6 - SystemC simulation

Chapter contents

Introduction

This chapter describes how to compile and simulate SystemC designs with ModelSim. ModelSim implements the SystemC language based on the Open SystemC Initiative (OSCI) SystemC 2.0.1 reference simulator. It is recommended that you obtain the OSCI functional specification, or the latest version of the SystemC Language Reference Manual as a reference manual. Visit *http://www.systemc.org* for details.

In addition to the functionality described in the OSCI specification, ModelSim for SystemC includes the following features:

- Single common Graphic Interface for SystemC and HDL languages.
- Extensive support for mixing SystemC, VHDL, and Verilog in the same design (SDF annotation for HDL only). For detailed information on mixing SystemC with HDL see *Chapter 7 - Mixed-language simulation*.

Supported platforms and compiler versions

SystemC runs on a subset of ModelSim supported platforms. The table below shows the currently supported platforms and compiler versions:

Platform	Supported compiler versions
HP-UX 11.0 or later	aCC 3.45 with associated patches
RedHat Linux 7.2 and 7.3 RedHat Linux Enterprise version 2.1	gcc 3.2.3
SunOS 5.6 or later	gcc 3.2
Windows NT and other NT-based platforms (win2K, XP, etc.)	Minimalist GNU for Windows (MinGW) gcc 3.2.3

Important: ModelSim SystemC has been tested with the gcc versions available from *ftp.model.com/pub/gcc*. Customized versions of gcc may cause problems. We strongly encourage you to download and use the gcc versions available on our FTP site (login as anonymous).

Building gcc with custom configuration options

We only test with our default options. **If you use advanced gcc configuration options, we cannot guarantee that ModelSim will work with those options.**

To use a custom gcc build, set the CppPath variable in the *modelsim.ini* file. This variable specifies the pathname to the compiler binary you intend to use.

When using a custom gcc, ModelSim requires that the custom gcc be built with several specific configuration options. These vary on a per-platform basis as shown in the following table:

Platform	Mandatory configuration options
Linux	none
Solaris	--with-gnu-ld --with-ld=/path/to/binutils-2.14/bin/ld --with-gnu-as --with-as=/path/to/binutils-2.14/bin/as
HP-UX	N/A
Win32 (MinGW)	--with-gnu-ld --with-gnu-as Do NOT build with the --enable-sjlj-exceptions option, as it can cause problems with catching exceptions thrown from SC_THREAD and SC_CTHREAD *ld.exe* and *as.exe* should be installed into the <*install_dir*>*/bin* before building gcc. *ld* and *as* are available in the binutils package. Modelsim uses binutils 2.13.90-20021006-2.

If you don't have a GNU binutils2.14 assembler and linker handy, you can use the as and ld programs distributed with ModelSim. They are located inside the built-in gcc in directory *<install_dir>/modeltech/gcc-3.2-<mtiplatform>/lib/gcc-lib/<gnuplatform>/3.2*.

By default ModelSim also uses the following options when configuring built-in gcc.

- --disable-nls
- --enable-languages=c,c++

These are not mandatory, but they do reduce the size of the gcc installation.

HP Limitations for SystemC

HP is supported for SystemC with the following limitations:

- variables are not supported
- aggregates are not supported
- objects must be explicitly named, using the same name as their object, in order to debug

SystemC simulation objects such as modules, primitive channels, and ports can be explicitly named by passing a name to the constructors of said objects. If an object is not constructed with an explicit name, then the OSCI reference simulator generates an internal name for it, using names such as "signal_0", "signal_1", etc..

Usage flow for SystemC-only designs

ModelSim allows users to simulate SystemC, either alone or in combination with other VHDL/Verilog modules. The following is an overview of the usage flow for strictly SystemC designs. More detailed instructions are presented in the sections that follow.

1. Create and map the working design library with the **vlib** and **vmap** statements, as appropriate to your needs.

2. Modify the SystemC source code, including the following highlights:
   - Replace **sc_main()** with an SC_MODULE, and potentially add a process to contain any testbench code
   - Replace **sc_start()** by using the **run** (CR-257) command in the GUI
   - Remove calls to **sc_initialize()**
   - Export the top level SystemC design unit(s) using the SC_MODULE_EXPORT macro

   See "Modifying SystemC source code" (UM-162) for a complete list of all modifications.

3. Analyze the SystemC source using **sccom** (CR-259). **sccom** invokes the native C++ compiler to create the C++ object files in the design library.

   See "Using sccom vs. raw C++ compiler" (UM-168) for information on when you are required to use **sccom** vs. another C++ compiler.

4. Perform a final link of the C++ source using **sccom -link** (UM-170). This process creates a shared object file in the current work library which will be loaded by **vsim** at runtime. **sccom -link** must be re-run before simulation if any new **sccom** compiles were performed.

5. Simulate the design using the standard **vsim** command.

6. Simulate the design using the **run** command, entered at the **vsim** command prompt.

7. Debug the design using ModelSim GUI features, including the Source and Wave windows.

Compiling SystemC files

To compile SystemC designs, you must

- create a design library
- modify the SystemC source code
- run the **sccom** (CR-259) SystemC compiler
- run the **sccom** (CR-259) SystemC linker (sccom -link)

Creating a design library

Before you can compile your design, you must create a library in which to store the compilation results. Use **vlib** (CR-361) to create a new library. For example:

```
vlib work
```

This creates a library named **work**. By default, compilation results are stored in the **work** library.

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX commands – always use the **vlib** command (CR-361).

See "Design libraries" (UM-55) for additional information on working with libraries.

Modifying SystemC source code

Several modifications must be applied to your original SystemC source code. To see example code containing the modifications listed below, see "Code modification examples" (UM-163).

Converting sc_main() *to a module*

In order for ModelSim to run the SystemC/C++ source code, the control function of **sc_main()** must be replaced by a constructor, SC_CTOR(), placed within a module at the top level of the design (see **mytop** in "Example 1" (UM-163)). In addition:

- any testbench code inside **sc_main()** should be moved to a process, normally an SC_THREAD process.
- all C++ variables in **sc_main()**, including SystemC primitive channels, ports, and modules, must be defined as members of sc_module. Therefore, initialization must take place in the SC_CTOR. For example, all **sc_clock()** and **sc_signal()** initializations must be moved into the constructor.

Replacing the sc_start() *function with the* run *command and options*

ModelSim uses the **run** command and its options in place of the **sc_start()** function. If **sc_main()** has multiple **sc_start()** calls mixed in with the testbench code, then use an **SC_THREAD()** with wait statements to emulate the same behavior. An example of this is shown below.

Removing calls to sc_initialize()

vsim calls **sc_initialize()** by default at the end of elaboration, so calls to **sc_initialize()** are unnecessary.

Exporting all top level SystemC modules

For SystemC designs, you must export all top level modules in your design to ModelSim. You do this with the **SC_MODULE_EXPORT(<sc_module_name>)** macro. SystemC templates are not supported as top level or boundary modules. See "Templatized SystemC modules" (UM-169). The **sc_module_name** is the name of the top level module to be simulated in ModelSim. You must specify this macro in a C++ source (*.cpp*) file. If the macro is contained in a header file instead of a C++ source file, an error may result.

For HP-UX: Explicitly naming signals, ports, and modules

Important: Verify that SystemC signals, ports, and modules are explicitly named to avoid port binding and debugging errors.

Code modification examples

Example 1

The following is a simple example of how to convert **sc_main** to a module and elaborate it with **vsim**.

Original OSCI code #1 (partial)	Modified code #1 (partial)

`int sc_main(int argc, char* argv[])`
`{`
`  sc_signal<bool> mysig;`
`  mymod mod("mod");`
`  mod.outp(mysig);`
 	
`  sc_start(100, SC_NS);`
`}`
	`SC_MODULE(mytop)`
	`{`
	`  sc_signal<bool> mysig;`
	`  mymod mod;`
	
	`  SC_CTOR(mytop)`
	`  : mysig("mysig"),`
	`  mod("mod")`
	`  {`
	`  mod.outp(mysig);`
	`  }`
	`};`
	
	`SC_MODULE_EXPORT(mytop);`

The run command equivalent to the `sc_start(100, SC_NS)` statement is:

```
run 100 ns
```

Example 2

This next example is slightly more complex, illustrating the use of **sc_main()** and signal assignments, and how you would get the same behavior using ModelSim.

Original OSCI code #2 (partial)	Modified ModelSim code #2 (partial)

`int sc_main(int, char**)`
`{`
`  sc_signal<bool> reset;`
`  counter_top top("top");`
`  sc_clock CLK("CLK", 10, SC_NS, 0.5, 0.0, SC_NS, false);`
 	
`  top.reset(reset);`
 	
`  reset.write(1);`
`  sc_start(5, SC_NS);`
`  reset.write(0);`
`  sc_start(100, SC_NS);`
`  reset.write(1);`
`  sc_start(5, SC_NS);`
`  reset.write(0);`
`  sc_start(100, SC_NS);`
`}`
	`SC_MODULE(new_top)`
	`{`
	`  sc_signal<bool> reset;`
	`  counter_top top;`
	`  sc_clock CLK;`
	
	`  void sc_main_body();`
	
	`  SC_CTOR(new_top)`
	`  : reset("reset"),`
	`  top("top")`
	`  CLK("CLK", 10, SC_NS, 0.5, 0.0, SC_NS, false)`
	`  {`
	`  top.reset(reset);`
	`  SC_THREAD(sc_main_body);`
	`  }`
	`};`
	
	`void`
	`new_top::sc_main_body()`
	`{`
	`  reset.write(1);`
	`  wait(5, SC_NS);`
	`  reset.write(0);`
	`  wait(100, SC_NS);`
	`  reset.write(1);`
	`  wait(5, SC_NS);`
	`  reset.write(0);`
	`  wait(100, SC_NS);`
	`}`
	
	`SC_MODULE_EXPORT(new_top);`

Example 3

One last example illustrates the correct way to modify a design using an SCV transaction database. ModelSim requires that the transaction database be created before calling the constructors on the design subelements. The example is as follows:

Original OSCI code # 3 (partial)	Modified ModelSim code #3 (partial)

`int sc_main(int argc, char* argv[])`
`{`
`  scv_startup();`
`  scv_tr_text_init();`
`  scv_tr_db db("my_db");`
`  scv_tr_db db::set_default_db(&db);`
 	
`  sc_clock clk ("clk",20,0.5,0,true);`
`  sc_signal<bool> rw;`
`  test t("t");`
 	
`  t.clk(clk);;`
`  t.rw(rw);`
 	
`  sc_start(100);`
`}`
	`SC_MODULE(top)`
	`{`
	`  sc_signal<bool>* rw;`
	`  test* t;`
	
	`  SC_CTOR(top)`
	`  {`
	`  scv_startup();`
	`  scv_tr_text_init()`
	`  scv_tr_db* db = new scv_tr_db("my_db");`
	`  scv_tr_db::set_default_db(db):;`
	
	`  clk = new sc_clock("clk",20,0.5,0,true);`
	`  rw = new sc_signal<bool> ("rw");`
	`  t = new test("t");`
	`  }`
	`};`
	
	`SC_MODULE_EXPORT(new_top);`

Take care to preserve the order of functions called in **sc_main()** of the original code.

Sub-elements cannot be placed in the initializer list, since the constructor body must be executed prior to their construction. Therefore, the sub-elements must be made pointer types, created with "new" in the SC_CTOR() module.

Invoking the SystemC compiler

ModelSim compiles one or more SystemC design units with a single invocation of **sccom** (CR-259), the SystemC compiler. The design units are compiled in the order that they appear on the command line. For SystemC designs, all design units must be compiled just as they would be for any C++ compilation. An example of an **sccom** command might be:

```
sccom -I ../myincludes mytop.cpp mydut.cpp
```

Compiling optimized and/or debug code

By default, **sccom** invokes the C++ compiler (g++ or aCC) without any optimizations. If desired, you can enter any g++/aCC optimization arguments at the **sccom** command line.

Also, source level debug of SystemC code is not available by default in ModelSim. To compile your SystemC code for debug, use the g++/aCC **-g** argument on the **sccom** command line.

Specifying an alternate g++ installation

We recommend using the version of g++ that is shipped with ModelSim on its various supported platforms. However, if you want to use your own installation, you can do so by setting the CppPath variable in the *modelsim.ini* file to the g++ executable location.

For example, if your g++ executable is installed in */u/abc/gcc-3.2/bin*, then you would set the variable as follows:

```
CppPath /u/abc/gcc-3.2/bin/g++
```

Maintaining portability between OSCI and ModelSim

If you intend to simulate on both ModelSim and the OSCI reference simulator, you can use the MTI_SYSTEMC macro to execute the ModelSim specific code in your design only when running ModelSim. The MTI_SYSTEMC macro is defined in ModelSim's *systemc.h* header file. When you #include this file in your SystemC code, you gain access to this macro. By including #ifdef/else statements in the code, you can then avoid having two copies of the design.

Using the original and modified code shown in the example shown on page 163, you might write the code as follows:

```
#ifdef MTI_SYSTEMC //If using the ModelSim simulator, sccom compiles this
SC_MODULE(mytop)
{
     sc_signal<bool> mysig;
     mymod mod;

     SC_CTOR(mytop)
         : mysig("mysig"),
           mod("mod")
     {
         mod.outp(mysig);
     }
};

SC_MODULE_EXPORT(top);

#else  //Otherwise, it compiles this
int sc_main(int argc, char* argv[])
{
     sc_signal<bool> mysig;
     mymod mod("mod");
     mod.outp(mysig);

     sc_start(100, SC_NS);
}
#endif
```

Restrictions on compiling with HP aCC

ModelSim uses the **aCC -AA** option by default when compiling C++ files on HP-UX. It does this so *cout* will function correctly in the *systemc.so* file. The **-AA** option tells **aCC** to use ANSI-compliant <iostream> rather than cfront-style <iostream.h>. Thus, all C++-based objects in a program must be compiled with **-AA**. This means you must use <iostream> and "using" clauses in your code. Also, you cannot use the **-AP** option, which is incompatible with **-AA**.

Switching platforms and compilation

Compiled SystemC libraries are platform dependent. If you move between platforms, you must remove all SystemC files from the working library and then recompile your SystemC source files. To remove SystemC files from the working directory, use the **vdel** (CR-332) command with the **-allsystemc** argument.

If you attempt to load a design that was compiled on a different platform, an error such as the following occurs:

```
# vsim work.test_ringbuf
# Loading work/systemc.so

# ** Error: (vsim-3197) Load of "work/systemc.so" failed:
work/systemc.so: ELF file data encoding not little-endian.

# ** Error: (vsim-3676) Could not load shared library
work/systemc.so for SystemC module 'test_ringbuf'.

# Error loading design
```

You can type **verror 3197** at the **vsim** command prompt and get details about what caused the error and how to fix it.

Using sccom vs. raw C++ compiler

When compiling complex C/C++ testbench environments, it is common to compile code with many separate runs of the compiler. Often users compile code into archives (.a files), and then link the archives at the last minute using the -L and -l link options.

When using ModelSim's SystemC, you may wish to compile a portion of your C design using raw g++ or aCC instead of **sccom**. Perhaps you have some legacy code or some non-SystemC utility code that you want to avoid compiling with **sccom**. You can do this, however, some caveats and rules apply.

Rules for sccom use

The rules governing when and how you must use **sccom** are as follows:

1. You must compile all code that references SystemC types or objects using **sccom** (CR-259).
2. When using **sccom**, you should not use the -I compiler option to point the compiler at any search directories containing OSCI or any other vendor supplied SystemC header files. **sccom** does this for you accurately and automatically.
3. If you do use the raw C++ compiler to compile C/C++ functionality into archives or shared objects, you must then link your design using the -L and -l options with the **sccom -link** command. These options effectively pull the non-SystemC C/C++ code into a simulation image that is used at runtime.

Failure to follow the above rules can result in link-time or elaboration-time errors due to mismatches between the OSCI or any other vendor supplied SystemC header files and the ModelSim SystemC header files.

Rules for using raw g++ to compile non-SystemC C/C++ code

If you use raw g++ to compile your non-systemC C/C++ code, the following rules apply:

1. The -fPIC option to g++ should be used during compilation with **sccom**.
2. For C++ code, you must use the built-in g++ delivered with ModelSim, or (if using a custom g++) use the one you built and specified with the CppPath .ini variable.

Otherwise binary incompatibilities may arise between code compiled by **sccom** and code compiled by raw g++.

Rules for using raw HP aCC to compile non-SystemC C/C++ code

If you use HP's aCC compiler to compile your non-systemC C/C++ code, the following rules apply:

1. For C++ code, you should use the +Z and -AA options during compilation
2. You must use HP aCC version 3.45 or higher.

Issues with C++ templates

Templatized SystemC modules

Templatized SystemC modules are not supported for use at:

- the top level of the design
- the boundary between SystemC and higher level HDL modules (i.e. the top level of the SystemC branch)

To convert a top level templatized SystemC module, you can either specialize the module to remove the template, or you can create a wrapper module that you can use as the top module.

For example, let's say you have a templatized SystemC module as shown below:

```
template <class T>
class top : public sc_module
{
    sc_signal<T> sig1;
    .
    .
    .
};
```

You can specialize the module by setting T = int, thereby removing the template, as follows:

```
class top : public sc_module
{
    sc_signal<int> sig 1;
    .
    .
    .
};
```

Or, alternatively, you could write a wrapper to be used over the template module:

```
class modelsim_top : public sc_module
{
    top<int> actual_top;
    .
    .
    .
};

SC_MODULE_EXPORT(modelsim_top);
```

Organizing templatized code

Suppose you have a class template, and it contains a certain number of member functions. All those member functions must be visible to the compiler when it compiles any instance of the class. For class templates, the C++ compiler generates code for each unique instance of the class template. Unless it can see the full implementation of the class template, it cannot generate code for it thus leaving the invisible parts as undefined. Since it is legal to have undefined symbols in a .so, **sccom -link** will not produce any errors or warnings. To make functions visible to the compiler, you must move them to the .h file.

Linking the compiled source

Once the design has been compiled, it must be linked using the **sccom** (CR-259) command with the **-link** argument.

sccom -link

The **sccom -link** command collects the object files created in the different design libraries, and uses them to build a shared library (.so) in the current work library or the library specified by the **-work** option. If you have changed your SystemC source code and recompiled it using **sccom**, then you must relink the design by running **sccom -link** before invoking **vsim**. Otherwise, your changes to the code are not recognized by the simulator. Remember that any dependent .a or .o files should be listed on the **sccom -link** command line before the .a or .o on which it depends. For more details on dependencies and other syntax issues, see **sccom** (CR-259).

Simulating SystemC designs

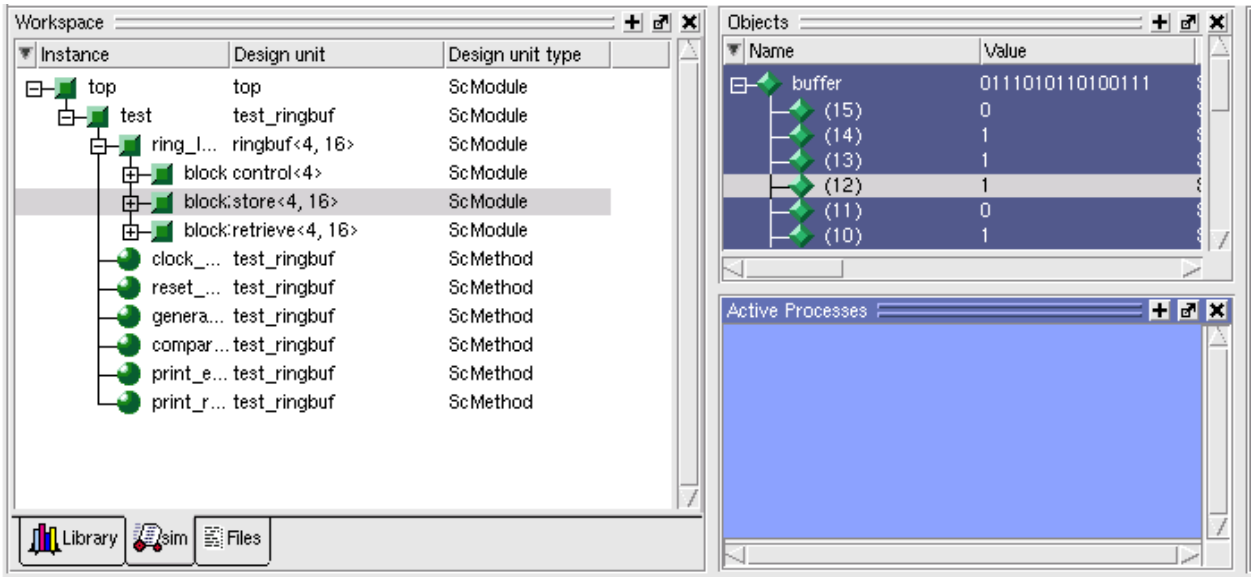
After compiling the SystemC source code, you can simulate your design with **vsim** (CR-378).

Loading the design

For SystemC, invoke **vsim** (CR-378) with the top-level module of the design. This example invokes **vsim** (CR-378) on a design named top:

```
vsim top
```

When the GUI comes up, you can expand the hierarchy of the design to view the SystemC modules. SystemC objects are denoted by green icons (see "Design object icons and their meaning" (GR-12) for more information).



To simulate from a command shell, without the GUI, invoke vsim with the -c option:

```
vsim -c <top_level_module>
```

Running simulation

Run the simulation using the **run** (CR-257) command or select one of the **Simulate** > **Run** options from the menu bar.

Simulator resolution limit

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. You can set the simulator resolution and user time unit from SystemC source code using the **sc_set_time_resolution()** and **sc_set_default_time_unit()** functions.

If the resolution is not set explicitly by **sc_set_time_resolution()**, the resolution limit defaults to the value specified by the **Resolution** (UM-507) variable in the *modelsim.ini* file. You can view the current resolution by invoking the **report** command (CR-249) with the **simulator state** option.

The rules vary if you have mixed-language designs. Please see "Simulator resolution limit" (UM-189) for details on mixed designs.

Choosing the resolution

Simulator resolution:

You should choose the coarsest simulator resolution limit possible that does not result in undesired rounding of your delays. However, the time precision should also not be set unnecessarily small, because in some cases performance will be degraded.

SystemC resolution:

The default resolution for all SystemC modules is 1ps. For all SystemC calls which don't explicitly specify units, the resolution is understood to be 1ps. The default is overridden by specifying units in the call.

Overriding the resolution

You can override ModelSim's default simulator resolution by specifying the **-t** option on the command line or by selecting a different Simulator Resolution in the **Simulate** dialog box. Available resolutions are: 1x, 10x, or 100x of fs, ps, ns, us, ms, or sec.

When deciding what to set the simulator's resolution to, you must keep in mind the relationship between the simulator's resolution and the SystemC time units specified in the source code. For example, with a time unit usage of:

```
sc_wait(10, SC_PS);
```

a simulator resolution of 10ps would be fine. No rounding off of the ones digits in the time units would occur. However, a specification of:

```
sc_wait(9, SC_PS);
```

would require you to set the resolution limit to 1ps in order to avoid inaccuracies caused by rounding.

Initialization and cleanup of SystemC state-based code

State-based code should not be used in Constructors and Destructors. Constructors and Destructors should be reserved for creating and destroying SystemC design objects, such as `sc_modules` or `sc_signals`. State-based code should also not be used in the elaboration phase callbacks **before_end_of_elaboration()** and **end_of_elaboration()**.

The following virtual functions should be used to initialize and clean up state-based code, such as logfiles or the VCD trace functionality of SystemC. They are virtual methods of the following classes: sc_port_base, sc_module, sc_channel, and sc_prim_channel. You can think of them as phase callback routines in the SystemC language:

- before_end_of_elaboration ()
  Called after all constructors are called, but before port binding.
- end_of_elaboration ()
  Called at the end of elaboration after port binding. This function is available in the SystemC 2.0.1 reference simulator.
- start_of_simulation ()
  Called before simulation starts. Simulation-specific initialization code can be placed in this function.
- end_of_simulation ()
  Called before ending the current simulation session.

The call sequence for these functions with respect to the SystemC object construction and destruction is as follows:

1. Constructors
2. before_end_of_elaboration ()
3. end_of_elaboration ()
4. start_of_simulation ()
5. end_of_simulation ()
6. Destructors

Usage of callbacks

The **start_of_simulation()** callback is used to initialize any state-based code. The corresponding cleanup code should be placed in the **end_of_simulation()** callback. These callbacks are only called during simulation by **vsim** and thus, are safe.

If you have a design in which some state-based code must be placed in the constructor, destructor, or the elaboration callbacks, you can use the **mti_IsVoptMode()** function to determine if the elaboration is being run by **vopt** (CR-376). You can use this function to prevent **vopt** from executing any state-based code.

Debugging the design

You can debug SystemC designs using all of ModelSim's debugging features, with the exception of the Dataflow window.

Viewable SystemC objects

Objects which may be viewed in SystemC for debugging purposes are as shown in the following table.

Channels	Ports	Variables	Aggregates
sc_signal<type> sc_signal_rv<width> sc_signal_resolved sc_clock (a hierarchical channel) sc_mutex sc_fifo	sc_in<type> sc_out<type> sc_inout<type> sc_in_rv<width> sc_out_rv<width> sc_inout_rv<width> sc_in_resolved sc_out_resolved sc_inout_resolved sc_in_clk sc_out_clk sc_inout_clk sc_fifo_in sc_fifo_out	Module member variables of all C++ and SystemC built-in types (listed in the Types list below) are supported.	Aggregates of SystemC signals or ports. Only three types of aggregates are supported for debug: struct class array

Types (<type>) of the objects which may be viewed for debugging are the following:

Types
bool, sc_bit
sc_logic
sc_bv<width>
sc_lv<width>
sc_int<width>
sc_uint<width>
sc_fix
sc_fix_fast
sc_fixed<W,I,Q,O,N>
sc_fixed_fast<W,I,Q,O,N>
sc_ufix
sc_ufix_fast
sc_ufixed
sc_ufixed_fast
sc_signed
sc_unsigned
char, unsigned char
int, unsigned int
short, unsigned short
long, unsigned long
sc_bigint<width>
sc_biguint<width>
sc_ufixed<W,I,Q,O,N>
short, unsigned short
long long, unsigned long long
float
double
enum
pointer
class
struct
union
bit_fields

Source-level debug

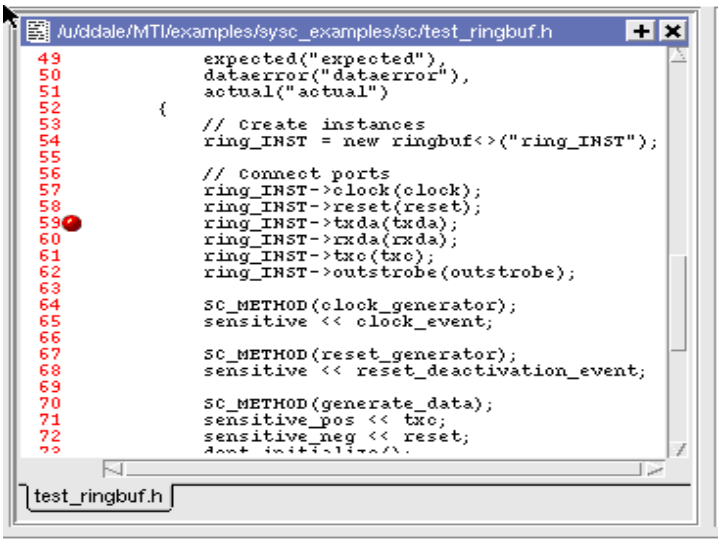
In order to debug your SystemC source code, you must compile the design for debug using the **-g** C++ compiler option. You can add this option directly to the **sccom** (CR-259) command line on a per run basis, with a command such as:

```
sccom mytop -g
```

Or, if you plan to use it every time you run the compiler, you can specify it in the *modelsim.ini* file with the **SccomCppOptions** variable. See "[sccom] SystemC compiler control variables" (UM-501) for more information.

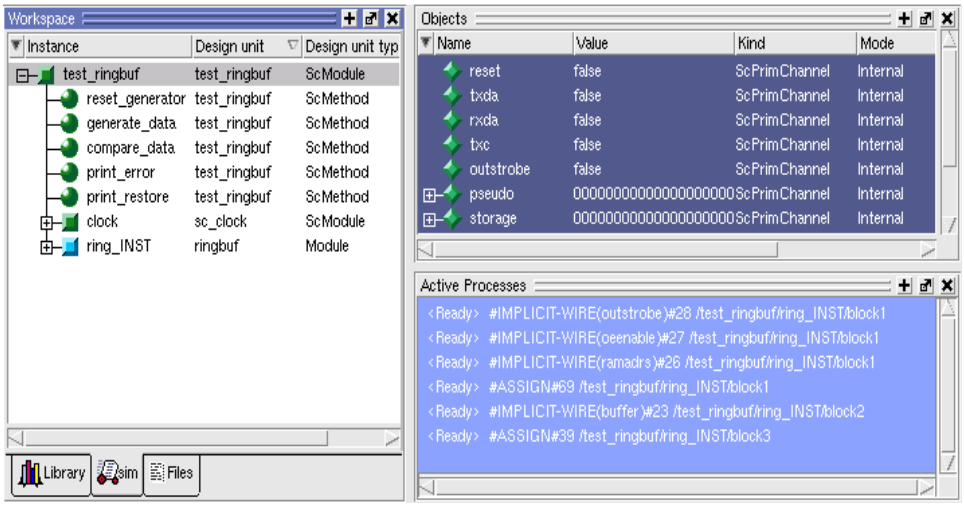
The source code debugger, C Debug (UM-399), is automatically invoked when the design is compiled for debug in this way.

You can set breakpoints in a Source window, and single-step through your SystemC/C++ source code. .



The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Try to avoid setting breakpoints in constructors of SystemC objects; it may crash the debugger.

You can view and expand SystemC objects in the Objects pane and processes in the Active Processes pane.



SystemC object and type display in ModelSim

This section contains information on how ModelSim displays certain objects and types, as they may differ from other simulators.

Support for aggregates

ModelSim supports aggregates of SystemC signals or ports. Three types of aggregates are supported: structures, classes, and arrays. Unions are not supported for debug. An aggregate of signals or ports will be shown as a signal of aggregate type. For example, an aggregate such as:

```
sc_signal <sc_logic> a[3];
```

is equivalent to:

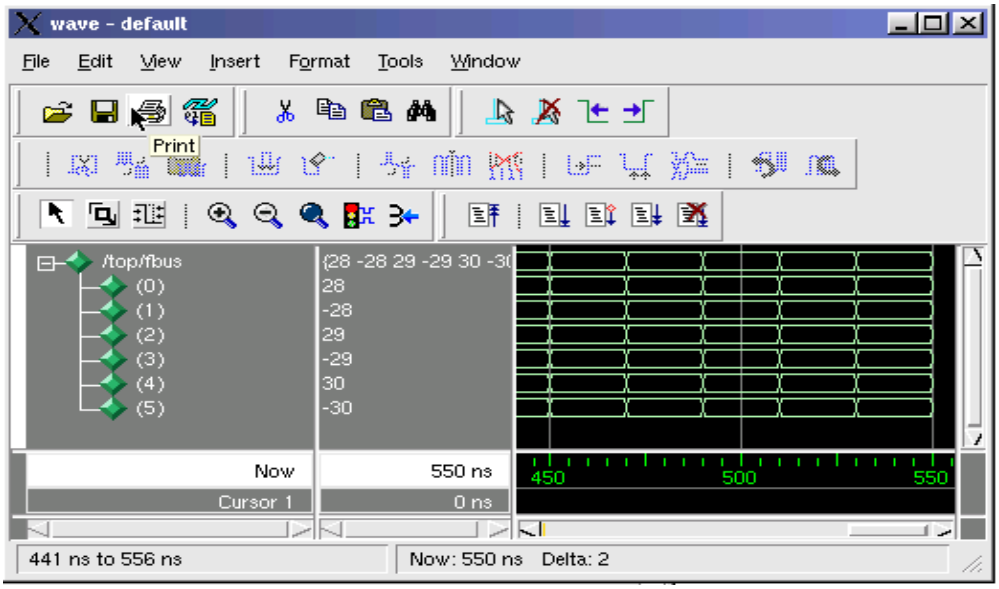
```
sc_signal <sc_lv<3>> a;
```

for debug purposes. ModelSim shows one signal - object "a" - in both cases.

The following aggregate

```
sc_signal <float> fbus [6];
```

when viewed in the Wave window, would appear as follows:



Viewing FIFOs

In ModelSim, the values contained in an sc_fifo appear in a definite order. The top-most or left-most value is always the next to be read from the FIFO. Elements of the FIFO that are not in use are not displayed.

Example of a signal where the FIFO has five elements:

```
# examine f_char
# {}
VSIM 4> # run 10
VSIM 6> # examine f_char
# A
VSIM 8> # run 10
VSIM 10> # examine f_char
# {A B}
VSIM 12> # run 10
VSIM 14> # examine f_char
# {A B C}
VSIM 16> # run 10
VSIM 18> # examine f_char
# {A B C D}
VSIM 20> # run 10
VSIM 22> # examine f_char
# {A B C D E}
VSIM 24> # run 10
VSIM 26> # examine f_char
# {B C D E}
VSIM 28> # run 10
VSIM 30> # examine f_char
# {C D E}
VSIM 32> # run 10
VSIM 34> # examine f_char
# {D E}
```

Differences between ModelSim and the OSCI simulator

ModelSim is based upon the 2.0.1 reference simulator provided by OSCI. However, there are some minor but key differences to understand:

- **vsim** calls **sc_initialize()** by default at the end of elaboration. The user has to explicitly call **sc_initialize()** in the reference simulator. You should remove calls to **sc_initialize()** from your code.
- The default time resolution of the reference simulator is 1ps. For **vsim** it is 1ns. The user can set the time resolution by using the **vsim** command with the **-t** option or by modifying the value of the Resolution (UM-507) variable in the *modelsim.ini* file.
- All SystemC processes without a **dont_initialize()** modifier are executed once at the end of elaboration. This can cause print messages to appear from user models before the first VSIM> prompt occurs. This behavior is normal and necessary in order to achieve compliance with both the SystemC and HDL LRMs.
- The **run** command in ModelSim is equivalent to **sc_start()**. In the reference simulator, **sc_start()** runs the simulation for the duration of time specified by its argument. In ModelSim the **run** command (CR-257) runs the simulation for the amount of time specified by its argument.
- The **sc_cycle()**, **sc_start()**, **sc_main()** & **sc_set_time_resolution()** functions are not supported in ModelSim.

Fixed point types

Contrary to OSCI, ModelSim compiles the SystemC kernel with support for fixed point types. If you want to compile your own SystemC code to enable that support, you'll need to define the compile time macro SC_INCLUDE_FX. You can do this in one of two ways:

- enter the g++/aCC argument -DSC_INCLUDE_FX on the **sccom** (CR-259) command line, such as:

```
sccom -DSC_INCLUDE_FX top.cpp
```

- add a define statement to the C++ source code before the inclusion of the *systemc.h*, as shown below:

```
#define SC_INCLUDE_FX
#include "systemc.h"
```

OSCI 2.1 features supported

ModelSim is fully compliant with the OSCI version 2.0.1. In addition, the following 2.1 features are supported:

Hierarchical reference SystemC functions

The following two member functions of sc_signal, used to control and observe hierarchical signals in a design, are supported:

- control_foreign_signal()
- observe_foreign_signal()

For more information regarding the use of these functions, see "Hierarchical references in mixed HDL/SystemC designs" (UM-190).

Phase callback

The following functions are supported for phase callbacks:

- before_end_of_elaboration()
- start_of_simulation()
- end_of_simulation()

For more information regarding the use of these functions, see "Initialization and cleanup of SystemC state-based code" (UM-173).

Accessing command-line arguments

The following global functions allow you to gain access to command-line arguments:

- sc_argc()
  Returns the number of arguments specified on the **vsim** (CR-378) command line with the **-sc_arg** argument. This function can be invoked from anywhere within SystemC code.
- sc_argv()
  Returns the arguments specified on the **vsim** (CR-378) command line with the **-sc_arg** argument. This function can be invoked from anywhere within SystemC code.

Example:

When **vsim** is invoked with the following command line:

```
vsim -sc_arg "-a" -c -sc_arg "-b -c" -t ns -sc_arg -d
```

sc_argc() and sc_argv() will behave as follows:

```
int argc;
const char * const * argv;

argc = sc_argc();
argv = sc_argv();
```

The number of arguments (argc) is now 4.

argv[0] is "vsim"
argv[1] is "-a"
argv[2] is "-b -c"
argv[3] is "-d"

Troubleshooting SystemC errors

In the process of modifying your SystemC design to run on ModelSim, you may encounter several common errors. This section highlights some actions you can take to correct such errors.

Errors during loading

When simulating your SystemC design, you might get a "failed to load sc lib" message because of an undefined symbol, looking something like this:

```
# Loading /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so

# ** Error: (vsim-3197) Load of "/home/cmg/newport2_systemc/chip/vhdl/work/
systemc.so" failed: ld.so.1:

/home/icds_nut/modelsim/5.8a/sunos5/vsimk: fatal: relocation error: file

/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so: symbol
_Z28host_respond_to_vhdl_requestPm:

referenced symbol not found.

# ** Error: (vsim-3676) Could not load shared library /home/cmg/
newport2_systemc/chip/vhdl/work/systemc.so for SystemC module 'host_xtor'.
```

Source of undefined symbol message

The causes for such an error could be:

- missing definition
- bad link order specified in sccom -link
- multiply-defined symbols

Missing definition

If the undefined symbol is a C function in your code or a library you are linking with, be sure that you declared it as an extern "C" function:

```
extern "C" void myFunc();
```

This should appear in any header files include in your C++ sources compiled by **sccom**. It tells the compiler to expect a regular C function; otherwise the compiler decorates the name for C++ and then the symbol can't be found.

Also, be sure that you actually linked with an object file that fully defines the symbol. You can use the "nm" utility on Unix platforms to test your SystemC object files and any libraries you link with your SystemC sources. For example, assume you ran the following commands:

```
sccom test.cpp
sccom -link libSupport.a
```

If there is an unresolved symbol and it is not defined in your sources, it should be correctly defined in any linked libraries:

```
nm libSupport.a | grep "mySymbol"
```

Misplaced "-link" option

The order in which you place the **-link** option within the **sccom -link** command is critical. There is a big difference between the following two commands:

```
sccom -link liblocal.a
```

and

```
sccom liblocal.a -link
```

The first command ensures that your SystemC object files are seen by the linker before the library "liblocal.a" and the second command ensures that "liblocal.a" is seen first. Some linkers can look for undefined symbols in libraries that follow the undefined reference while others can look both ways. For more information on command syntax and dependencies, see **sccom** (CR-259).

Multiple symbol definition errors

The most common type of error found during **sccom -link** operation is the multiple symbol definition error. This typically arises when the same global symbol is present in more than one *.o* file. The error message looks something like this:

```
work/sc/gensrc/test_ringbuf.o: In function
`test_ringbuf::clock_generator(void)':

work/sc/gensrc/test_ringbuf.o(.text+0x4): multiple definition of
`test_ringbuf::clock_generator(void)'

work/sc/test_ringbuf.o(.text+0x4): first defined here
```

A common cause of multiple symbol definitions involves incorrect definition of symbols in header files. If you have an out-of-line function (one that isn't preceded by the "inline" keyword) or a variable defined (i.e. not just referenced or prototyped, but truly defined) in a *.h* file, you can't include that *.h* file in more than one *.cpp* file.

Text in *.h* files is included into *.cpp* files by the C++ preprocessor. By the time the compiler sees the text, it's just as if you had typed the entire text from the *.h* file into the *.cpp* file. So a *.h* file included into two *.cpp* files results in lots of duplicate text being processed by the C++ compiler when it starts up. Include guards are a common technique to avoid duplicate text problems. See "Errors during loading" (UM-182) for more information on include guards.

If an *.h* file has an out-of-line function defined, and that *.h* file is included into two *.c* files, then the out-of-line function symbol will be defined in the two corresponding. *o* files. This leads to a multiple symbol definition error during **sccom -link**.

To solve this problem, add the "inline" keyword to give the function "internal linkage". This makes the function internal to the *.o* file, and prevents the function's symbol from colliding with a symbol in another *.o* file.

For free functions or variables, you could modify the function definition by adding the "static" keyword instead of "inline", although "inline" is better for efficiency.

Sometimes compilers do not honor the "inline" keyword. In such cases, you should move your function(s) from a header file into an out-of-line implementation in a *.cpp* file.