sc2 Simulation and Runtime Implementation

Los Alamos National Laboratory
Los Alamos, NM 87545

Abstract

sc2 is a new implementation of the Streams-C language and compiler. This document describes the design and implementation of the sc2 preprocessors and runtime and simulation libraries. The sc2 Reference Manual contains more information on the Streams-C programming language and the sc2 compiler structure. It should be read before reading this document. The sc2 preprocessors take as input the sc2 language program and translate the user directives into C++ code in the case of simulation or synthesis. For hardware processes, the preprocessor produces compiler pragmas as well. The Sim/RT libraries contain the code for initiating and terminating processes, communicating through streams and signals and the sc integer types.

1. Introduction

This document describes the implementation of the sc2 preprocessors and the simulation and runtime (Sim/RT) libraries. First, we will describe the goals of the sc2 Sim/RT implementation and changes from the previous implementation. We describe the organization of the sc2 compiler up to, but not including the back end stages, which translate the program's hardware processes into VHDL for the FPGAs. We describe the different paths the program can take depending on whether the user is compiling for simulation or synthesis. In the synthesis case, we describe the preprocessing required for the user's hardware processes versus software processes. We also describe the organization of the Sim/RT libraries – the way they implement processes, streams, signals and sc types. This description will also include how the intrinsic functions are implemented. A brief summary of the directory structure of the Sim/RT libraries will be presented as well.

2. Goals of the sc2 Sim/RT libraries

In the case of simulation, the goals of the Sim/RT libraries are to translate the user's directives into code necessary to implement software and hardware processes, streams and signals in software. The program should do the same thing it would if it were running with a combination of hardware and software processes, however it will not have the same performance, since it is only running in software. The outputs, however, should be the same. The simulation case should be used to check that process, stream and signal connections are correct. It should be able to provide information, via print statements, to the user about its execution paths and state.

In the case of synthesis, the goals of the Sim/RT libraries are to do two translations. For the software processes, it translates the user's directives into code necessary to implement them in software. For the hardware processes, it does a separate translation which includes the C code for the hardware processes. It also generates compiler pragmas for the synthesis compiler. The synthesis compiler processes all of this to generate the hardware codes (in VHDL) needed to run the processes on FPGAs.

New goals for the sc2 Sim/RT library implementation (beyond what was in the original version):

- Signals and their accompanying wait() and post() functions
- Predefined integer datatypes (signed and unsigned integers of various bit lengths)

- Arrays of processes, which implies arrays of streams and signals, since each process will contain the same number and type of streams and signals, just perhaps connected differently.
- Different way of specifying processes, streams and connections in the directives written by the user. sc2 eliminates STREAM directives and INPUT and OUTPUT directives in the PROCESS definition. sc2 adds stream and signal element datatypes to the IN/OUT STREAM and SIGNAL directives and adds a CONNECT definition to connect the streams and signals of the different processes.
- Processes can be initiated and terminated dynamically by main() or other software processes.
- sc2 allows the possibility of running processes on other machines in a network or cluster. The communication portion of the implementation then needs to be designed to allow you to implement the sc2 software processes in different ways (threads or regular UNIX processes on one host or over a network) and to be able to implement the hardware processes on FPGAs on other machines as well.

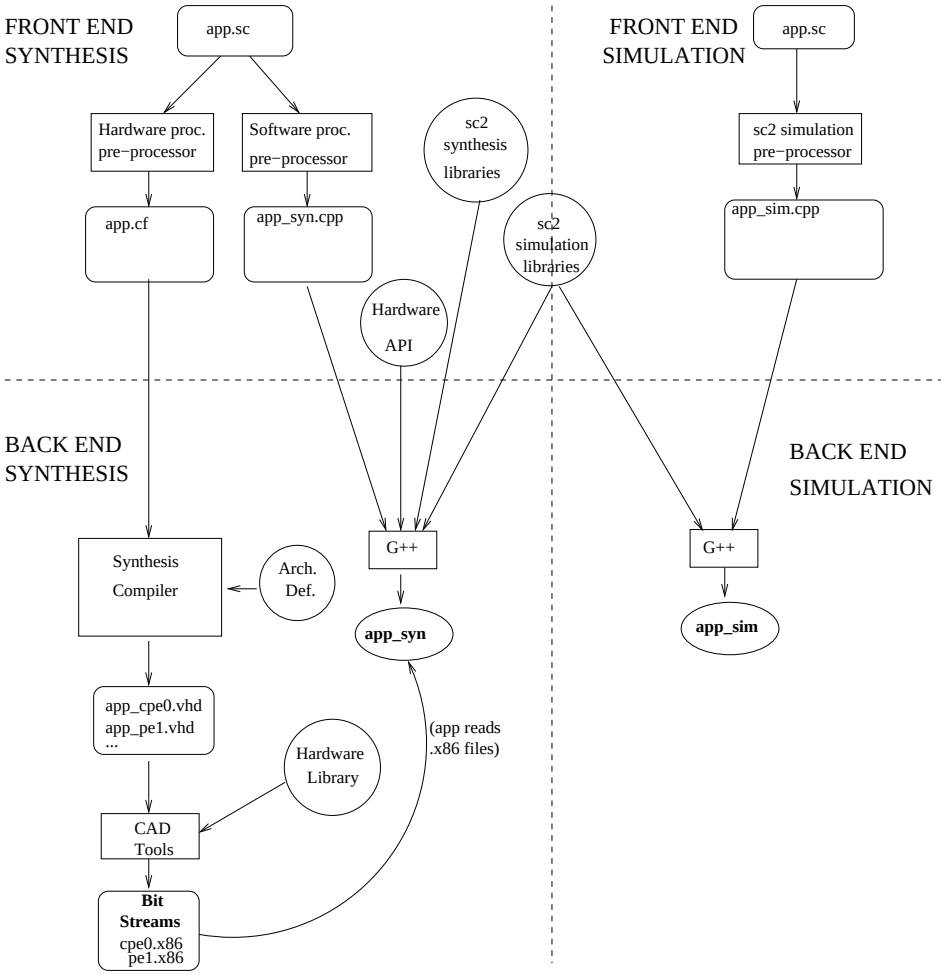

Figure 1. Organization of sc2

3. Organization of the sc2 Sim/RT Libraries

The figure above shows the organization of the sc2 compiler.

The Sim/RT portion described by this document includes all of the components above the dotted line. Below the dotted line lies the back end of the system, the true compilers. On the left is the synthesis path of execution. In this path, the user expects the program to be compiled to run software processes in software and to run hardware processes on the FPGAs. Within the synthesis side, the user's program is broken down into two parts, the software and hardware processes which get processed further either by linking to the Sim/RT libraries or by going through the synthesis compiler. The Sim/RT libraries contain the code that implements the processes, streams, signals and sc datatypes. The hardware API library contains the interface to the hardware used by the hardware processes in the program. This library is used when software processes need to communicate with hardware processes (in the implementation of streams and signals). The synthesis compiler outputs bit stream files, which represent the functions run on each of the FPGA board processors. These bit stream files are read in by the application executable. In contrast, on the right side of the figure is the simulation path of execution. In this path all processes are implemented in software.

4. The Preprocessors

4.1. Preprocessing for Software Side of Synthesis

For software processes, the app.sc file is preprocessed into a file called app_syn.cpp. Each process defined becomes a new derived class of sc_sw_process or sc_hw_process. Each process class defines a constructor and member function run() (which merely calls the PROCESS_FUN defined for that process). The PROCESS_FUN directive becomes a function entitled "[PROCESS name]_run" taking one argument, a pointer to an object that is the class type of the process. Each process function is allowed one parameter, which is implemented as a member of the newly defined process class. Streams and signals are also members of each process class declared. They all can be easily accessed in the run functions since the process object is passed as an argument. A global variable, sc2_processes, is an array of the newly created process objects. An sc_system object is created as a global variable, "sc2_system". The app_syn.cpp file will contain class definitions of software and hardware processes, but run functions only of the software processes. The main() will be augmented to contain the connect calls to connect streams and signals between software processes and between software and hardware processes. It also contains the assignment of registers to stream and signal ports if they are in hardware processes. Hardware to hardware connections will not be included. The main() is also augmented to start up and shut down the processors being used in synthesis as well as to start up and shut down the software system.

4.2. Preprocessing for Hardware Side of Synthesis

On the hardware side, the app.sc file is processed and translated into one file called app.cf before being processed by the synthesis compiler. It includes:

- the architecture definition of the hardware (in pragmas).
- include files, such as macros for the hardware compiler and other things.
- extern declarations of functions that are running in software and their pragmas
- definitions of functions running in hardware and their pragmas.
- pragmas for each process declared in ///PROCESS
- pragmas for each connection declared in ///CONNECT

Function headers have void return values and take processes and parameters as arguments. Pragma syntax is defined in the sc2 Reference Manual.

4.3. Preprocessing for Simulation

The app.sc file is processed and translated into app_sim.cpp. This is done in a similar manner to preprocessing for the software side of synthesis, however every process is a derived class of sc_sw_process. All connections are represented and all run functions are included. There are, however, no register assignments or startup/shutdown of processors.

4.4. Preprocessor Examples

Some examples are presented here to illustrate the C++ code generated for synthesis and simulation. The pragmas generated by the preprocessor for the hardware (the .cf file) are not presented here, as they are explained in the sc2 Reference Manual.

Consider the following sc2 directives for a hardware process.

```
/// PROCESS_FUN controller_run
/// PROCESS_FUN_BODY
/* code omitted */
/// PROCESS_FUN_END

/// PROCESS controller PROCESS_FUN controller_run TYPE HP ON pe0
```

For the software side of synthesis, it will become a sc_hw_process, running on a processor, pe0 (the run function is not included for hardware processes).

```
class controller_c : public sc_hw_process {
public:
  controller_c(const char* process_name,
      unsigned int new_id,
      sc_system* sys,
      processor* pe)
    :sc_hw_process(process_name, new_id, sys, pe)
    {
    }
  void* run(){};
};

controller_c* controller = new controller_c("controller", 1, sc2_system, pe0);
```

For the simulation, it will become an sc_sw_process running on the host machine with code included for the run function:

```
class controller_c : public sc_sw_process {
public:
  controller_c(const char* process_name,
      unsigned int new_id,
      sc_system* sys,
      host_machine* hm)
    :sc_sw_process(process_name, new_id, sys, hm)
    {
    }
  void* run();
};

controller_c* controller =
    new controller_c("controller", 1, sc2_system, sc_host);

void* controller_run (controller_c* proc)
{
/* code omitted */
}
```

Consider the following sc2 directives for a software process with a port (an output stream) and a parameter:

```
/// PROCESS_FUN host1_run
/// OUT_STREAM sc_int32 output_stream
/// PARAM int iterations
/// PROCESS_FUN_BODY
/* code not included here for simplicity */
/// PROCESS_FUN_END

/// PROCESS host1 PROCESS_FUN host1_run
```

For the software side of synthesis and for simulation, it becomes an sc_sw_process (with process id 3) with a stream and the parameter as members of its class.

```
class host1_c : public sc_sw_process {
public:
```

```
  sc_outstream<sc_int32>*  output_stream;
  int  iterations_param;
  host1_c(const char* process_name,
      unsigned int new_id,
      sc_system* sys,
      host_machine* hm)
    :sc_sw_process(process_name, new_id, sys, hm)
    {
      output_stream = new sc_outstream<sc_int32>(this);
    }
  void* params(int iterations);
  void* run();
};

host1_c* host1 = new host1_c("host1", 3, sc2_system, sc_host);

void* host1_run (host1_c* proc, int iterations)
{
/* code not included here for simplicity */
}

void* host1_c::params(int iterations)
{
  iterations_param = iterations;
}

void* host1_c::run()
{
  host1_run(this, iterations_param);
}
```

Consider the following example with an insignal on which it will wait.

```
/// PROCESS_FUN waiter_run
/// IN_SIGNAL sc_int32 insig
/// PROCESS_FUN_BODY
  sc_int32 data = sc_wait(insig);
/// PROCESS_FUN_END

/// PROCESS_FUN poster_run
/// OUT_SIGNAL sc_int32 outsig
/// PROCESS_FUN_BODY
  sc_post(outsig, 1);
/// PROCESS_FUN_END

/// PROCESS waiter PROCESS_FUN waiter_run TYPE SP ON sc_host
/// PROCESS poster PROCESS_FUN poster_run TYPE SP ON sc_host
```

The following code would be generated for the sythesis and simulation cases. Notice that the sc_wait() call must be transformed to indicate the signal datatype and the number of signals for which the process is waiting.

```
class waiter_c : public sc_sw_process {
public:
  sc_insignal<sc_int32>*  insig;
```

```
  waiter_c(const char* process_name,
      unsigned int new_id,
      sc_system* sys,
      host_machine* hm)
    :sc_sw_process(process_name, new_id, sys, hm)
    {
      insig = new sc_insignal<sc_int32>(this);
    }
  void* run();
};


class poster_c : public sc_sw_process {
public:
  sc_outsignal<sc_int32>*  outsig;
  poster_c(const char* process_name,
      unsigned int new_id,
      sc_system* sys,
      host_machine* hm)
    :sc_sw_process(process_name, new_id, sys, hm)
    {
      outsig = new sc_outsignal<sc_int32>(this);
    }
  void* run();
};

waiter_c* waiter = new waiter_c("waiter", 1, sc2_system, sc_host);
poster_c* poster = new poster_c("poster", 2, sc2_system, sc_host);

void* waiter_run (waiter_c* proc)
{
  sc_int32 data = sc_wait<sc_int32>(1, proc->insig);
} /* Process Fun End */

void* waiter_c::run()
{
  waiter_run(this);
}

void* poster_run (poster_c* proc)
{
  sc_post(outsig, 1);
} /* Process Fun End */

void* poster1_c::run()
{
  poster1_run(this);
}
```

For the same example, consider the sc2 program's CONNECT directive and main():

```
/// CONNECT waiter.insig poster1.outsig

void main(int argc, char *argv[])
```

```
{
  sc_initiate(waiter);
  sc_initiate(poster);
}
```

The following would be produced for the simulation and synthesis code. The preprocessor has added the connect() call, the system startup and shutdown code.

```
void main(int argc, char *argv[])
{
  connect(waiter->insig, 16, poster->outsig, 16);
  sc2_system->startup();

  sc_initiate(waiter);
  sc_initiate(poster);

  sc2_system->shutdown_wait();
}
```

Consider the array of processes indicated by the following directives:

```
/// PROCESS kmeans[2] PROCESS_FUN kmeans_run TYPE HP ON sc_host

/// CONNECT  kmeans[0].output_stream kmeans[1].input_stream
```

For simulation and synthesis, the following would be generated (assuming there is a prior kmeans_c class definition, which is omitted for simplicity).

```
sc_process_array<kmeans_c, 2>* kmeans_array =
  new sc_process_array<kmeans_c, 2>("kmeans", 2, sc2_system, sc_host);
kmeans_c** kmeans = kmeans_array->proc_array;

void main(int argc, char *argv[]) {
  /* code omitted */
  connect(kmeans[0]->output_stream, 16, kmeans[1]->input_stream, 16);
  /* code omitted */
}
```

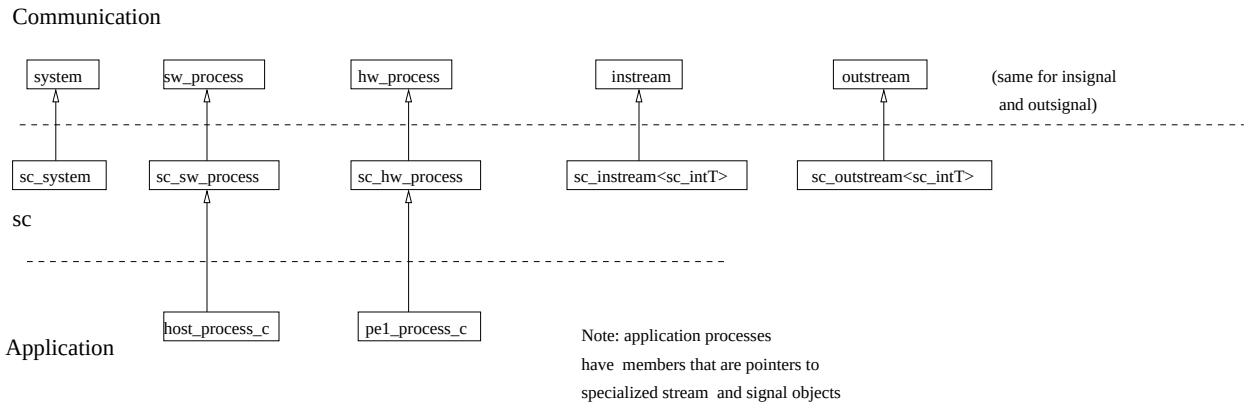

Figure 2. sc2 Sim/RT Class Hierarchy

5. The Sim/RT Libraries

The figure above shows the different layers in the Sim/RT library implementation.

At the lowest place in the hierarchy is the application layer, which may, for example, define two processes, “host_process” and “pe1_process”, connected by one stream. For this example, “host_process’ is a software process, “pe1_process” is a hardware process and the stream is an sc_int32 datatype stream. The preprocessor translates the user directives into classes “host_process_c” and “pe1_process_c”. As you can see from the class hierarchy diagram, the application level classes are derived from a set of “sc” classes. These are a generic set of classes used for implementing the sc2 user directives. However, they are really implemented for the underlying communication protocols by the communication layer at the top of the hierarchy diagram. The “sc” classes are all derived from classes in an upper layer, the communication layer. For this implementation, these communication layer classes and their methods are specifically implemented for threads. Other communication protocols at this level must provide the same named classes and methods.

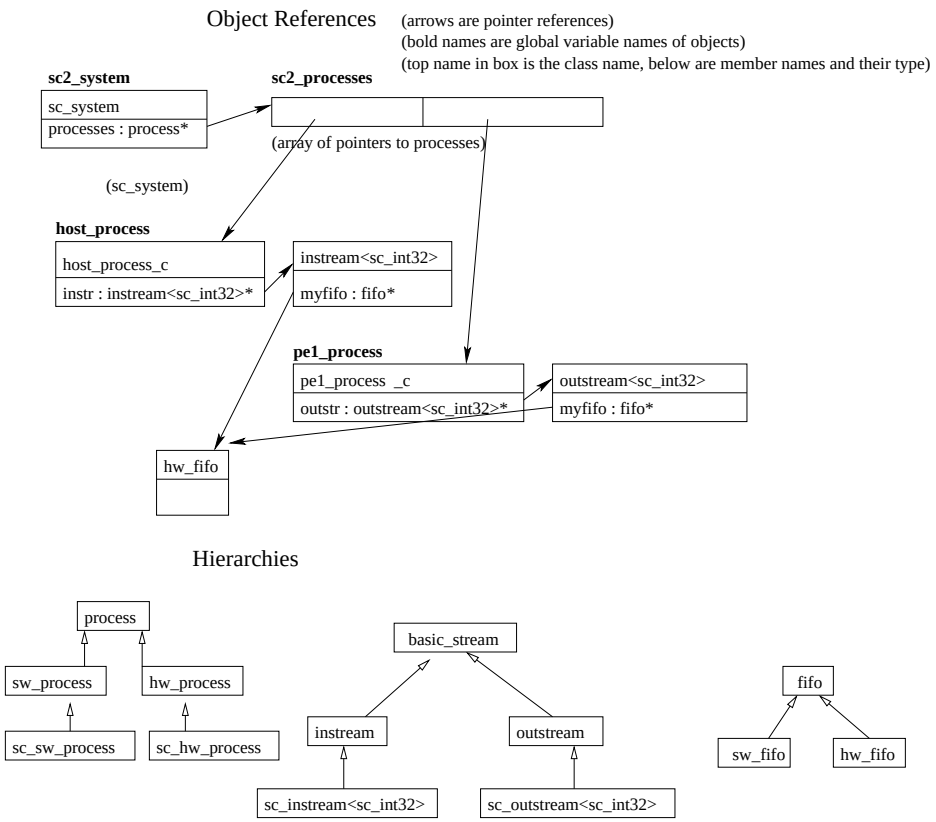

Figure 3. sc2 Sim/RT Object Connections

The figure above illustrates the way objects in the Sim/RT would be connected in the previous example where software process "host_process" is connected to hardware process "pe1_process" by streams. The words in the bold type indicate a global variable name for the object. There are global variables for the sc_system object, the process objects and the array that refers to a list of the processes in the system. The top word in a box is the object's class. For software to hardware connections in the pthreads implementation, the hardware fifo is implemented with a single object, pointed to by both the instream and outstream. Note that software to software connections use a software fifo.

The following sections will describe in more detail the implementation of each of the Sim/RT layers and classes in the layers.

5.1. System

Since the sc2 program can be implemented with different process models and because processes can be dynamically intiated and terminated, the sc2 implementation must provide a system class that represents the system as a whole.

Processes can potentially execute on different machines, or on the same machine, so there must be a way to get information about them. Also, since processes can be dynamically initiated and terminated, the system class must keep track of them. These were not issues in the previous version of Streams-C, because processes were only implemented with threads residing on one machine and were initiated all at once in main(), never reinitiated. At the end of main, it would do a join to make sure all the processes were finished before exiting.

The sc_system class represents the system object. It must somehow contain the process objects running in the program. It is constructed by giving an array of sc_process objects and array length.

The sc_system class is derived from base class system, which is expected to be provided by the communication layer. The system class is expected to provide the following operations: startup(), initiate_process(), terminate_process() and shut-

down_wait().

5.2. Processes

sc2 processes are represented by objects of classes that are derived from sc_sw_process and sc_hw_process. These classes are initialized with a name and unique ID of the process. The sc_[sw,hw]_process classes are derived from [sw,hw]_process classes in the communication layer, which implements initiate() and terminate().

5.3. Built-in sc Types

In a regular sc2 implementation (without an added fixed point library) 6 sc types are implemented: sc_int8, sc_int16, sc_int32, sc_uint8, sc_uint16, and sc_uint32. In order to implement the full range of types defined in the sc2 language, the user must use an added fixed point library and change the sc_types.h file to provide the bit insert and bit extract operations.

5.4. Streams

There are two types of streams at the comm layer. In the sc layer, a template allows the user to define a stream with an sc integer type as its parameter. For example, sc_instream<sc_int32> is a 32 bit signed integer type of instream. The stream classes are derived from non-templated classes instream and outstream. These base classes are implemented in the communcation layer. For example, sc_instream<sc_int32> would be derived from the instream class.

In main() streams are connected to each other if they are both in software processes, or one is in software and one is in hardware. Hardware to hardware connections are not represented. The fifo size (number of elements in the fifo) is passed as parameter when streams are connected.

5.5. Signals

Signals work in an analogous way to streams as far as being templated and their derived classes. They are also connected in a similar way.

5.6. Error Checking

Error checking is done at the sc level, so that the implementation provides consistent error checking no matter what the communication implementation is.

6. sc2 Intrinsic Functions

Most of the sc2 intrinsic functions are implemented as macros. This way, the correct function can be called depending upon what kind of process, stream, signal or sc integer type is passed as an argument or expected as a return type. This works, because in the process run functions, the process object is passed in as a parameter “proc”. The streams and signals are accessible from the process object. sc2_system is a global variable, so is accessible that way.

7. The Communication Layer

The various implementations of the communication layer must implement the following interface, which is expected by the “sc” layer of the Sim/RT. (Make this a table.)

- system_c, with methods register_process(), initiate_process() and terminate_process(), startup() and shutdown_wait().

sw,hw _process with methods initiate() and terminate()

- instream, with methods set_element_bytes(), read() and connect_ports()
- outstream, with methods set_element_bytes(), write() and connect_ports()

- insignal, with methods set_element_bytes(), connect_ports(), wait_for_me(), register_wait(), wait_for_many()
- outsignal, with methods set_element_bytes(), connect_ports() and post()
- host_machine

8. Pthreads Implementation of Communication Layer

8.1. System

For the pthreads implementation of the sc2 processes, the system object contains an array of the processes that can be initiated by the program. It also contains a pointer to a process that represents its own process. It must also contain a mutex and condition variable so that it can wait until all processes have exited before exiting the main() routine. It cannot exit main() early in shutdown_wait() or else that will terminate all the other running threads prematurely.

8.2. Processes

Each process class object has state which says whether it has been initiated. This is set to false in the constructor, then set to true after the pthread_create call is done. There is a mutex protecting this member. It also contains a pointer back to the sc_system object and a pthread handle for itself.

In the implementation of initiate() and terminate() the sc_system class must get ahold of the mutex in order to set the "initiated" flag for the object.

8.3. Streams

As mentioned in the Stream section, the sc streams are derived from classes instream and outstream. The software to software connectiosn are implemented in pthreads with only a fifo member. A stream of a software process will share a sw_fifo with another stream of a software process. If the connection is hardware to software or software to hardware, they will both be set to point to a hw_fifo object.

sw_fifo objects contain a pointer to a buffer and pointers to the head and tail of the fifo. There is a mutex for the fifo and two condition variables, representing the conditions "closed or not empty" and "not full". "closed or not empty" is required because the outstream may set itself to closed, in which case the instream should not be able to read from it, even if there is data there. So, the instream checks for the outstream setting the fifo to closed, or to not empty. In the case of closed, it returns with the data set to 0.

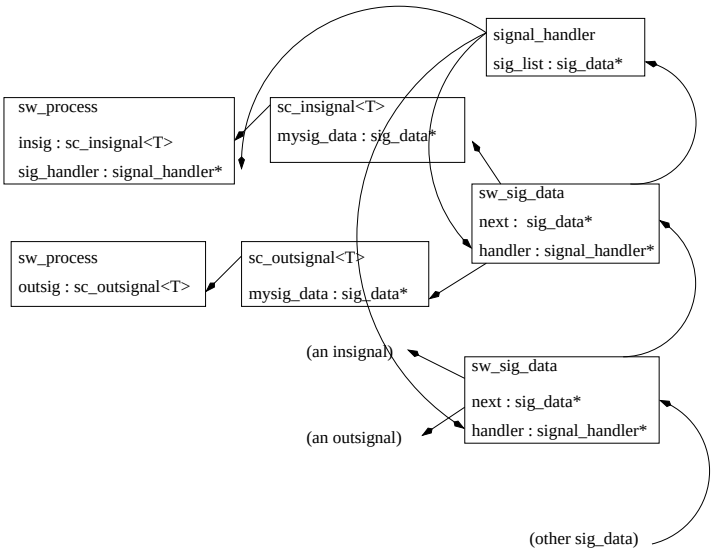

Figure 4. sc2 Signal Implementation

8.4. Signals

When connected, signals point to either sw_sig_data or hw_sig_data objects in a similar way to how streams work. The sig_data objects contain the information needed to store signal values, indicate that a value is stored, and indicate that it is a signal for which the process should wait. sig_data objects that represent data for insignals for a particular process are chained together in a linked list. A signal_handler object associated with the process has access to this list and uses it to manage waiting for signals.

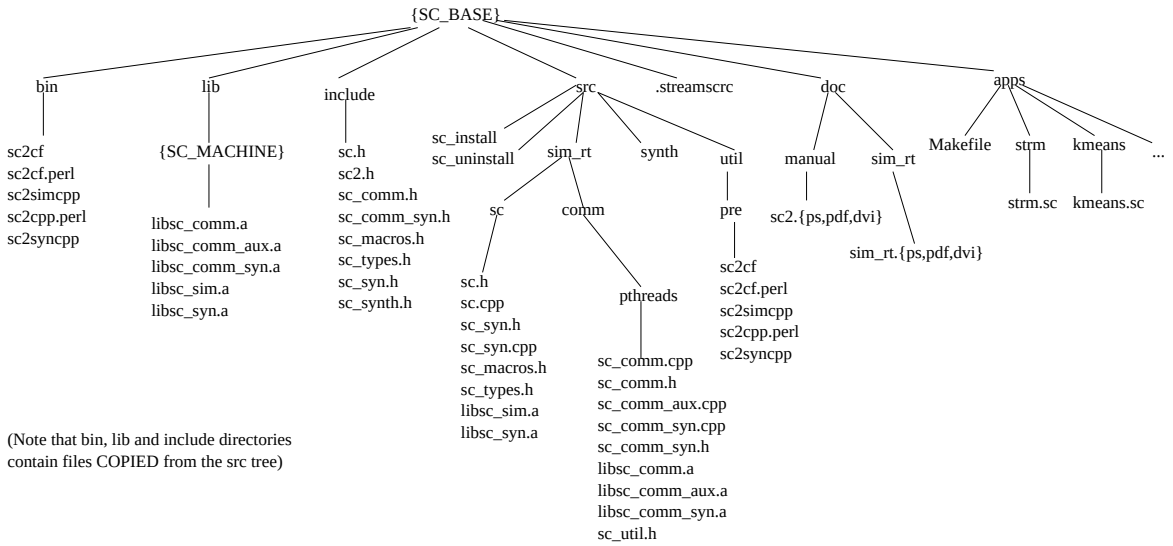

Figure 5. Organization of Sim/RT code

9. Directory Structure

The “src/util/pre” directory contains the preprocessors for simulation and synthesis. The “/src/sim_rt/sc” directory contains the “sc_” classes. The “comm” directory is meant to include potentially many implementations of the communication layer of the Sim/RT. Right now, only “pthreads” is being implemented. The “lib” directory contains the libraries needed by an application program to be processed by the Sim/RT. These libraries include the “sc” and “comm” libraries. The “include” directory includes all the include files for Sim/RT processing as well. The “bin” directory includes the perl scripts. The “apps” directory contains example applicatiosn for the user to read and run to help understand how to use sc2. The “doc” directory includes the sc2 reference manual (sc2.ps,dvi,pdf), and this document (sim_rt.ps,dvi,pdf).