Here's Exactly What You Can Do with the New SystemC Standard!

John Aynsley, Doulos, Ringwood, UK.

email: john.aynsley@doulos.com

Abstract

This paper, written by an author of the SystemC Language Reference Manual, presents a unique perspective on the changes made to the SystemC class library during the standardization process, on the new features added to SystemC, and on the important clarifications of intent that resulted from this process. The SystemC LRM draws a clear line between the features that the class library is intended to provide to the user, and the features that are merely artifacts of the implementation. This information is important for both SystemC users and tool developers.

Introduction

SystemC has been in use for over five years. During that time is has undergone a number of changes, culminating in the publication of the IEEE Std 1666-2005 SystemC Language Reference Manual (LRM), which was approved as an IEEE standard in December 2005. SystemC version 1 concentrated on hardware and fixed-point arithmetic modeling, version 2 on the abstraction of communication. These SystemC releases took the form of an open source C++ class library with accompanying informal documentation, so any questions of definition could only be answered by reference to the C++ source code. Standardization required a shift of authority from the source code to the LRM. The OSCI open-source simulator now has the status of a proof-of-concept implementation of SystemC as defined by the LRM. Nonetheless the OSCI simulator remains the basis for every current full SystemC implementation, and OSCI is committed to keeping the proof-of-concept simulator compliant with the IEEE standard as it evolves.

The standardization process has fixed a number of inconsistencies within the existing class library, has identified several features as being undesirable and therefore deprecated, and has raised requests for new features. By and large, these changes have been accommodated while retaining backward compatibility with existing SystemC code, but there are a few exceptions where existing code will require modification. Deprecated features will remain in the OSCI open source simulator indefinitely, so users need not rush to modify legacy code.

The process of shifting authority from the source code to the LRM has forced the working group to clarify a number of issues concerning which parts of the class library are to be regarded as user-level features, and which parts are mere artifacts of the implementation. In some cases, this has resulted in the promotion of certain artifacts to become user-level features. In a few cases this has involved changes and enhancements that destroy strict backward compatibility with earlier versions.

This paper will not enumerate every change, but will concentrate on the more interesting user-level features and will provide use cases and examples of interest to end users and tool developers. A more comprehensive listing is provided as an Annex to the LRM.

Starting from the top

The SystemC class library provides a single header file, **systemc.h**, for inclusion in applications. This header file adds all of the names from the SystemC class library to the declarative region in which it is included, together with many other names from the C and C++ standard libraries. This is convenient, but is namespace pollution on a grand scale, contradicting good software design practice and the modern C++ practice of using explicit namespaces.

The new standard addresses this issue by introducing a new header file **systemc**. Although the header file **systemc.h** will continue to be supported, the new header file **systemc** is now preferred. This new header file only introduces two names, **sc_core** and **sc_dt**, each of which is the name of a C++ namespace. A SystemC application can now be selective about which names it chooses to use, thus avoiding collisions between names from different header files. For example:

```
#include "systemc"
using sc_core::sc_module;
using sc_core::sc_signal;
using std::cout;
using std::endl;

struct Mod: sc_module
{
  sc_signal<sc_dt::sc_logic> sig;
  ...
  cout << endl;
  ...
};
```

Standards within standards

At the point in time when SystemC was developed, the standard C++ library (also known as STL, the Standard Template Library) had the reputation of being non-portable between popular C++ compilers, and hence the SystemC originators took the deliberate decision to avoid STL and implement their own container classes in SystemC. These classes included **sc_string** and **sc_pvector**. But times move on, the standard C++ libraries now enjoy full and consistent support across the main C++ compilers, and this fact is now recognized in SystemC.

The new SystemC standard replaces **sc_string** with **std::string**, **sc_pvector** with **std::vector**, and **sc_exception** with **std::exception**. The old non-standard classes are now deprecated. By default, the name **sc_string** is undefined, but the old **sc_string** class is still part of the source code under the name of **sc_string_old**. All new SystemC applications should use **std::string** exclusively. These changes will render obsolete code that uses the old classes. The OSCI simulator contains the following definitions to enable a degree of backward compatibility:

```
typedef std::exception sc_exception;

namespace sc_dt {
        class sc_string_old;
}

#ifdef SC_USE_SC_STRING_OLD
        typedef sc_dt::sc_string_old sc_string;
#endif
#ifdef SC_USE_STD_STRING
        typedef ::std::string sc_string;
#endif
```

An application wishing to use the deprecated type name **sc_string** can define one of the two macros and thus pick up the old or the new definition.

Allow me to elaborate

SystemC shares the concept of elaboration with VHDL and Verilog. A basic concept in SystemC is the static elaboration of the module hierarchy; all module instantiation and port binding must be completed before the end of elaboration, while the execution of processes and the notification of events is strictly confined to simulation. Furthermore, elaboration strictly precedes simulation. SystemC version 2.1 added new callbacks and dynamic processes, which muddied the waters a little. As a result, the definition of elaboration required some clarification as SystemC was standardized.

One clarification made in the new standard is that SystemC is a class library, not a language. SystemC has neither a concrete nor an abstract syntax of its own. The semantics of elaboration depend on the order in which objects are constructed, not on the existence of certain C++ variables or particular class data members. This has the practical consequence that a module or channel instance does not need a corresponding name in the C++ syntax, which makes the writing of a general SystemC parser problematic.

Modules, ports, exports and primitive channels can only be instantiated during elaboration, and ports can only be bound during elaboration. Another clarification made in the new standard is that the precise mechanism for port binding is implementation-defined, and port binding is not guaranteed to be complete until the end of elaboration. This clarification implies that any code that assumes a port is bound before the end of elaboration could be non-portable or even cause a fatal error. The **end_of_elaboration()** callbacks are made just after the end of elaboration, and hence can rely on instantiation and port binding having been completed, but cannot instantiate further objects in the module hierarchy or bind ports. On the other hand, the **before_end_of_elaboration()** callbacks are made before the end of elaboration (as you might infer from the function name), so can instantiate modules or bind ports, but cannot rely on port binding having been completed.

A use case that arises in transaction-level modeling is the binding of monitor ports on modules instantiated throughout the hierarchy to a single monitor channel instantiated at the top level. Because elaboration is performed depth-first, bottom-up, the monitor ports cannot be bound until the construction of the monitor channel at the top-level has been completed, as shown in the following example:

```
Monitor_channel *global_mon;

struct Component: sc_module {
  sc_port<mon_if> monitor_port;
  ...
  void before_end_of_elaboration() {
    monitor_port.bind(*global_mon);
  }
};

struct Top: sc_module
{
  SC_CTOR(Top) {
    ...
    global_mon = new Monitor_channel("mon");
  }
  ...
};
```

In general, the writer must assume that the monitor channel has not yet been constructed at the point when module Component is instantiated, so the port cannot be bound at that point. In order to avoid unwanted

dependencies between modules, port binding is performed in the **before_end_of_elaboration()** callback of module Component itself rather than at the top level after the instantiation of the monitor.

Finding the big event

Because a SystemC implementation may defer the completion of port binding until a later stage during elaboration, SystemC requires the use of an *event finder* when a port is added to the static sensitivity of a process during elaboration. An event finder is a member function of a port that serves to defer the interface method call that retrieves an event until just before the end of elaboration, after port binding has been completed. Perhaps the most well-known examples are the event finders **pos()** and **neg()** of the class **sc_in<bool>**. The point is that the port will not have been bound to a channel at the point when the port is added to the static sensitivity of a process; indeed, the channel may not yet have been instantiated, so clearly the event within that channel cannot be retrieved at this stage. The order of proceedings is:

1. Construction of the module hierarchy using a depth-first descent through the module constructors.

2. The **before_end_of_elaboration()** callbacks

3. Port binding is completed

4. Event finders retrieve events from channels and complete the building of static sensitivity

5. The **end_of_elaboration()** callbacks

Event finders have their limitations. For example, it is not possible to pass an argument to an event finder function, perhaps to select one from a number of events. When a multiport is bound to more than one channel instance, event finders in versions of the OSCI simulator up to version 2.1 have the annoying feature of only retrieving an event from the first such channel. The new standard makes it clear that static sensitivity should include the events from every channel instance to which a multiport is bound.

An alternative is to bypass the event finder mechanism by only adding events to the static sensitivity of processes after the end of elaboration, in the **end_of_elaboration()** callbacks. At this stage, event finders are not necessary (indeed, they cannot be called) because port binding has been completed. Instead, interface method calls can be made to retrieve events. The question then arises as to what use can be made of these events after the end of elaboration, when no further module instantiation is permitted.

The process macros SC_METHOD, SC_THREAD and SC_CTHREAD can be invoked during elaboration. These process macros have always been permitted in **end_of_elaboration()**, and the new standard clarifies that this is a deliberate feature rather than an accident of the implementation. Here is an example of bypassing the event finder mechanism:

```
struct M: sc_module {
  sc_port<sc_signal_in_if<int>,0> p;     // Multiport

  void end_of_elaboration() {               // Callback
    SC_METHOD(action);
    for (int i = 0; i < p.size(); i++)
      sensitive << p[i]->value_changed_event();
  }
  ...
};
```

From SystemC version 2.1 onward it is also possible to spawn processes by calling **sc_spawn()** both during elaboration and during simulation. Hence processes created either by the process macros (also known as *unspawned* processes) or spawned processes can be made statically sensitive to events in the **end_of_elaboration()** callbacks, avoiding the need to call event finders. In the new standard, processes created before the end of elaboration are known as *static* processes, whereas those created after the end of elaboration are known as *dynamic* processes. Technically, the process macros and **sc_spawn** can each create either static or dynamic processes. Static processes must use event finders when being made sensitive to ports, but dynamic processes cannot use event finders when being made sensitive.

No port of call

It was a rule in all versions of SystemC up to version 2.1 that every port and every export must be bound at least once. Unbound ports were not permitted. This was inconvenient at best, and made it hard to create optional ports. Consider the following example, which attempts to workaround this limitation:

```
SC_MODULE(M){
  sc_port<i_f> opt_port;
  ...
  void before_end_of_elaboration() {
    if ( opt_port.size() == 0 ) {
      dummy = new dummy_channel();
      opt_port.bind(*dummy);
      ...
```

The callback **before_end_of_elaboration()** checks whether the optional port has been bound. If not, it binds the port to a local dummy channel. Although this code fragment works in the OSCI simulator, its behavior is strictly indeterminate because the new standard makes clear that port binding is not actually guaranteed complete before the end of elaboration.

The new SystemC standard provides a better solution by introducing the port binding policy. A third argument to the **sc_port** class template specifies

whether the port may remain unbound at the end of elaboration and during simulation.

```
sc_port<interface, max, policy>
```

The default policy of SC_ONE_OR_MORE_BOUND means that the port must be bound to at least one channel, the maximum number being given by the *max* argument, and with a value of zero meaning no upper limit. This was the old behavior. Two new policies have been added. The policy SC_ZERO_OR_MORE_BOUND permits the port to remain unbound. SC_ALL_BOUND means that the port must be bound exactly *max* times.

It is now possible to create an optional port as follows:

```
sc_port<i_f, 1, SC_ZERO_OR_MORE_BOUND> opt_port;
```

This port can be bound to zero or one channel, so it is possible for the port to remain unbound during simulation. Making an interface method call through an unbound port is a run-time error, so it would be safest to check the binding first:

```
if (opt_port.get_interface()) opt_port->IMC();
```

Port policies are checked independently for each port, and are not passed from parent to child when a port is bound to another port.

The object in question

The objects that are instantiated to form the module hierarchy all share a common base class **sc_object**. As discussed above, modules, port, exports and primitive channels can only be instantiated during elaboration, but other user-defined classes derived from **sc_object** may be instantiated during elaboration or simulation. This distinguishes the module hierarchy from the object hierarchy, with user-defined **sc_objects** and dynamic processes forming part of the object hierarchy, but not the static module hierarchy.

In the new standard, new methods have been added to class **sc_object** to support hierarchy traversal. SystemC has always had features for hierarchy traversal, but these were somewhat ad hoc and it was not clear whether they were user-level features. There is now a very simple but well-defined API for navigating around the object hierarchy and discovering the objects. The new API looks like this:

```
class sc_object
{
 public:
   const char* name() const;
   virtual const char* kind() const;
   ...
   virtual const
   std::vector<sc_object*>& get_child_objects() const;

   sc_object* get_parent_object() const;
};

const std::vector<sc_object*>& sc_get_top_level_objects();
sc_object* sc_find_object( const char* );
```

There is a strict parent-child relationship between all the objects of the object hierarchy. Any given object may have any number of child objects, or be childless. Top-level objects have no parent, and every other object has precisely one parent object so that the hierarchy forms a number of separate trees with no closed loops. Function **sc_get_top_level_objects()** retrieves all of the top level objects, and **get_child_objects()** retrieves all of the children of a given object. With these features in the new standard, it is now possible to traverse the entire object hierarchy using a simple code fragment as follows:

```
void recursive_descent( sc_object* obj )
{
 // Insert some specific action here
 ...
 std::vector<sc_object*> children =
                                    obj->get_child_objects();
 for ( unsigned i = 0; i < children.size(); i++ )
   recursive_descent( children[i] );
}

void traverse_hierarchy()
{
 std::vector<sc_object*> tops = sc_get_top_level_objects();

 for ( unsigned i = 0; i < tops.size(); i++ )
   recursive_descent( tops[i] );
}
```

The trick is that any classes derived from **sc_object** that may have children override the virtual function **get_child_objects()**. This includes **sc_module** and the classes associated with process instances. The default definition of **get_child_objects()** in **sc_object** returns an empty vector, because only modules and processes have children.

Function **sc_find_object()** permits an object to be found given its hierarchical name, and function **get_parent_object()** does just what its name would suggest.

Having retrieved an instance of an **sc_object**, one may then need to identify the kind of the object in order to call methods specific to the class of the object: module, port, export, primitive channel or process. Given a pointer to an object, the kind of object can be identified by calling its **kind()** method, by using RTTI, or by attempting a downcast. Here are examples of each:

```
sc_object* obj = sc_find_object("foo.foobar");
cout << obj->kind() << endl;
cout << typeid(*obj).name() << endl;    // RTTI
```

```
sc_module* m;
m = dynamic_cast<sc_module*>(obj);  // Downcast
if (m)
  ... // Now we know it's a module
```

As a use case for this feature, consider the following function to trace every bit-level signal below a given module instance:

```
void sc_trace(sc_trace_file* tf,
              const sc_module& mod, const std::string& txt)
{
  std::vector<sc_object*> ch = mod.get_child_objects();

  for ( unsigned i = 0; i < ch.size(); i++ ) {
    sc_object* obj = ch[i];

    sc_signal<bool>* sb;
    if (sb = dynamic_cast<sc_signal<bool>*>(obj))
      sc_trace(tf, *sb, sb->name());

    sc_signal<sc_logic>* sl;
    if (sl = dynamic_cast<sc_signal<sc_logic>*>(obj))
      sc_trace(tf, *sl, sl->name());

    sc_module* m;
    if (m = dynamic_cast<sc_module*>(obj))
      // Recursive descent...
      sc_trace(tf, *m, m->name());
  }
}
```

Note that the example above does the job of tracing all **bool** and **sc_logic** signals, but is far from an ideal solution because each candidate object type needs to be tested for explicitly with a dynamic cast. In order to trace signals of other kinds, further code must be added to the function. This inelegance comes about because of the general lack of any facility for *introspection* within C++. It is not possible to determine the data type and members of an object in a running program. For further information on introspection, search for "C++ introspection" on the web.

Getting a handle

It was mentioned above that processes of both the static and the dynamic variety form part of the object hierarchy. SystemC version 2.1 introduced the notion of process handles into SystemC for the first time, but these were confined to spawned processes and had limited functionality. In the new standard, process handles have been extended to all kinds of process and have some useful new functionality.

The reason for the process handle is to give the user an object through which they can access details of the process, and which will not be destroyed when the process is terminated. The process handle in the new standard treats unspawned and spawned processes in a uniform way. The more important member functions are as follows:

```
class sc_process_handle
{
public:
  ...
  explicit sc_process_handle( sc_object* );
  bool valid() const;
  const char* name() const;
  sc_curr_proc_kind proc_kind() const;
  const std::vector<sc_object*>& get_child_objects() const;
  sc_object* get_parent_object() const;
  sc_object* get_process_object() const;
  bool dynamic() const;
  bool terminated() const;
  const sc_event& terminated_event() const;
};
```

The code fragment given earlier for traversing the object hierarchy will visit the **sc_object** associated with every static and dynamic process. However, it is generally bad practice to rely on pointers to objects associated with dynamic processes, since the timing of the creation and deletion of such objects is implementation-defined. Process handles provide a safe mechanism for accessing process instances, both spawned and unspawned. (Indeed, the type **sc_process_b,** which was occasionally used in SystemC applications, is now deprecated)

Given a pointer to an **sc_object,** a process handle can be constructed as follows:

```
sc_object* obj = sc_find_object("foo.foobar");
sc_process_handle h = sc_process_handle(obj);
```

Function **sc_spawn** returns a process handle to the spawned process:

```
sc_process_handle h = sc_spawn(&proc, "proc");
```

Function **sc_get_current_process_handle()** returns a process handle for the currently executing process:

```
sc_process_handle h = sc_get_current_process_handle();
```

The validity of a process handle should always be tested before relying on its methods:

```
if (h.valid()) {
  std::vector<sc_object*> children = h.get_child_objects();
  for (unsigned i = 0; i < children.size(); i++)  {
    sc_process_handle ch = sc_process_handle(children[i]);
    if (ch.valid())
      ...
  }
}
```

The above code fragment finds all the child processes spawned from a given process. Note that since it is

implementation-defined when the **sc_objects** associated with process instances get deleted, it is always best to manipulate processes using process handles.

The process handle provides a function to determine whether the process is a method process, thread process or clocked thread process:

```
sc_process_handle h = sc_get_current_process_handle();
switch (h.proc_kind()) {
  case SC_METHOD_PROC_ : ...
  case SC_THREAD_PROC_ : ...
  case SC_CTHREAD_PROC_ : ...
}
```

This mechanism provides an efficient way to know what kind of process is being executed, and hence of knowing whether or not the function **wait()** or a blocking interface method may be called. This is an important consideration when building transaction-level interfaces. In principle, an interface method could determine whether it was executing in the context of a thread process or a method process, and then call either **wait()** or **next_trigger()** respectively. Alternatively, a method could bail out gracefully if called from the wrong context.

A thread process is *terminated* when control is returned from the associated function. This is a very useful concept in connection with spawned processes, because it allows some action to be taken when a process is completed. The new standard provides a simple mechanism to determine when a thread is terminated: the member functions **terminated()** and **terminated_event()** of the process handle class. These methods can be used to synchronize the termination of concurrent threads. For example:

```
sc_process_handle h1 = sc_spawn(&proc1, "proc1");
sc_process_handle h2 = sc_spawn(&proc2, "proc2");
wait(h1.terminated_event() & h2.terminated_event());
```

The above code fragment spawns two concurrent processes proc1 and proc2, then suspends until both processes have terminated, in any order. The effect is similar to the *fork-join* construct in Verilog. Note that the **sc_objects** associated with the spawned process instances may or may not have been deleted at the point when execution is resumed following the wait, but the process handles will certainly still exist.

One event or another

It would also be possible to resume a parent process when the first of a number of child processes terminates by using an *event or list* as follows. The effect is similar to the *fork-join_any* construct in SystemVerilog:

```
sc_process_handle h1 = sc_spawn(&proc1, "proc1");
sc_process_handle h2 = sc_spawn(&proc2, "proc2");
wait(h1.terminated_event() | h2.terminated_event());
```

The *event and list* and *event or list* are useful constructs for creating dynamic sensitivity to a number of different events. A use case that sometimes arises in transaction-level modeling is making a process sensitive to events in each of a number of channels, where the number is determined at elaboration time. Here the language seems to breaks down, because an expression cannot consist of a variable number of terms. A loop is necessary to combine the events from a variable number of channels, but writing such a loop requires access to the class **sc_event_or_list** that underlies the event list. But is this class a user-level feature or an artifact of the implementation? This question typifies the kind of issue faced during the SystemC standardization process.

The new standard clarifies that the following solution is allowed:

```
struct M: sc_module
{
  sc_port<i_f, 0> p;  // Multiport

  sc_event_or_list& all_events() {
    sc_event_or_list& or_list = p[0]->event() | p[0]->event();
    for (int i = 1; i < p.size(); i++)
      or_list | p[i]->event();
    return or_list;
  }
  ...
  wait(all_events());
  ...
```

The new standard forbids an application from creating an object of type **sc_event_or_list**. However, an application may create a reference to such a type. The *or-list* object itself is created by the operator | of class **sc_event**, then further events are added to the list using a side-effect of the operator. The function **all_events()** returns a reference to the complete *or-list*.

Another clarification concerns what the application can rely on regarding the lifetime of the *or-list* object. The application should not assume that the *or-list* will remain valid once the process has suspended. This is why in the example above the *or-list* is recreated dynamically on each call to **wait**.

In exceptional circumstances

SystemC version 2.1 opened up the internal error reporting mechanism of the OSCI simulator for use in applications. The new standard clarifies some points of usage and tidies up the mechanism for catching exceptions.

Early versions of the reporting mechanism used a unique integer identifier for each message type. The integer identifiers are now deprecated, and instead the new standard uses text strings to distinguish message types. End users can program the actions to be performed when a report is generated, dependent on both the severity of the report and the message type. The intent is that anyone generating reports will choose appropriate message types so that end users can customize the report handling with an appropriate degree of granularity. Any reports assigned the same message type will get handled in a similar way.

The new standard recommends that creators of pre-compiled SystemC models should choose message types that are likely to be globally unique, having the form:

"/company/product/subcategory/subcategory..."

Furthermore, these message types should be published in the documentation that accompanies the models. This step should minimize the probability of clashes between message types from different model sources, and should help clarify the origin of any messages encountered when running an application.

As mentioned briefly above, the new standard replaces the type **sc_exception** with the C++ standard type **std::exception**. The report handler now throws an object of class **sc_report**, which is derived from **std::exception**. A C++ exception handler can now interrogate the report object, as shown by the following example:

```
try {
  ... // SC_REPORT_ERROR called here
}
catch (sc_report rpt) {
  cout << rpt.get_severity() << endl;
  cout << rpt.get_msg_type() << endl;
  cout << rpt.get_msg() << endl;
  cout << rpt.get_file_name() << endl;
  cout << rpt.get_line_number() << endl;
  cout << rpt.get_time() << endl;
  cout << rpt.get_process_name() << endl;
  cout << rpt.what() << endl;
}
```

Give me the context

The OSCI simulator has an important class **sc_simcontext** that represents a running simulation. During the standardization process this class was declared an implementation artifact, not a user-level feature, so certain features needed to be provided in other ways. **sc_simcontext** is now deprecated. Instead, the new standard adds the following global functions:

```
sc_process_handle sc_get_current_process_handle();
const std::vector<sc_object*>& sc_get_top_level_objects();
sc_object* sc_find_object( const char* );
const sc_dt::uint64 sc_delta_count();
bool sc_is_running();
```

Several of these functions have been discussed earlier. Function **sc_delta_count()** returns the absolute number of delta cycles that have occurred during simulation, counting from zero. Function **sc_is_running()** returns true only when called during simulation.

Conclusion

The new SystemC standard offers a precise and complete specification of the SystemC class library. It offers a number of useful new features, as well as deprecating some bad old features. The OSCI open-source proof-of-concept simulator will continue to be maintained.

The new IEEE standard includes Annexes that list the changes made from earlier versions, and the deprecated features.

Enjoy!

References

1. The OSCI SystemC 2.1 Language Reference Manual, May 2005, available from http://www.systemc.org
2. IEEE Std 1666-2005 SystemC Language Reference Manual (not yet published)