on all supported platforms, a different path we did not follow. Instead, we created a Perl script to parse the source code. Still, the restrictions we impose are relatively small. To understand the source of these restrictions, it is helpful to understand what the PRIME SSF preprocessor does.

As an SSF process executes it may call other methods. At the point it suspends, any number of method calls may be piled up on the runtime stack. When the process resumes execution, it needs to be able to return to the exact location of the function (back through that call-chain) where the process was suspended. We call these methods procedures since it is possible for the methods to be on the runtime stack at the point of a process suspension. Any method that calls a wait statement is a procedure. Any method that calls a procedure is also a procedure. The source-to-source translator instruments the bodies of procedures as well as the class declaration of classes that contain procedures.

The instrumentation of a class definition is minor, all that happens is that two new method declarations are inserted. The instrumentation of a procedure body is more complex. To begin with, the local variables of a procedure body have to be dealt with specially, at least those variables whose values are expected to persist across process suspension. The source-to-source translator requires the user to identify such local variables syntactically. It removes them as local variables altogether, creating the corresponding variables in a “frame” data structure that represents the the procedure call. All references to such variables in the body of the procedure are transformed into references to their representation in the frame. Since input

arguments are local variables that are expected to persist across process suspension, they too appear in the frame. The frame for a procedure must be created *before* the actual procedure call. The definition, construction, and placement of a procedure's frame plays a large part of the main task of the source-to-source translator. Additional complications arise if the procedure returns a value to its caller—space for an address for receiving the return value is also required in the frame.

5.2 Annotations

To find the right place to instrument the source code, PRIME SSF requires help from the modeler. We adopt a method similar to the High Performance Fortran, where special comments are used to annotate the source code in necessary places. Likewise, the PRIME SSF source-to-source translator looks for the annotations. These special annotations are flagged by the sequence `//! SSF`. A space between `!` and SSF is expected, but no other white space is tolerated.

5.2.1 State Variables

Local variables in a procedure body that must persist across suspension have to be explicitly identified as state variables. This is done using the `//! SSF STATE` annotation at the end of the line where the variable is declared inside the procedure. An example is shown below:

```
int foo() {
  ltime_t x; //! SSF STATE
  float y;   //! SSF STATE
  int z;     //! SSF STATE
  ...
};
```

Several limitations exist:

- Only one variable may appear on an annotated line;
- An annotated variable may not be initialized in its declaration;
- The annotated variables must be declared at the beginning of the procedure body;
- The variables do not respect scope, that is, the scope of an annotated variable is the entire procedure body. Consider the wrong code below:

  ```
  int foo() {
    int x;
    ...
    {  int x;
       int y;
       ...
    }
  }
  ```

  In C++ this would be perfectly legitimate code; in PRIME SSF, it is impossible to declare both instances `x` to be procedure state variables. There is no distinction between the two instances of `x`, since in PRIME SSF both of them will be visible in the procedure. The reason for this limitation is that the

source-to-source translator just scans the beginning of a procedure body, makes a list of STATE variables, and creates a frame class that has a member variable for each identified procedure state variable. A reference to `x` is transformed in the source to a reference to `frame->x`. It is within the realm of possibility to make the translator smart enough to understand nesting level and distinguish between commonly named variables at different scoping levels, but at this point the benefit is outweighed by the effort required.

A very important thing to note is that the source-to-source translator cannot detect whether a variable is used in a way that necessitates it being a state variable (e.g., its value is set before a call that potentially leads to process suspension, and that value is used again after the process resumes execution). Failure to declare such a variable to be a state variable leads to mysterious bugs. It is recommended therefore all procedure local variables should be declared as state variables.

5.2.2 Procedures

The `//! SSF PROCEDURE` and the `//! SSF PROCEDURE SIMPLE` annotations are used in two places. The line containing the declaration of a method that will be designated as a procedure (not just the starting procedure) needs to be annotated at the end of the line where the method is declared inside the class declaration. For instance, if the `update` method is a procedure declared in the **MyEntity** class, then the method should be declared like

```
class MyEntity : public Entity {
 pubic:
  ...
  void update(int); //! SSF PROCEDURE
  ...
};
```

Elsewhere, presumably in the source file, the code body for the `update` method must be given. Here too the fact that `update` is a procedure must be flagged with an annotation. In this case the annotation needs to be placed on the line immediately prior to the declaration:

```
//! SSF PROCEDURE
void MyEntity::update(int x) {
  ...
}
```

Another point, somewhat subtle, has to do with inheritance in C++. If method `A::foo()` is a virtual method and is a procedure, then any method `B::foo()` in a class `B` that is derived from `A` must also be a procedure. Similarly, if any class `C` derived from `A` has method `C::foo()` that is a procedure, then `A::foo()` must too be declared as a procedure, as must `B::foo()` for any other class `B` that derives from `A`. The reason for this is that when `A::foo()` is called, the source-to-source cannot tell which of the derived classes is actually being called. If any one of them is a procedure, then they all must be so declared in order for the instrumentation to work properly.

The SSF API calls for the identification of “simple” processes. A simple process is implemented using continuation. Therefore, we do not need to create a stack frame for the method call. To annotate a simple process one only needs to add the word SIMPLE at the end of the normal procedure annotation:

```
class MyEntity : public Entity {
 public:
```

```
  ...
  void update(int); //! SSF PROCEDURE SIMPLE
  ...
};

//! SSF PROCEDURE SIMPLE
void A::update(int x) {
  ...
}
```

Inside a simple procedure, the user does not need to use other annotations since a simple procedure body is treated exactly the same as regular methods. In particular, a simple process does not have procedure state variables (marked by `//! SSF STATE`) since there are no variables persistent across simple process suspensions. Also, a simple procedure cannot call a non-simple procedure since that could violate its own definition of being simple. That is, the `//! SSF CALL` annotation (which we discuss in the next section) should never be used in a simple procedure.

5.2.3 Procedure Calls

Instrumentation occurs where one procedure calls another. In principle one would think that, since all procedures are identified, when a call to one of these procedures is found in code body of another procedure, it should be recognized. That is true for a compiler, which has semantic information, but a mere preprocessor that handles the source code at the textual level is not that smart. Our source-to-source translator can recognize that a word in the text corresponds to some procedure method name, but it cannot figure out the class instance. We require the modeler to tell the preprocessor when a procedure call is being made. The annotation for this is `//! SSF CALL`. This is placed on the line *immediately* above the one where the call is made. A couple of examples are shown as follows:

```
//! SSF CALL
join_server(x,y,z);

//! SSF CALL
x = getMin(x,a);
```

The second case shows how to get a return value from a procedure. For this to work `x` has to be declared earlier in the code body as a procedure state variable. It is unnecessary for `a`, `x`, and `y` to be state variables, at least not by virtue of being an argument to a procedure. If the code expects their values to persist after the procedure call though, they need to be declared as state variables. It is also required that a procedure call must occupy one and only one line. And it is generally a good idea to separate the procedure call from other expressions that involving the calculation of the arguments and the return value of this procedure call, for the sake of simplicity.

5.3 Limitations

Instead of a full-fledged C++ parser, we use the simple preprocessor written in Perl to do the source-code translation. There are limitations on what one can reasonably expect what a preprocessor can accomplish. There are more limitations than what we could list here to reflect this fact.

Procedure bodies lie outside of class definition. Every Procedure method must be declared on one line in its class definition, and have the code body described outside of the class definition. The source-to-source translator may not be able to detect a violation of this.

Name uniqueness. Procedures must be distinguishable from each other by class name or function name. Other elements that constitutes a traditional C++ signature—return and argument types—do not play a role. For instance, one cannot in the same class use two procedures with the same name, one that accepts an integer argument and one that accepts a floating point argument.

Variable naming conventions. Most internally declared variables have an `_ssf_` prefix. One needs to avoid names with this prefix so that they will not conflict with any variables or methods used by the runtime system.

Comments. Nested comments are not dealt with very well by the preprocessor. Avoid them.

6 SSF DML Model

6.1 Model Construction

There are two methods one can use to construct a simulation model in PRIME SSF: the user can use the SSF DML model description to partition and construct model on a distributed simulation environment. DML stands for the Domain Modeling Language, which is in general a hierarchical list of attributes (we discuss DML in the next section). An SSF DML model description is a specialization of DML, which is used to describe simulation models in SSF. An SSF DML model description contains three DML files: *i)* a model DML file used to describe the topology of the model, including entity definitions, entity alignments, and channel mappings, *ii)* a machine DML file used to specify the hardware platform on which the simulation will execute, including the number of distributed machines that will be involved in the simulation run, as well as the type and the number of processors on each machine, and *iii)* a runtime DML file used to provide parameters for the particular simulation run, such as the starting time and end time of the simulation run.

The three DML files are first used for partitioning the simulation model. From the model topology (in the model DML file) and information about the target architecture (in the machine DML file), a graph is constructed with the nodes representing timelines and edges representing channels mappings that span across the timelines. This graph is then partitioned into N submodels, one for each distributed machine that we will use in the simulation run. The submodel can be further partitioned among multiple processors on each distributed node. The result of the partition is put in another DML file called SSF submodel DML file, which is structurally equivalent to its original DML files with additional information about the result of the partition as well as simulation runtime parameters.

When the simulation starts, the SSF submodel DML file produced by the partitioner is parsed by the SSF runtime system running on each distributed machine. The runtime system recognizes timelines, entities, and the channel mapping directives in the DML file. Each distributed machine constructs the partition of the simulation model that is assigned to it. Channels are mapped accordingly. The model is therefore built and ready for the simulation phase to start. These steps are illustrated in figure 2.

PRIME SSF also supports model construction using the user main function. The user can provide a main function to construct the simulation model programmatically in addition to or instead of the SSF DML model description. If defined, the main function must have the following function signature:

```
int main(int argc, char** argv);
```

PRIME SSF adopts the SPMD programming style. That is, the main function will be called on each of the distributed machines. Even if a machine has more than one processor, the main function is called only on processor 0. In the main function, the user is expected to create the submodel that resides on the current machine. The skeleton of the main function is as follows:

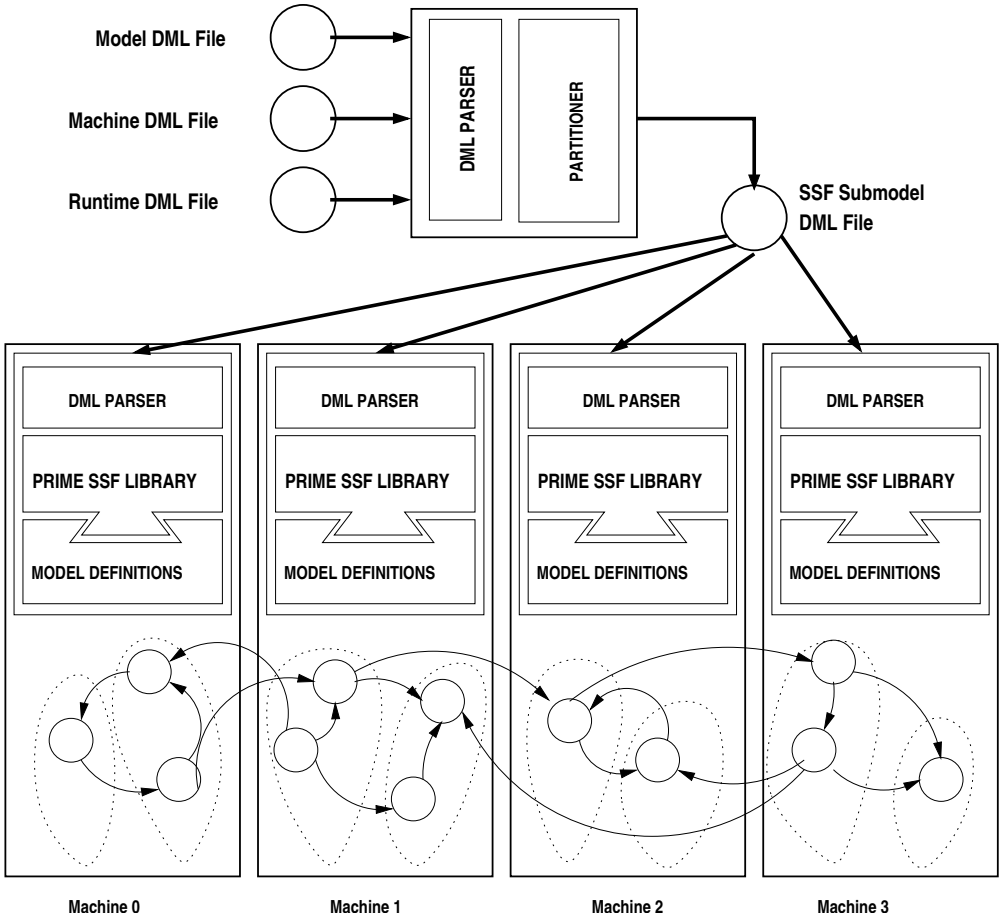

Figure 2: Model construction using the SSF DML model description.

```
int main(int argc, char** argv) {
  int nmachs = ssf_num_machines();  // total distributed machines
  int machid = ssf_machine_index(); // machine index

  /* create entities that belong to current machine (machid) */
  ...

  /* map channels of the entities */
  ...

  Entity::startAll(start_time, end_time);
  Entity::joinAll();

  if(machid == 0) {
    /* collect and report statistics */
    ...
  }
}
```

In the main function, all entities that are assigned to this machine can be created before the `Entity::startAll`

method is called (see the muxtree example in section 3). Alternatively, one can only instantiate some of the entities assigned to this machine. The runtime system will distribute these entities among all local processors randomly in a balanced fashion. When the `startAll` method is called, the runtime system will invoke the `init` method of these entities, in which other entities can be created and their `init` methods will be invoked also. In this way, entities can be created hierarchically and the model construction is carried out in parallel among multiple processors. Therefore, it is more efficient, especially for large models.

When the entities are created, processes, input channels, and output channels are also created. The user can also realign entities and map these channels. The `startAll` method only returns when the model construction has been completed and the simulation starts running. The user is expected to call the `Entity::joinAll` method right afterwards to wait for the simulation to finish. After the `joinAll` method returns, user should collect and report the statistics of the simulation run, either within the main function or with a global wrapup function.

In the remainder of this section, we describe how one can build a simulation model using the SSF DML model description. But first we need to describe DML in more detail.

6.2 Domain Modeling Language (DML)

6.2.1 DML Specification

Our approach in PRIME SSF to deploy models, especially in a distributed-memory environment, is to use the Domain Modeling Language (DML). At the most basic level, a DML script is described as a recursively defined list of attributes. A model written in DML can be thought of as a database, and can be queried like a database. An attribute in DML is a whitespace-separated key-value pair. A key is an alpha-numerical string. DML allows attributes having the same key. A value can be of any primitive type: an integer, a floating point number, or a string. A string can be a sequence of any arbitrary ASCII characters in double quotes (which can be omitted if the string does not include whitespace nor square brackets). A value can also be a list of attributes enclosed in square brackets. The DML grammar can be illustrated (informally) as the following. `INTEGER`, `FLOAT`, and `STRING` are terminals and their formats follow the standard C++ language.

```
DML             ::=  attribute-list
attribute-list  ::=  empty | attribute-list attribute
attribute       ::=  key value | key [ attribute-list ]
key             ::=  [a-zA-Z_][a-zA-Z0-9_]*
value           ::=  INTEGER | FLOAT | STRING
```

Quite rich descriptions can be embedded in this simple framework. To illustrate, the following shows a simple example. Note that Comments can be added in the DML description; everything on a line after "#" is ignored by the DML parser.

```
planet Earth    # name of the planet
weight 6.5e21   # in tons
population 6242324585  # as of 15:00:29 GMT on August 8, 2002
```

The DML specification does not impose domain specific interpretations on the attributes or values that are expressed. The interpretation is left to the domain specific code that uses DML framework. The above DML script contains three attributes. The `planet` attribute has a string value that is not placed in quotes. It is allowed since the string does not include whitespace nor square brackets. The value of the `weight` attribute is a float-point number and the value of the `population` is an integer.

DML also allows nested attributes. That is, the value of an attribute can be a list of attributes, enclosed in brackets. Now, we can expand the previous example like this:

```
planet Earth
weight 6.5e21
population [
    number 6242324585
    date "08/08/2002"
    time "15:00:29 GMT"
]
```

A DML model can always be represented as a tree of attributes. If the attribute is simple, the tree node contains its value; if the attribute is a list of other attributes, the tree node just serves as the parent to each of attributes in that list. The tree structure only reflects the necessarily hierarchical structure of definition. In this way, any attribute in DML can be referred to using a search path. This approach is analogous to naming the files and directories in a file system: an attribute is similar to a file and an attribute list is similar to a directory. Rather than using a slash "/" to separate files in a search path, DML uses a dot ".". For example, we can refer to the population number in the previous example using ".population.number". The first dot is to tell the DML parser that the search path should start from the tree root. Of course, one can use a relative search path that is not started with a dot, in which case the DML parser knows that the search path starts from the tree node representing the attribute list it is currently processing.

The standard DML specification defines three special attribute keys: `_schema`, `_find`, and `_extends`. The attribute key `_schema` supposedly is used to apply schema checking to a DML script. It is not implemented in PRIME SSF. The attribute key `_find` is used to point to another attribute that the DML parser needs to replace the current attribute with. For example, the following two DML scripts are the same:

```
today [
    date "08/08/2002"
    time "15:00:29 GMT"
]
planet Earth
weight 6.5e21
population [
    number 6242324585
    _find .today.date
    _find .today.time
]
```

```
today [
    date "08/08/2002"
    time "15:00:29 GMT"
]
planet Earth
  weight 6.5e21
  population [
    number 6242324585
    date "08/08/2002"
    time "15:00:29 GMT"
  ]
```

The attribute key `_extends` functions similar to `_find`; it is used to replace the current attribute with a list of attributes it is pointed to. When the same list of attributes is used many times in a DML script, using `_extends` can tremendously reduce the size of the DML file. The following shows another way of writing the same DML as in the previous example:

```
today [
    date "08/08/2002"
    time "15:00:29 GMT"
]
planet Earth
weight 6.5e21
population [
    number 6242324585
    _extends .today
]
```

6.2.2 DML Programming Interface

PRIME SSF implements a simple programming interface for the user to access DML attributes in a DML script. The interface provides methods for querying DML attributes according to its hierarchical tree structure. The DML programming interface is defined as follows:

```
class Configuration {
 public:
  virtual void* findSingle(char* path) = 0;
  virtual Enumeration* find(char* path) = 0;

  int isConf(void* path);
  char* getKey(void* path)
};

class dmlConfig : public Configuration {
 public:
  dmlConfig(char* filename = 0);
  dmlConfig(char** filenames);
  void load(char* filename);
  void load(char** filenames);

  virtual void* findSingle(char* p);
  virtual Enumeration* find(char* p);
};
```

The `Configuration` class represents the a node in the DML tree. It is an abstract class that defines the allowable operations to a DML script. The DML attributes in SSF is represented by the `dmlConfig` class, which is a derived from the `Configuration` class. An instance of the `dmlConfig` class represents a DML attribute, which is actually a node in the DML tree.

The constructors of the `dmlConfig` class build the DML tree from a DML file or a list of DML files stored in null-terminated array. The constructors actually invoke the DML parser to parse the DML file. The `load` methods can also be used to build the DML tree from a given DML file or files. The user can start with an empty DML tree (obtained if the constructor is called without an argument) and populate the tree using the `load` method.

The `findSingle` method is used to locate the first attribute pointed to by the given search path `path`. As mentioned early, the search path are attribute keys separated by dots. A search path can be either absolute or relative. An absolute search path starts with a dot "." and the search is expected to start from the root of the DML tree. A relative search path starts from the current DML tree node. A null will be returned if no match is found. The type of the returned value depends on the value of the attribute found. If the return value is an attribute list, the returned value should be cast to a `Configuration` type. Otherwise, the method returns a pointer to a character string and the user needs to convert the string to an appropriate type. For example, if the attribute value is an integer, the user can call `atoi` function to convert the returned the string to an integer.

The `find` method returns an enumeration of all DML attributes that match the search path. An enumeration is represented by the `Enumeration` class, an abstract class that defines two methods shown in the following:

```
class Enumeration {
```

```
    virtual int hasMoreElements() = 0;
    virtual void* nextElement() = 0;
  }
```

An enumeration object contains a list of elements. The user first needs to call the `hasMoreElements` method to find out whether the enumeration has reached the end. If there is more elements, the method returns true and the element can be retrieved by the nextElement method. Note that the return value is just a generic pointer (`void*`). The user needs to cast this pointer to the right type. If the attribute pointed to by the search path is another attribute list, the returned element should be cast to a `Configuration` type. Otherwise, it is a pointer to a character string and the user needs to convert the string to an appropriate type.

Problem arises if the user does not know *a priori* whether the search path ends with a simple attribute value or another list of attributes. In the former case the user needs to cast it to a string and in the latter case the user needs to treat it as a pointer to a `Configuration` type. Standard DML specification does not provide a solution. PRIME SSF offers the *static* `isConf` method to help the user determine whether a generic pointer returned by the `findSingle` method or an item in the enumeration structure returned by the `find` method is a list of DML attributes and therefore should be cast to a `Configuration` type. The following example shows that the user can use this function to check whether the DML script conforms to the expectation:

```
char* planet = (char*)cfg->findSingle(''planet'');
if(Configuration::isConfig(planet)) {
  printf(''ERROR: 'planet' should be simple\n'');
  return;
}

Configuration* popcfg = (Configuration*)cfg->find(''population'');
if(Configuration::isConfig(popcfg)) {
  printf(''ERROR: 'population' should contain a list of attributes\n'');
  return;
}
```

PRIME SSF also allows the user to search for attributes with a search path containing wildcards "?" and "*". For example, if the user wants to locate all DML attributes in the current attribute list with a key that starts with "abc". The user can call `find(''abc*'')` to get an enumeration of all matched attributes. Now the problem is how to find out the exact key of each of the elements. To solve this problem, we add another method called `getKey`, which returns the key of the attribute returned by the `findSingle` method or an enumerated element in the enumeration structure returned by the `find` method.

6.3 SSF DML Model Description

PRIME SSF uses DML as an option to construct and initialize an SSF model. There are three DML files needed and they are listed as follows. The parsing of these domain-specific DML files is done by the runtime system and the model is built automatically before the simulation starts.

- Model DML File: describes the topology of the model. The model is composed of entities, possibly aligned with one another in a certain way. The parameters used to construct these entities are also given in the file. These entities are connected together via channels. The description of entities and mappings uses the external names of the derived `Entity` , `inChannel` , and `outChannel` classes.
- Machine DML File: specifies the hardware platform where the simulation runs. It lists machines that are used in the simulation as well as the number of processors on each machine.

- Runtime DML File: provides simulation runtime information. It contains simulation parameters, such as the start simulation time, the simulation termination time, the SSF model used by the simulation (i.e., the name of the model DML file), the machine platform it is running on (i.e., the name of the machine DML file), etc.

The three DMLs are domain-specific DML. PRIME SSF separates the concepts of model topology, machine environment, and simulation runtime. It is conceivable to have a situation where a model may be run on different machine platforms and with different simulation runtime parameters, in which case only the corresponding DML files will be replaced. The following describes these DML formats.

6.3.1 Model DML

The topology of an SSF model is described in the model DML format. An SSF model is a list of entities defined and organized in a hierarchical tree. Note that although the model is defined in a hierarchical style, its topology can be everything and not necessarily a tree structure. At the top level, the model definition starts from the `MODEL` attribute. The topology of the model is the value of the `MODEL` attribute, therefore is enclosed within `MODEL [...]`. The attributes in `MODEL` specify the list of entities (defined by `ENTITY`) or clusters of entities (defined by `CLUSTER`), together with the entity alignments and channel mappings.

Entities and clusters in the model DML file are identified by their unique `ID` attributes. The `ID` values are decoded starting from the root of the definition hierarchy. The absolute identity of an entity or a cluster can be obtained by concatenating `ID` values from the root (separated by "."). An absolute ID differs from the relative ID by a prefix ".". For example, the following model DML description defines three nested clusters and one entity:

```
MODEL [
  CLUSTER [                 # cluster ".0"
    ID 0
    CLUSTER [               # cluster ".0.1"
      ID 1
      CLUSTER [             # cluster ".0.1.2"
        ID 2
        ENTITY [ID 3 ...]   # entity ".0.1.2.3"
      ]
    ]
  ]
]
```

An entity in the model is specified with `ENTITY` attribute. The `ENTITY` attribute contains a list of attributes that specify information needed for constructing the entity:

- `ID` is the identity of the entity in the model DML file.
- `INSTANCEOF` is the name of the entity. The name should match the name of the entity defined in the model source code. The name is used by the modeler to register the derived entity class (using `SSF_REGISTER_ENTITY` macro).
- `PARAMS` is the list of parameters to be passed to the entity constructor. The number of the parameters and their respective types should match the definition of the derived entity class. If the constructor of the entity does not require any parameters; an empty list must be given: `PARAMS []`. A parameter can be one of three possible types: `INT`, `FLOAT`, or `STRING`. The parameters are given as a list of

attributes with the parameter types as attribute keys and parameter values as attribute values. Here is an example:

```
PARAMS [ INT 123 FLOAT 1.35e-4 STRING ‘‘this is a string’’ ]
```

There are four special parameter values that can be recognized by the model DML interpreter:

- `%N`: the total number of entities defined in the model DML (type `INT`).
- `%n`: the index of the entity in the order of its appearance in the DML file; it ranges between 0 and (`%N`)-1 (type `INT`).
- `%I`: the absolute entity ID (type `STRING`), e.g. “0.1.2.3”.
- `%i`: the relative entity ID (type `STRING`), e.g. “3”.

- `CONFIGURE` is a list of arbitrary DML attributes. The attribute is optional. If the `CONFIGURE` attribute is present and its list is not empty, the SSF runtime system will invoke the `config` method of the entity after it is constructed and pass to it the uninterpreted DML tree node (i.e., a pointer to a `Configuration` object).

The `CLUSTER` attribute specifies a cluster of entities and it may contain other clusters as well. The use of this attribute helps define the model in a hierarchical fashion. The `MODEL` attribute is semantically the same as the `CLUSTER` attribute, except that `MODEL` does not require `ID` attribute since it is unique being the root. A `CLUSTER` attribute contains a list of the following attributes:

- `ID` is the identity of the cluster in the model DML file.
- `CLUSTER` is the sub-cluster contained in this cluster. This attribute is optional.
- `ENTITY` is an entity in the cluster. This attribute is also optional.
- `ALIGN` describes alignment of the entities (usually defined in this cluster). Its value is a list of th following two attributes:
  - `FROM`: the identity of an entity that initiates the alignment operation. We call it the *align-from* entity.
  - `TO`: the identity of an entity that is the target of the alignment operation. We call it the *align-to* entity.

  SSF runtime system will break the align-from entity from its original timeline (if it is not independent) and make it share the same timeline of the align-to entity. The result is equivalent to calling the `alignto` method.
- `MAP` describes the mapping of an external local output channel of an entity to an external local input channel of another entity. This result is equivalent to calling the `mapto` method. The value of this attribute is a list of the following attributes:
  - `FROM` specifies the output channel of this mapping. The value of this attribute has the following format: `entity-identity(outchannel-name)`, where `entity-identity` is the entity’s identity defined in the model DML file, and `outchannel-name` is the name of the external local output channel of the entity. An external local output channel is created in the model either as a named output channel or using the `SSF_PUBLISH_DML_OUTCHANNEL` macro.

- TO specifies the input channel of this mapping. The value of this attribute has a similar format: entity-identity(inchannel-name). The entity-identity is defined in the model DML file, while inchannel-name is the name of the external local input channel of an entity. An external local input channel is an unnamed input channel published using the SSF_PUBLISH_DML_INCHANNEL macro.
- DELAY is the mapping delay. This attribute is optional. If it is omitted, the runtime system will assume the mapping delay is zero.

A simple model DML file can be flat if the user decides not to use the CLUSTER attribute. This might be a good choice at times when the user only needs to describe a simple model, or the model DML file is the result of a user pre-processor that automatically generates model topologies from some complicated parameters. The following example is a model DML that specifies a network of four entities connected as a ring. The entities share two timelines. In the source code (not shown), an entity class Queue must be defined and registered and the entity must contain one external output channel named myoc and one external input channel named myic.

```
MODEL [
  ENTITY [ID 0 INSTANCEOF "Queue" PARAM [INT %n]]
  ENTITY [ID 1 INSTANCEOF "Queue" PARAM [INT %n]]
  ENTITY [ID 2 INSTANCEOF "Queue" PARAM [INT %n]]
  ENTITY [ID 3 INSTANCEOF "Queue" PARAM [INT %n]]

  ALIGN [FROM 0 TO 1]
  ALIGN [FROM 2 TO 3]

  MAP [FROM 0(myoc) TO 1(myic) DELAY 1.0]
  MAP [FROM 1(myoc) TO 2(myic) DELAY 1.0]
  MAP [FROM 2(myoc) TO 3(myic) DELAY 1.0]
  MAP [FROM 3(myoc) TO 0(myic) DELAY 1.0]
]
```

To illustrate the benefit of using hierarchical structure to define models, we use the model DML file to define the topology of the muxtree example (see section 3). We here describe a simplified version of model. Since the topology of the muxtree is defined completely here using the model DML, we do not need to provide the tree level as an argument to the constructor of the MultiplexerEntity class. Also, we use an ID that is globally unique (rather than using the entity index within a tree level as the ID). The input and output channels are external *local* and therefore do not need to be globally unique. We simply name them IC and OC. The root of the tree is an entity from the MultiplexerEntity class (with an ID of ".8"). The root is connected to eight sub-trees represented by clusters named ".0" to ".7" (assuming the muxtree has a fan-in of 8). Each of the sub-trees contains eight entities from the SourceEntity class and one entity from the MultiplexerEntity class. In this case, the muxtree has only two levels, for the sake of easy exposition. If more levels are defined, it should be immediately clear that the model description can be significantly reduced by nesting the definitions hierarchically.

```
network [
  source [
    INSTANCEOF "SourceEntity"
    PARAMS [ INT %n ]
  ]
```

```
    multiplexer [
      INSTANCEOF "MultiplexerEntity"
      PARAMS [ INT %n ]
    ]

    firstlevel [
      ENTITY [ID 0 _extends .network.source]
      ...
      ENTITY [ID 7 _extends .network.source]
      ENTITY [ID 8 _extends .network.multiplexer]

      ALIGN [FROM 0 TO 8]
      ...
      ALIGN [FROM 7 TO 8]

      MAP [FROM 0(OC) TO 8(IC) DELAY 1.0]
      ...
      MAP [FROM 7(OC) TO 8(IC) DELAY 1.0]
    ]
  ]

  MODEL [
    CLUSTER [ID 0 _extends .network.firstlevel]
    ...
    CLUSTER [ID 7 _extends .network.firstlevel]
    ENTITY [ID 8 _extends .network.multiplexer]

    ALIGN [FROM 8 TO 0.8]

    MAP [FROM 0.8(OC) TO 8(IC) DELAY 1.0]
    ...
    MAP [FROM 7.8(OC) TO 8(IC) DELAY 1.0]
  ]
```

6.3.2 Machine DML

Machine DML specifies the hardware platform on which the simulation is expected to run. It contains information how the simulation model is expected to be partitioned and how the runtime system is expected to be set up. This information is only applicable if we use MPI for distributed simulation. This machine DML file is used to create a machine profile for the MPI program. And even so, the information may be useless since some MPI implementations do not allow the user to provide the machine profile.

The machine DML file contains a list of one or more `MACHINE` attributes. Each `MACHINE` attribute specifies one node in a distributed system. A node can be either a uniprocessor machine or a shared-memory machine with multiple processors. The value of the `MACHINE` attribute is a list of the following attributes:

- `IPADDR` is the IP address of the distributed node.
- `ARCHITECTURE` is the machine type of the distributed node. The information provided by this

attribute is actually not used by the runtime system.

- `PARTITION` specifies the number of processors on the distributed node. The default of value of this attribute is one.

An example of the machine DML file is as follows. The hardware system consists of four dual-processor Linux workstations:

```
MACHINE [IPADDR mach1.xyz.com ARCHITECTURE linux PARTITION 2]
MACHINE [IPADDR mach2.xyz.com ARCHITECTURE linux PARTITION 2]
MACHINE [IPADDR mach3.xyz.com ARCHITECTURE linux PARTITION 2]
MACHINE [IPADDR mach4.xyz.com ARCHITECTURE linux PARTITION 2]
```

6.3.3 Runtime DML

Runtime DML specifies the simulation runtime parameters:

- `EXECUTABLE` is the name of the target executable file.
- `MODEL` is the name of the model DML file.
- `MACHINE` is the name of the machine DML file.
- `STARTTIME` is the start simulation time.
- `ENDTIME` is the end simulation time.
- `ENVIRONMENT` specifies the runtime environment variables. This attribute is optional and, if present, should contain a list of attributes, where the keys are environment variables and the values are the values of the environment variables (of string type). In simulation, the model can query for the value of a specific environment variable (using the `Entity::getenv` method).
- `DATAFILE` is the *prefix* to the name of the data output file. This attribute is optional. If omitted, the system will use temporary files and delete them when the simulation finishes. If the attribute is present, these data files will be kept after the simulation. There is one data output file assigned to a processor used for the simulation run. For example, if the system contains three dual-processor machines and if `DATAFILE` is `out`, six output files will be created: `out-0-0`, `out-0-1`, `out-1-0`, `out-1-1`, `out-2-0`, and `out-2-1`. The data written out by the user are stored in these output files, which can be retrieved and analyzed at the end of the simulation or after the simulation run. This attribute is used if the user decides to keep the output files for postmortem analysis.

The following is an example of a runtime DML file:

```
EXECUTABLE    "muxtree"
MODEL         "model.dml"
MACHINE       "machine.dml"
STARTTIME     0
ENDTIME       100
ENVIRONMENT [
  author      "John Doe"
  date        "unknown"
]
DATAFILE      "data/out"
```

The final executable file is named `muxtree`. The name of the model DML file and the name of the machine DML file are `model.dml` and `machine.dml` respectively. The simulation will run from 0 to 100 simulation time units. Two environment variables are defined and the user can query for their values using the `Entity::getenv` method in the simulation program. The statistics data output will be created under `data` directory and its name prefix is `out`.

7 Extended Utilities

PRIME SSF provides a number of extensions to the standard SSF API. These extensions are not required for developing an SSF model, but they can ease the development of the simulation model and in many cases increase the performance of the simulator. We have seen some of the extensions previously. In this section, we describe in detail these extensions. All these extensions are embedded in the SSF API and the user does not need to include separate header files in order to use these extensions in the simulation program.

7.1 Random Number Generation

Having a good random number generator is an important aspect of any stochastic simulator. PRIME SSF provides good random number generation capabilities with a library that consists of more than forty different random number generators and approximately a dozen common probability distributions. These random number generators are ported from three different sources:

DaSSF Random Number Generators. For compatibility reasons, we continue to include the Lehmer random number generator used by DaSSF as default. The generator is a linear congruential pseudo random number generator that uses a multiplier of 48271 and a modulus of $2^{31} - 1$. The generator has 256 random streams (with a jump multiplier being 22925). For details, consult the book "Discrete-Event Simulation: a First Course", by Lawrence Leemis and Steve Park, published by Prentice Hall in 2005.

The Mersenne Twister random number generator was also used in DaSSF. The generator has a period of $2^{19937} - 1$; it gives a sequence that is 623-dimensionally equidistributed. The code is modified from that of Richard J. Wagner, which is based on the code written by Makoto Matsumoto, Takuji Nishimura, and Shawn Cokus. For more information, go to the inventors' web page at `http://-www.math.keio.ac.jp/~matumoto/emt.html`.

Random Number Generators from the SPRNG Library. We include the random number generators of the SPRNG library (version 2.0), developed by Michael Mascagni et al. at Florida State University (`http://sprng.cs.fsu.edu`), which features 6 classes of random number generators:

1. Combined Multiple Recursive Generator (CMRG): the period of this generator is around 2^{219}; the number of distinct streams available is over 10^8.
2. 48-Bit Linear Congruential Generator with Prime Addend (LCG): the period of this generator is 2^{48}; the number of distinct streams available is of the order of 2^{19}.
3. 64-Bit Linear Congruential Generator with Prime Addend (LCG64): the period of this generator is 2^{64}; the number of distinct streams available is over 10^8.
4. Modified Lagged Fibonacci Generator (LFG): the period of this generator is $2^{31}(2^k - 1)$ where k is the lag; the number of distinct streams available is $2^{[31(k-1)-1]}$.
5. Multiplicative Lagged Fibonacci Generator (MLFG): the period of this generator is $2^{61}(2^k - 1)$, where k is the lag; the number of distinct streams available is $2^{[63(k-1)-1]}$.

6. Prime Modulus Linear Congruential Generator (PMLCG): the period of this generator is $2^{61} - 2$; the number of distinct streams available is roughly 258. This generator is not included in our library as it requires the GNU multi-precision arithmetic library (`libgmp.a`).

For more information, we encourage the user to consult the SPRNG 2.0 User's Guide.

Random Number Generators from the BOOST Library. We also include the random number generators implemented in the Boost library (`http://www.boost.org/libs/random`). The performance of these generators are listed on the boost web site. The important numbers are copied here for reference (for speed, larger means faster):

generator	period	memory	speed
`MINSTD_RAND0`	$2^{31} - 2$	`sizeof(int32_t)`	40
`MINSTD_RAND`	$2^{31} - 2$	`sizeof(int32_t)`	40
`RAND48`	$2^{48} - 1$	`sizeof(uint64_t)`	80
`ECUYER1988`	$\sim 2^{61}$	`2*sizeof(int32_t)`	20
`KREUTZER1986`	?	`1368*sizeof(uint32_t)`	60
`HELLEKALEK1995`	$2^{31} - 1$	`sizeof(int32_t)`	3
`MT11213B`	$2^{11213} - 1$	`352*sizeof(uint32_t)`	100
`MT19937`	$2^{19937} - 1$	`625*sizeof(uint32_t)`	100
`LF607`	$\sim 2^{32000}$	`607*sizeof(double)`	150
`LF1279`	$\sim 2^{67000}$	`1279*sizeof(double)`	150
`LF2281`	$\sim 2^{120000}$	`2281*sizeof(double)`	150
`LF3217`	$\sim 2^{170000}$	`3217*sizeof(double)`	150
`LF4423`	$\sim 2^{230000}$	`4423*sizeof(double)`	150
`LF9689`	$\sim 2^{510000}$	`9689*sizeof(double)`	150
`LF19937`	$\sim 2^{1050000}$	`19937*sizeof(double)`	150
`LF23209`	$\sim 2^{1200000}$	`23209*sizeof(double)`	140
`LF44497`	$\sim 2^{2300000}$	`44497*sizeof(double)`	60

Each instance of `Random` class represents an independent random number stream. The constructors of the `Random` class are declared as follows:

```
Random(int type = USE_RNG_LEHMER, int seed = 0);
Random(int type, int idx, int n, int seed = 0);
```

The first constructor creates a random number stream started from the given initial seed. The user can choose any of more than forty different random number generator types defined by the library. The default is the Lehmer random number generator originally used by DaSSF. If the seed is omitted or zero, the random stream is seeded by the system clock, in which case the user may not be able to reproduce the random stream in a separate simulation run.

The second constructor chooses a random stream among multiple random streams. For a given random number generator type, it is expected that multiple random streams provide sufficient separation (i.e., with minimal correlation) between random numbers drawn from separate random streams. The user should consult the particular random number generator for the maximum number of random streams that can be used simultaneous and still generate acceptable result. The user is expected to provide the same random seed (other than zero) to start multiple random streams.

Depending on the way how the random seed is chosen, the system can have different level of independence. For example, the initial seed could be a function of the entity ID. In that case, stochastic processes within an entity (using the same *Random* instance) should be correlated in a way that random numbers are

drawn from the same random stream. On the other hand, the random stream generated should be independent of the entity construction process, which depends on the implementation of SSF. So that the simulation can still be repeatable across different implementations. Actually, in the runtime system, the user-provided seed is mingled with a random number drawn from a system-wide random stream. The system-wide random stream is initially seeded by the command-line option `-seed`. Thus, different random streams can be generated for different simulation runs. Yet the simulation can be repeatable if the same initial seed is provided for the system-wide random stream.

Methods of the `Random` class also provide samples from various distributions. The methods are listed as follows:

```
/* continuous distributions */
double uniform();
double uniform(double a, double b);
double exponential(double x);
double erlang(long n, double x);
double pareto(double k, double a);
double normal(double m, double s);
double lognormal(double a, double b);
double chisquare(long n);
double student(long n);

/* discrete distributions */
long bernoulli(double p);
long equilikely(long a, long b);
long binomial(long n, double p);
long geometric(double p);
long pascal(long n, double p);
long poisson(double m);
```

These distributions are described below. Integer parameters (shown below in capital letters) are of type long; floating point parameters (in lower-case letters) are of type double.

Distribution	Description
uniform()	uniform distribution over $(0, 1)$.
uniform(A, B)	uniform distribution over $[A, B]$.
exponential(r)	mean $1.0/r$
erlang(N,x)	N-stage Erlang, each stage has rate x
normal(m,s)	mean m and variance s^2
lognormal(a,b)	mean $\exp(a + 0.5 * b^2)$, variance $(\exp(b^2) - 1) * \exp(2a + b^2)$
chisquare(N)	mean N, variance $2n$
student(N)	mean 0 ($N > 1$), variance $N/(N - 2)$, ($N > 2$)
bernoulli(p)	mean p, variance $p(1 - p)$
equilikely(a,b)	mean $(a + b)/2$, variance $((b - a + 1) * (b - a + 1) - 1)/12$
binomial(N,p)	mean $N * p$, variance $N * p * (1 - p)$
geometric(p)	mean $1/p$, variance $(1 - p)/(p * p)$
pascal(N, p)	mean N/p, variance $N * (1 - p)/(p * p)$
poisson(r)	mean r, variance r

Moreover, the `Random` class defines two methods that can help give a permutation of n numbers (indexed from 0 to $n - 1$). The user can either provide the pre-allocated array or the system will create the

array for the user that contains the permutation.

```
void permute(long *array, long n);
long *permute(long n);
```

7.2 Semaphores

It is frequently the case that one simulation process needs to signal another (within the same entity), without passing data. In the standard SSF API, this can be accomplished only through internal channels: two channels—one output channel and one input channel—are mapped together and events are sent from the signaling process to the waiting process (see the muxtree example in section 3). This situation could become complicated when there are more than two processes involved or if one wishes for that signal to be "buffered".

As this is a case frequently occurred in the design of a simulation model, PRIME SSF supports its own semaphore class to facilitate the design:

```
class ssf_semaphore {
 public:
  ssf_semaphore(int initval = 1, Entity* owner = 0);
  ~ssf_semaphore();
  void semWait();
  void semSignal();
  int  semValue();
};
```

The constructor establishes the initial value of the semaphore, passed as an argument, or uses the default value of 1. Presumably, a semaphore is used as a state variable of an entity and therefore should be declared with an entity as its owner. The establishment of this ownership can be delay upon the first time when the semaphore is used. A call to the `semWait` method decrements the semaphore, blocks the caller if the result is negative, and appends the blocking process to the end of the semaphore's list of waiting processes. A call to the `semSignal` method increments the semaphore, and if the result is zero, unblocks the first process in the waiting list, if any. Processes are thus released from the semaphore in the order in which they were blocked upon it. The `semValue` method returns the current value of the semaphore.

Processes that coordinate through a semaphore must belong to the same entity, or entities that are coaligned in the same timeline. Also, the `semWait` method must be called within a process. Since the process could be suspended from further execution due to the method call, `semWait` is actually a wait statement.

7.3 Timers

PRIME SSF provides a notion of a "timer". Basically, a timer allows the modeler to schedule function invocations at a future simulation time. In addition, such function invocations can be cancelled if the condition is changed.

In PRIME SSF, a timer is represented as an `ssf_timer` object. A timer is associated with an entity object. It encapsulates a callback function, which can either be an `Entity` method or a C++ regular function, with a designated function signature. The timer can be scheduled to fire off at a future simulation time. It can also be cancelled between the time of the scheduling and the time the timer goes off. If the timer goes off, the system will call the `Entity` callback method with a pointer to the timer as its parameter. In this way, the user can design a derived class from the `ssf_timer` class to carry user data, so that when the timer goes off the callback function will be able to extract the data from the argument.

```
class ssf_timer {
 public:
  ssf_timer(Entity*, void (Entity::*)(ssf_timer*));
  ssf_timer(Entity*, void (*)(ssf_timer*));
  virtual ~ssf_timer();

  void schedule(ltime_t delay);
  void reschedule(ltime_t delay);
  void cancel();

  int is_untouched();
  int is_cancelled();
  int is_finished();
  int is_running();

  Entity* owner();
  ltime_t fire_time();
};
```

The callback function is set at the time when the timer is constructed. A timer belongs to an entity, whose method is the callback function. The ownership as well as the callback function are immutable during the life time of the timer object. The life cycle of a timer consists of four possible states. When the timer is first constructed, it is "untouched". The user can schedule the timer to fire off in the simulated future by calling the `schedule` method with a specified delay. After the call, the timer is regarded as "running". If the timer is already running, the call to `schedule` method is an error. However, the timer firing can be cancelled by calling the `cancel` method. If the timer is not running, the method call is ignored. A cancelled timer is in the state of "cancelled" and is ready to be scheduled again. A quick way to change a pre-scheduled timer is to use the `reschedule` method. The method will cancel any previous scheduled invocation if the timer is running and reschedule the timer to fire off in the simulated future. The user can delete the timer only when it's not running. Usually, the user can only do so when the callback method is called.

There are methods in *ssf_timer* to query the current state as well as the owner entity of the timer. The method `fire_time` returns the scheduled fire time of the timer if the timer is running. Otherwise, the return value is undefined.

7.4 Data Serialization

There are at least two situations where data serialization is needed. When an instance of a user-defined event class must be sent across memory space boundaries, the runtime system must transform the event object into a machine-independent byte stream and send it via message passing mechanisms. The runtime system requires that the user takes the responsibility of packing the data fields of the derived event class when the event is about to be shipped to a remote machine. As we mentioned in section 4.7, the class derived from `Event` should provide the virtual method `pack`, within which the user transforms the event object into a byte stream represented as the `ssf_compact` class. As the object is shipped to the destination, the user should be able to unpack the data into a newly created event object.

Another situation for data serialization is when we need to collect statistics from various entities spread across the entire system (particularly in a distributed simulation environment). Statistics collection is described in detail in the next section. To avoid situations where the system is composed of machines with heterogeneous architectures, the data dumped for later analysis must first be serialized into a byte stream.

When the user later analyzes the data, the byte stream is read from the file system and then unpacked. The transformation is machine architecture independent.

```
class ssf_compact {
 public:
  ssf_compact();
  virtual ~ssf_compact();

  /* methods for packing a single primitive value */
  void add_float(float val);
  void add_double(double val);
  void add_long_double(long double val);
  void add_char(char val);
  void add_unsigned_char(unsigned char val);
  void add_short(short val);
  void add_unsigned_short(unsigned short val);
  void add_int(int val);
  void add_unsigned_int(unsigned int val);
  void add_long(long val);
  void add_unsigned_long(unsigned long val);
  void add_long_long(SSF_LONG_LONG val);
  void add_unsigned_long_long(SSF_UNSIGNED_LONG_LONG val);
  void add_ltime(ltime_t val);

  /* methods for packet an array of primitive values */
  void add_float_array(int nitems, float* val_array);
  void add_double_array(int nitems, double* val_array);
  void add_long_double_array(int nitems,
    long double* val_array);
  void add_char_array(int nitems, char* val_array);
  void add_unsigned_char_array(int nitems,
    unsigned char* val_array);
  void add_short_array(int nitems, short* val_array);
  void add_unsigned_short_array(int nitems,
    unsigned short* val_array);
  void add_int_array(int nitems, int* val_array);
  void add_unsigned_int_array(int nitems, unsigned int* val_array);
  void add_long_array(int nitems, long* val_array);
  void add_unsigned_long_array(int nitems,
    unsigned long* val_array);
  void add_long_long_array(int nitems, SSF_LONG_LONG* val_array);
  void add_unsigned_long_long_array(int nitems,
    SSF_UNSIGNED_LONG_LONG* val_array);
  void add_ltime_array(int nitems, ltime_t* val_array);

  /* pack a (null-terminated) string */
  void add_string(const char* valstr);
```

```
  /* methods for unpacking an array of primitive values */
  int get_float(float* addr, int nitems = 1);
  int get_double(double* addr, int nitems = 1);
  int get_long_double(long double* addr, int nitems = 1);
  int get_char(char* addr, int nitems = 1);
  int get_unsigned_char(unsigned char* addr, int nitems = 1);
  int get_short(short* addr, int nitems = 1);
  int get_unsigned_short(unsigned short* addr, int nitems = 1);
  int get_int(int* addr, int nitems = 1);
  int get_unsigned_int(unsigned* addr, int nitems = 1);
  int get_long(long* addr, int nitems = 1);
  int get_unsigned_long(unsigned long* addr, int nitems = 1);
  int get_long_long(SSF_LONG_LONG* addr, int nitems = 1);
  int get_unsigned_long_long(SSF_UNSIGNED_LONG_LONG* addr,
    int nitems = 1);
  int get_ltime(ltime_t* addr, int nitems = 1);

  /* unpack a (null-terminated) string */
  char* get_string();

  /* static methods for serializing primitive values */
  static void* serialize(prime::ssf::uint8 data,  void* buf = 0);
  static void* serialize(prime::ssf::uint16 data, void* buf = 0);
  static void* serialize(prime::ssf::uint32 data, void* buf = 0);
  static void* serialize(prime::ssf::uint64 data, void* buf = 0);
  static void* serialize(prime::ssf::int8 data,   void* buf = 0);
  static void* serialize(prime::ssf::int16 data,  void* buf = 0);
  static void* serialize(prime::ssf::int32 data,  void* buf = 0);
  static void* serialize(prime::ssf::int64 data,  void* buf = 0);
  static void* serialize(float data,  void* buf = 0);
  static void* serialize(double data, void* buf = 0);

  /* static method for deserializing primitive values */
  static int deserialize(prime::ssf::uint8& data,  void* buf);
  static int deserialize(prime::ssf::uint16& data, void* buf);
  static int deserialize(prime::ssf::uint32& data, void* buf);
  static int deserialize(prime::ssf::uint64& data, void* buf);
  static int deserialize(prime::ssf::int8& data,   void* buf);
  static int deserialize(prime::ssf::int16& data,  void* buf);
  static int deserialize(prime::ssf::int32& data,  void* buf);
  static int deserialize(prime::ssf::int64& data,  void* buf);
  static int deserialize(float& data,  void* buf);
  static int deserialize(double& data, void* buf);
};
```

The definition of `ssf_compact` class is shown above. The non-static methods are divided into methods for packing and methods for unpacking. At first, a new `ssf_compact` object is constructed. Data can be added to the object by calling `add_*` methods, one at a time. The `add_*_array` methods could also be

used, in which case the user calls the methods with an array of elements of the same primitive type and adds them to the object. The size of the array needs to be provided as the first parameter. After serializing the data, the object is given to the runtime system, which will pack the data stored in the object into a byte stream in a machine-independent fashion, before it is either sent to a remote machine or written to a file. At time when the user needs to retrieve the data, the runtime system will deserialize the data into an `ssf_compact` object and hand it to the user. The user will then call `get_*` methods to retrieve data from the object.

The data in `ssf_compact` is stored in a first-in-first-out order. That is, the first value added to the object will be retrieved first. Also, the `ssf_compact` class assumes the user understands the exact sequence of the data packed inside the object; retrieving the data of a unmatched type results in an error.

The static methods in the `ssf_compact` class are used to serialize (i.e., pack) a variable of a particular type into a byte array and deserialize (i.e., unpack) the data store in the byte array into a variable of a particular type. These methods provide a way for the user to translate between different data representations. PRIME SSF uses little endian as the machine independent format.

7.5 Statistics Collection

The data output files are managed by the runtime system. Each processor is designated with one such file to store the data records written out the user. The benefit of this approach is that the user does not need to concern about the file operations. Instead, he or she only needs to use the `Entity::dumpData` method to create a data record when needed. This method creates a user-defined record in the system data output file. The record will be marked with the entity's internal serial number and a simulation timestamp this record is created. A user-defined type is given as the first parameter of this method. It is to distinguish records of different types when the records are retrieved. The user data must be packed into a `ssf_compact` object (discussed in the previous section) and provided as the second parameter of this method.

The user can generate an arbitrary number of records for an entity during the simulation. It is often the case though that the user may want to dump the statistics collected during the simulation at an entity at the very end of the simulation. This is done by calling the `dumpData` method in the entity's `wrapup` method. When the simulation ends and the runtime system starts to reclaim entities, the `wrapup` method of each entity will be called before the entity is reclaimed. The following is taken from the muxtree example:

```
void MultiplexerEntity::wrapup() {
  ssf_compact* dp = new ssf_compact;
  dp->add_long(nrecv);
  dp->add_long(nlost);
  dp->add_long(nsent);
  dumpData(1, dp);
  delete dp;
}
```

The variables `nrecv`, `nlost`, and `nsent` are used to record the number of packets that are received by, lost at, or sent from the entity that models a multiplexer. The modeler decided to create a record that contains these variables for each of the multiplexer entities in the model, so that, at the end of the simulation, the totals could be computed.

If the main function is defined in the simulation program, after the `Entity::joinAll` method returns, the user is given the opportunity to collect and report simulation statistics. If the main function is omitted, the user can instead specify a global wrapup function, which is called at the end of the simulation, where dumped records of all entities in the model can be retrieved and analyzed. PRIME SSF provides a macro to set the wrapup function. The macro should be used at the place immediately after the global wrapup function is defined. And there can be at most one such function in the whole model.

```
SSF_SET_GLOBAL_WRAPUP(address-of-global-wrapup-function)
```

In the global wrapup function, the user can use the `retrieveDataDumped` method to retrieve the records written by all entities in the model, one at a time. The method returns an enumeration of the `ssf_entity_data` objects, which is used to encapsulate records generated by all entities of the model throughout the simulation. The `ssf_entity_data` class is a wrapper for a `ssf_compact` object and contains additional information about the user-defined data record. It includes the type of the record defined by the user (when the `dumpData` method is called), the simulation time when the record is written, and the serial number of the entity from which the record is generated. The definition of the `ssf_entity_data` class is as follows:

```
class ssf_entity_data {
 public:
  int source();
  int type();
  ltime_t date();
  ssf_compact* data();
};
```

The class contains methods that are used to retrieve the information about the user record. The `source` method returns the serial number of the source entity that creates the record. The `type` method returns the integer valued type associated with the record. The `date` method returns the simulation timestamp of the record. The record itself, of an ssf_compact type, can be retrieved by calling the `data` method. We have shown that the muxtree example (in section 3) uses a global wrapup function to sum the total number of packets received, lost, and sent during the simulation.

7.6 Machine Configurations

Sometimes the application needs to be aware of the configurations of the machine on which the simulation runs. For instance, in the muxtree example, the program needs to be aware of the total number of distributed machines that run the simulation and the current machine index, in order for the simulation program to partition the muxtree among these distributed machines. PRIME SSF provides a list of functions related to the hardware environment. These functions can be called everywhere in the program even if the simulation is running in the stand-alone mode.

```
extern int ssf_num_machines();
extern int ssf_machine_index();
extern int ssf_num_processors();
extern int ssf_num_processors(int machidx);
extern int ssf_total_num_processors();
extern int ssf_processor_index();
extern int ssf_total_processor_index();
extern int ssf_processor_range(int& startidx, int& endidx);
```

The `ssf_num_machines` returns the number of distributed machines that run the simulation. The distributed machines are indexed from zero to the total number of machines minus one. The `ssf_machine_index` method returns the machine index. In cases where there are multiple processors on each distributed machine, the `ssf_num_processors` method can be used to tell the total number of processors on the current machine. If the method is called with an integer argument, the function returns the number of processors on the specified machine. The processors are indexed both globally for the entire simulation environment and

locally on each machine. The `ssf_processor_index` function returns the index of the processor on the local machine, while the `ssf_total_processor_index` returns the global index of this processor. The `ssf_processor_range` method gets the global index range of the processors on the local machine and also returns the global processor index of the running processor.

7.7 Shared-Memory Synchronization

PRIME SSF provides the user with a barrier synchronization mechanism on shared memory, which contains a set of functions for the user to synchronize processes on shared memory. These functions can be used to combine an event-driven simulation with a time-driven simulation, supporting simulation applications where both discrete-event simulation and time-stepping simulation are necessary to model the system. These functions are also helpful for an application to obtain a snapshot of the simulation. In parallel simulation, different parts of the runtime system may advance their simulation clocks asynchronously. SSF API dictates that no direct memory access is allowed cross timeline boundaries. In particular, entities on different timelines cannot access other's state variables.

To this end, PRIME SSF provides a simple mechanism that allows the user to set a barrier synchronization point at a specified simulation time among processors on the same shared-memory. Processors sharing the same memory space will halt at the barrier and a callback function will be invoked by the runtime system on each processor. Inside the callback function, the user can access the global simulation state shared by the processors on the local machine. Note that the barrier is local to the shared-memory machine. The user is responsible to synchronize with the distributed machines (e.g., using the `MPI_Barrier` function inside the callback function).

```
long ssf_shmem_barrier_set(void(*fct)(long, ltime_t, void*),
  ltime_t tm, void *dat, boolean recur=false);
boolean ssf_shmem_barrier_cancel (long handle);
```

The `ssf_shmem_barrier_set` method is used to schedule a barrier at the specific simulation time or with the specified simulation time interval if the barrier is defined as recurring. Upon reaching the barrier, all processors sharing the same shared memory will be blocked and the callback function will be invoked only on processor 0. When the callback function returns, the barrier will be released and processors can then continue their execution. There can be multiple synchronization barriers defined in the simulation program and they are distinguished by the returned long integer handle. A barrier can by cancelled using the `ssf_shmem_barrier_cancel` method using the handle.

In order to maintain global information of the simulation run, PRIME SSF provides methods for accessing global variables. A global object is a memory block in the shared memory with, which a unique name. A global object can be shared by multiple processors on the same shared memory.

```
void ssf_set_global_object(char* objname, void* obj);
void* ssf_get_global_object(char* objname);
```

The `ssf_set_global_object` method is used to attach a name to a memory block. The name should be globally unique among all global objects on the current shared memory. The `ssf_get_global_object` method is used to retrieve the global object with the given name. Accessing global variables may create contentions if they are to be shared and accessed by multiple processors on the shared memory. To solve this problem, PRIME SSF provides synchronization primitives for mutual exclusive access to the shared memory blocks. An `ssf_mutex` type is defined and can be used in the following functions to ensure exclusive access to global variables.

```
extern void ssf_init_mutex(ssf_mutex* lock);
extern void ssf_mutex_wait(ssf_mutex* lock);
extern void ssf_mutex_signal(ssf_mutex* lock);
extern int  ssf_mutex_try(ssf_mutex* lock);
```

The `ssf_mutex_init` method initializes the mutex variable (or the lock), which is in an unlocked state. The `ssf_mutex_wait` method acquires the lock. The process will be suspended if it is already locked. The `ssf_mutex_signal` method releases the lock, which will unblock another process waiting on the same lock. The `ssf_mutex_try` method attempts to acquire the lock. If it is unlocked, the process acquires the lock and the function returns one. Otherwise, the function returns zero indicating that the lock acquisition is unsuccessful.

The above functions are provided to the user on an experimental basis. Blocking a process this way does not involve the simulation runtime timeline scheduling mechanism and therefore could result in a deadlock if it is not handled properly. These function are therefore only recommended to advanced users.

8 Real-Time Simulation Support

PRIME SSF provides real-time simulation support allowing the simulator to interact with real applications in real time. The real-time simulation support is designed as an extension to the standard SSF API. We extend the concept of input and output channels using them as the conduit for the simulator to send and receive events from outside the simulator. In particular, an input (output) channel can be associated with a reader (writer) thread that is responsible for the conversion between (external) physical events and (internal) virtual events. In addition, the runtime system can dynamically throttle the emulation speed in relation to the wall-clock time.

8.1 Real-Time Input Channel and Reader Thread

For real-time simulation, an input channel can be used as a conduit for the simulator to receive physical events, converts them into virtual events, and then injects into the simulator. When real-time simulation support is enabled, the PRIME SSF provides the following methods in addition to the standard SSF input channel API:

```
class inChannel {
 public:
  ...
  inChannel(Entity* owner, void(*reader)(inChannel*, void*),
    void* ctxt=0, ltime_t dly=0);
  void putRealEvent(Event* evt, double d=0.0,
    void(*cb)(ltime_t evtime, void* dat)=0, void *dat=0);
  void putVirtualEvent(Event* evt, ltime_t d=0,
    void(*cb)(ltime_t evtime, void* dat)=0, void* dat=0);
};
```

The new constructor creates a special input channel that supports real-time interactions with an external device. In addition to that, the runtime system creates a reader thread to handle data from the external device. The user must provide a function that uses system blocking calls (such as `select`) to do the real job. Once the data is read from the real-time device, the user needs to translate the data into a simulation event and pass it to the simulator using the `putRealEvent` method. There is a mapping delay associated with the input channel: it is the delay between the user calls the `putRealEvent` method and the event finally shows up

in the input channel, as if it went through a regular output channel that maps to this input channel. The special input channel cannot be mapped from another output channel. A context pointer (`ctxt`) is provided in case the user wants to pass data to the reader thread once it is created. The data becomes owned by the thread function once this method is called and should not be altered by the caller afterwards.

Inside the simulation program, the real-time input channel functions exactly the same as any regular input channels. A process may wait on the input channel for events to arrive. From this perspective, the model does not need to know that the input channel is receiving events from the a physical device.

The reader thread starts from the function provided as the second argument to the constructor. The reader thread is responsible for receiving data from the external physical device or devices. Typically this is done by having the reader thread make a blocking system call and wait for the physical event to happen. For example, the reader thread can call the `select` function on a file descriptor that represents an incoming TCP connection from which the function is expected data to be sent to this machine. Once a connection is established, data is retrieved from the file (using the `read` function). It is then converted to a simulation event (i.e., a new simulation event is created to represent the data received). Now, there are two methods that the user can use to inject the event into the simulator.

The `putRealEvent` method can be used to insert the event into the simulator with a real-time delay. This method can only be called in the reader thread. The simulation runtime system inserts this event into the receiving timeline's event list, which will soon emerge in the user space as if it is a regular user event delivered from a regular output channel. The delay will be the sum of the mapping delay (which is in simulation time and is set when the input channel constructor is called) and the write delay (in real time, provided as the second argument of this method). The translation from the real wall-clock time to the simulation time is subject to the current emulation speed specified by the users, which we describe in the a following section. Conforming to the normal SSF event referencing rules, the event given as the argument is presented to the system and therefore no longer belongs to the user. Since SSF's reference counter scheme is not thread-safe, it is important that one does not keep a reference to the event (e.g., via the `Event::save` method). If one wants to maintain a copy of this event, an explicit copy of this event must be made before calling the `putRealEvent` method. If a callback function is provided, the callback function will be invoked at the exact simulation time when this event is injected to the system. Both the simulation time and the user data will be passed to the callback function, which is responsible for reclaiming the data afterwards.

In certain circumstances, the user may want to insert an event with a simulation time delay using the `putVirtualEvent` method. The user should anticipate that there will be some real time delay before an event can be inserted into the simulation system, which is largely due to the context switching time between the reader thread and the simulation (kernel) thread.

8.2 Real-Time Output Channel and Writer Thread

An output channel can be used as a conduit for the simulator to send physical events, which are converted from the simulation events from the simulator. When real-time simulation support is enabled, the PRIME SSF provides the following methods in addition to the standard SSF output channel API:

```
class outChannel {
 public:
  ...
  outChannel(Entity* owner, void (*writer)(outChannel*, void*),
    void* ctxt=0, ltime_t cd=0);
  Event* getRealEvent(double &rdly);
};
```

The new constructor creates a special output channel that supports real-time interactions with an external device. The runtime system will create a writer thread to write data to the external device. The user must provide a function to do the real job. In particular, the function will be used to translate simulation events into the real data. The simulation events are provided by the runtime system to the user via the `getRealEvent` method. A delay is associated with the event indicating the expect latency before the event shall be delivered to the external device. The special output channel cannot mapped to another input channel. A context pointer can be used to pass any user-defined data to the function (`writer` that starts the writer thread.

If the writer thread, the user is expected to call the `getRealEvent` method, which gets from the queue maintained by the simulator the next event waiting to be transferred to the external device. The writer thread may be blocked if there is no event in the queue. Different from the normal SSF event referencing rules, the returned event belongs to the user and the user is responsible to reclaim it afterwards. We do this because SSF's reference counter scheme is not thread-safe. A real-time delay is also returned from this method. The delay is in seconds and is the expected delay before this event affects the external device.

The returned events should be converted to a real data format before it is sent to the real application. For example, in a real-time network simulation application, if the simulator is exporting a TCP packet to a remote host, the simulation event that represents the TCP packet should be converted to a real TCP packet format before it is sent to the remote machine that accepts the TCP connection.

8.3 Emulation Entity

In real-time simulation, entities can associate their simulation time advancement with the wall-clock time. When real-time simulation support is enabled, the PRIME SSF provides the following methods in addition to the standard SSF entity API:

```
class Entity {
 public:
  ...
  Entity(boolean emu = false);
  ltime_t real_now();
  static void startAll(ltime_t endtime, double r = 1.0);
  static void startAll(ltime_t starttime, ltime_t endtime,
                       double r = 1.0);
  static void throttle(double r,
    boolean (*cond)(double T, double T0,
                    ltime_t t, ltime_t t0, void* ctxt)=0,
    void* ctxt = 0);
};
```

Using the new constructor, the user can specify whether an entity is an emulation entity, which means that the time advancement of this entity is tied up with the real wall-clock time. An entity with a special input channel and/or a special output channel that interacts with the physical device is an emulation entity, regardless of the setting by the user. If an entity is an emulation entity, its simulation time can be written as a function of the wall-clock time. And vice versa. While the current simulation time of the entity is returned by the `now` method, the current wall-clock time of the entity is returned by the `real_now` method. For example, if the simulation starts at real time T_0 and virtual time t_0, and if the current real time is T when the method is called, also if the emulation ratio is r, then the returned simulation time is $(T - T_0) \times r + t_0$.

The emulation ratio described previous dictates the emulation speed, which is defined by the number of simulation time ticks (of `ltime_t` type) allowed for each real-time second. The `startAll` methods are added with an optional argument to allow the user to specify the emulation ratio. The ratio specified in these

methods is only affecting the emulation entities. The default is one, which means the emulation runs in synchrony with the wall-clock time. That is, one simulation tick takes one second in real time. Presumably one can accelerate the emulation by choosing an emulation ratio that is greater than one. If the ratio is set to be ∞ (by using a large double-precision floating-point number like 1e38), the emulation will be equivalent to pure simulation: events are processed by the simulation runtime system as fast as possible. If the ratio is zero, the simulation simply halts. No event will be executed and the system will not advance the simulation clock.

The `throttle` method can be used by the user to adjust the emulation speed dynamically during simulation. Once an emulation execution ratio is set by this method, the runtime system will maintain this ratio until the throttling termination condition becomes true. That is, if the callback function `cond` is not NULL and when called the function returns true, the runtime system will reset the ratio back to the initial value set by the `startAll` method at the beginning of the simulation. Note that once the emulation ratio is set to be ∞, there will be no throttling back. The simulation runtime system will treat all events (including both external and internal events) as backlogs. Therefore, if one wants to "catch up" a simulation, the user needs to find a reasonable constant factor to speed up the emulation rather than using infinite.

9 Caveats

This section lists the problems that are most likely encountered in the development stage of a PRIME SSF model. They include both common compile-time and runtime errors that could result from mistakes in models easily overlooked by both inexperienced and experienced users. We hereby address these common problems as well as places in the model that require special attention. We stress the important issues in the model design process.

9.1 Source Code Instrumentation

In the beginning development stage of a PRIME SSF model, it is very likely the modeler would experience problems related to adding annotations. These annotations help the source-to-source translator to identify places at which special code must be inserted to support the handcrafted multithreading mechanism. Fail to properly add these annotations may cause cryptic compilation errors and mysterious runtime bugs.

- In the simulation program, `main` is actually a predefined macro, which is designed to rename the user main function. It is so that the main function defined by the runtime system becomes the starting function of the simulation program. The user's main function is renamed to some other name and is invoked by the runtime system after some initial setup. The caveat of this approach is that the user should not use the symbol `main` other than to define the main function in a canonical way:

  ```
  int main(int argc, char** argv) {
    ...
  }
  ```

  In particular, the user cannot use `main` as variable names or method names.

- `//! SSF PROCEDURE` or `//! SSF PROCEDURE SIMPLE` must be placed at the same line following the declaration of a procedure.

- `//! SSF PROCEDURE` or `//! SSF PROCEDURE SIMPLE` must be placed at the line immediately above where the procedure method is defined.

- Procedures must be declared (in class definition in header files) and defined (typically in source files) separately.
- The procedure of a simple process must be annotated with `//! SSF PROCEDURE SIMPLE`. The procedure of a non-simple process must use `//! SSF PROCEDURE`. The annotations must match with the type of the process that invokes the procedures.
- If the start procedure of process is an entity method, the method must use the follow function prototype:

```
void entity_method_name(Process*);
```

- A simple process must be simple. That is, there must be one and only one wait statement in any possible execution path and the wait statement must be the very last statement to execute before the process' starting procedure reaches the end the function..
- A local variable in a *non-simple* procedure must be declared as a state variable (annotated by `//! SSF STATE`), if the variable is to be used across process suspensions. It's worth mentioning that the boolean variable used to store the return value of the `waitOnFor` or `waitOnUntil` methods must be marked by `//! SSF STATE` as it logically spans across suspension. When in doubt, always declare the variable as a state variable.
- Remember that the scope of state variables is the entire procedure. Therefore, nested local variables should be avoided in procedures.
- The procedure state variables must always declared first at the beginning of a procedure, one variable per line, and they must not be initialized during declaration.
- Procedure state variables that are declared by `//! SSF STATE` will be replaced by the source-to-source translator inside the procedure body. That is, all symbols that match the name of the variable will be replaces no matter whether or not they are referring to the same variable syntactically. It is wise to make sure that a procedure state variable is uniquely defined in the procedure body.
- `//! SSF CALL` must be placed immediately above the line of a procedure call. There must be at most one procedure call at a line.
- It is not allowed to make a function call to a procedure method inside a regular (non-procedure) function. It is allowed however to make a regular function call inside a procedure body.

9.2 Event Referencing Rules

PRIME SSF imposes strict rules for referencing simulation events. It helps improve the efficiency of the simulator by allowing both user and the runtime system to share the events.

- When an event object is created, it is *owned* by the user. The user can almost do everything to this event, including modifying the content of the event or even reclaiming the event when necessary.
- When the event is given to the runtime system (e.g., by calling the `outChannel::write` method), the event becomes owned by the runtime system. This process is irreversible. When a user's callback function is invoked and the event is provided by the runtime system as an argument to the call (such as the callback function defined in the `ssf_timer` class), or the user explicitly retrieves the arrival

event(s) at an input channel (such as the `inChannel::activeEvents` method), the events are still owned by the runtime system. The user only has limited access right to a system-owned event. The user can inspect the event before the callback function returns or the process becomes suspended again. The user, however, cannot access an event owned by the system after process suspension, unless the user creates another reference to the event by calling the `Event::save` method.

- An event is read-only if it is not aliased (determined by a call to the `aliased` method). If you need to modify an aliased event, you should make an explicit copy of the event first.
- An event that has been saved cannot be reclaimed by the runtime system unless a matching number of `release` calls are made. The number of `release` calls must never be larger than the previous calls to the `save` method.
- It is required that the copy constructor as well as the `clone` method should be provided for any derived event class.

9.3 Entity and Event Registrations

In order to build a simulation model on a distributed platform using the SSF DML model specification, correct mapping must be established between the model DML and the class definitions in the source code. This is done in part by using those macros defined by PRIME SSF.

- Every derived entity class that is public should provide a factory callback method that can be used to create an instance of the entity from a list of parameters. The prototype of the factory method must be

```
static Entity* entity-factory-method(const ssf_entity_dml_param**);
```

- The public entity class must be registered with the runtime system to associate the name of the class with the factory callback method:

```
SSF_REGISTER_ENTITY(entity-name, address-of-entity-factory-method);
```

  The name of the entity is given as the first argument of the macro and it should match the entity name in the model DML file (i.e., the value of the `INSTANCEOF` attribute).

- A derived event class should override the `pack` method to help the kernel translate the event object into a machine-independent byte stream (represented by `ssf_compact`).
- A factory callback method must also be defined for the event class. The callback method is responsible for creating a new instance of the event class and deserialize the member data of this event from a byte stream. The prototype of the event factory callback method must be:

```
static Event* event_factory_method(ssf_compact*);
```

- A derived event class must use the `SSF_DECLARE_EVENT` macro to declare this event class with runtime system in the class definition and use the `SSF_REGISTER_EVENT` macro to associate the event class with the factory method in the source code.