SystemC Verification Standard Specification

Version 1.0e

Submission to SystemC Steering Group

May 16, 2003

Written by the Members
of the SystemC Verification Working Group

SystemC Verification Working Group's Active Participants:

Adam Rose, Motorola (Chairman)
Axel Braun, University of Tuebingen
Mark Glasser, Cadence
Thorsten Groetker, Synopsys
C. Norris Ip, Cadence
Mike Meredith, Forte
Hillel Miller, Motorola
Bill Paulsen, Cadence
Logie Ramachandran, Synopsys
Robert Siegmund, University of Chemnitz
Rob Slater, Motorola
Stuart Swan, Cadence

Contents

1 Introduction

This document describes the specification of the SystemC Verification Standard. The SystemC Verification Working Group has identified the applicable verification requirements, discussed proposals from various members, and settled on the current specification as the set of features to be incorporated into the SystemC Standard.

This document is not written exclusively from the user's point of view, but also includes explanation from a more complex developer's point of view. The number of classes & complexity that need to be exposed to the user is much simpler, and they are captured in table form within this document. This document mixes the specification of the API with examples & explanation of how to use it. This is necessary to get everyone to a common level of understanding, but perhaps adds to the perception of overall complexity, when in fact the proposed APIs to be standardized are relatively small. Separate LRM, User Guide, and Implementation Specification etc. will be created in the future.

The main items within the SystemC Verification Standard are:

- data introspection (manipulation of arbitrary data types)
- transaction-based verification (test bench modeling style and transaction recording)
- randomization
- constraints for randomization
- weights for randomization

A short description of many of the above concepts can be found in [1]. This specification assumes that the reader is familiar with SystemC 2.0. A general introduction to SystemC 2.0 can be found in [18].

There are several small items that perhaps should eventually be added to the core SystemC standard. They are included here in Section 7 because they are either particularly useful in a test bench or a pre-requisite for test bench modeling.

- basic HDL connection
- a small set of data structures
- exception handling
- debugging

The majority of Verification Working Group members consider these small items, while generic in nature, to be particularly important to functional verification. Without a standardized API for these facilities, it would be difficult to create verification IPs that are interoperable among multiple SystemC implementations. As a result, they are included in this specification, but we will leave it up to the SystemC Steering Committee to decide whether to allow these items to remain part of the SystemC Verification Standard or to move them to other SystemC Facilities.

Apart from the current specification, the SystemC Verification Working Group will continue to consider new features, such as those itemized in section 8. However, we think the current proposal contains sufficient materials for people to start writing effective test benches in SystemC.

2 Glossary

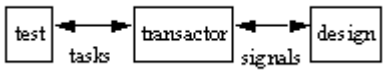
- **transaction**: A set of information that represents some bounded activity within the execution of a design or testbench. A transaction has a begin-time, an end-time, and a set of attributes (name-value pairs).
- **transactor**: An adaptor between a transaction-level test and a design typically at a different level. It is also referred to as a transaction verification model (TVM).
- **transaction-level tests**: Test case scenarios that are written in terms of transactions, calling transactor methods in an abstract transactor interface.
- **transactor interface**: An interface from which a test communicates to a transactor.
- **port interface**: A list of ports from which a transactor communicates to the design.
- **transactor method**: A method defined in a transactor interface.
- **transaction recording**: The process of storing transaction information into a database for post-simulation analysis.
- **automatic transaction recording**: A transactor modeling methodology that enables transactions to be recorded without the transactor writer consciously putting transaction recording code into the transactor. (The current specification does not specifically address this feature.)
- **transaction stream**: An object designed for transaction recording that groups related transaction information in the database. A transaction stream supports overlapping transactions. There is no implicit synchronization when manipulating a transaction stream. The test bench should exhibit the same run-time behavior with or without manipulation of transaction streams.
- **data introspection**: The ability to query a data object through a predefined abstract interface to find out what it is and how it can be manipulated.
- **interface introspection**: The ability to query a module or a channel through a predefined abstract interface to find out what it is and how it can be manipulated. (The current specification does not specifically address this feature.)
- **constraint**: Unless otherwise specified, it is a Boolean predicate representing a non-temporal constraint for the random value generation process. Because constraints and assertions are both Boolean predicates, sometimes people prefer to use the same Boolean predicate for both purposes.
- **assertion:** Unless otherwise specified, it is a Boolean predicate representing a non-temporal assertion for the detection of design errors during simulation. Because constraints and assertions are both Boolean predicates, sometimes people prefer to use the same Boolean predicate for both purposes.
- **temporal constraint and temporal assertion**: Temporal assertions are composed of multiple Boolean predicates and checks for conditions across a finite or infinite interval of time. Temporal constraints are composed of multiple Boolean predicates and generate multiple random values across a finite or infinite interval of time. (The current specification does not specifically address this feature.)

3 Overview

While the existing SystemC 2.0 standard can be used to perform basic verification of a design, the SystemC Verification Standard improves the capability by providing APIs for transaction-based verification, constrained randomization, exception handling, and other verification tasks. Using a generic data introspection capability, it enables the library to manipulate arbitrary data types in a consistent way, including C/C++ built-in types, SystemC built-in types, and user-defined composite types and enumerations. As a result, arbitrary data types can be used in variable recording, transaction recording, constraints, randomization, and other functions.

In this section, we provide an overview of the facility through an example of a transaction-based test bench. The design uses a pipelined bus, and a transactor is used to act as the adaptor between the transaction-level test and the signal-level design. The reader should keep in mind that this entire section is only an overview to capabilities that will be presented more completely later within this document. Because of this, this section is not a formal part of the specification of the Verification Standard.

Because the SystemC 2.0 standard supports communication refinement, a transactor can be modeled as an adaptor channel in SystemC 2.0.. Section 3.1 describes the structure of a transaction-based test bench in SystemC. As shown in following figure, a transaction-based verification methodology partitions the system into transaction-level tests, transactors, and the design. The tests and the transactors communicate through transactor method invocation, at a level above the RTL level of abstraction. The transactors and the design communicate through signal manipulation at the RTL/signal level.



Section 3.2 describes an example of dynamic concurrency in a test bench. While a design typically contains static concurrency to model hardware concurrency, a test bench does not have such a restriction and typically employs dynamic concurrency. The pipelined protocol in the example allows a maximum of two overlapping operations, and the corresponding test can use dynamic threads to generate concurrent activities. While the APIs for manipulation of dynamic threads are not part of the SystemC Verification Standard, they are an essential pre-requisite to the modeling of a test bench. We are currently coordinating our activities with the SystemC Language Working Group to create a standard for dynamic threads.

The remaining subsections describe an overview of various facilities in the SystemC Verification Standard, which includes transactor modeling and transaction recording, constrained randomization, HDL connection, and exception handling.

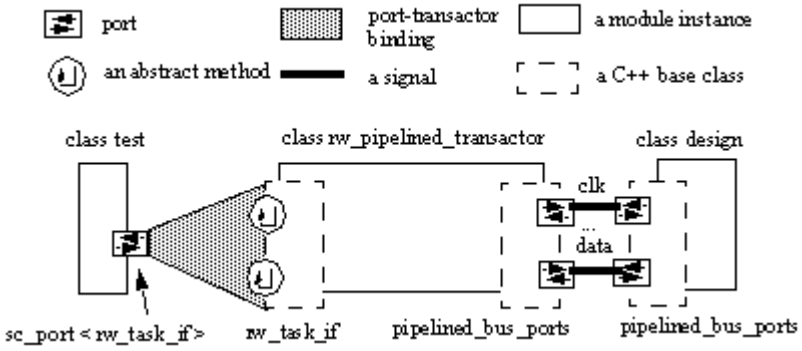
The SystemC Verification Standard is implemented in a name space to avoid name conflict. The tentative name of the name space is *systemc_verification_library*. The corresponding header file will contain a “using” statement, so the use of the name space will be transparent to the user. When there is a name clash, the user can specify the specific API by appending the name of the name space to the call.

The classes and functions in the SystemC Verification Standard use the prefix *scv_*.

3.1 Transactor Modeling Style

The VWG considers transaction-based verification to be a main strategy in functional verification using SystemC. By creating a test bench at the transaction level, the test bench can be used in both system-level simulation and RTL-level simulation, using a user-defined channel. The code in this section uses the SystemC 2.0 standard only. The complete source code for this example is in appendix C.

In this section, we illustrate this methodology through a transaction-based test bench for a design with a pipelined bus interface, as shown in Figure 1b.



Using abstract methods in C++, the transactor interface can be declared as follows:

```
class rw_task_if : virtual public sc_interface {
public:
        typedef sc_uint<64> addr_t;
        typedef sc_uint<64> data_t;
        struct write_t {
                addr_t addr;
                data_t data;
        };
        virtual data_t read( const addr_t * ) = 0;
        virtual void write( const write_t * ) = 0;
};
```

This abstract base class specifies two abstract methods, *read* and *write*, and their related data types, representing the abstract level in which a test is to be written. The class *sc_interface* is provided by SystemC to facilitate the creation of such interfaces. The template *sc_uint* and other similar templates are provided by SystemC to support data objects with different bit widths and different operator semantics.

The communication between the transactor and the design is captured in another class with signal-level ports:

```
class pipelined_bus_ports : public sc_module {
public:
        sc_in<bool> clk;
        sc_inout<bool> rw;
        sc_inout<bool> addr_req;
        sc_inout<bool> addr_ack;
        sc_inout< sc_uint<8> > bus_addr;
        sc_inout<bool> data_rdy;
        sc_inout< sc_uint<8> > bus_data;
};
```

The signals in this class represent the RTL-level interface by which a design under verification communicates to its environment. The class *sc_module* is provided by SystemC to specify a module. Signal ports are created by using the templates *sc_in, sc_out*, and *sc_inout*, indicating a read-only port, a write-only port, and a read-write port respectively.

A transaction-based verification methodology relies on transactors to act as the adaptors between the abstract tests and the concrete design. By capturing these transactors as reusable IP, new tests with complex concurrent behavior can be quickly created.

In this example, a transactor is created as a class deriving from both aforementioned interfaces:

```
class rw_pipelined_transactor
:       public rw_task_if,
        public pipelined_bus_ports {
        …
public:
        SC_CTOR(rw_pipelined_transactor) {}
        virtual data_t read( const addr_t *);
        virtual void write( const write_t *);
};
```

The macro *SC_CTOR* is provided by SystemC to specify the constructor of a module. The implementation of read() and write() convert the transaction-level operations to lower-level activities in the signal-level ports. With the detailed protocol abstracted by the transactor, a test can be written in a form independent of the actual signal-level interface, similar to the following code:

```
class test : public sc_module {
public:
        sc_port< rw_task_if > transactor;
        SC_CTOR(test) { SC_THREAD(main); }
        void main();
};

void test::main() {
        // simple sequential tests
        for (int i=0; i<3; i++) {
                rw_task_if::addr_t addr = i;
                rw_task_if::data_t data = transactor->read(&addr);
                cout << "received data : " << data << endl;
        }
        …
}
```

The test above generates a series of three read transactions, starting a new one only after the previous one has completed. The macro *SC_THREAD* creates a new thread of execution for the method *main()* during simulation.

It is important to note that the port of this test has the *rw_task_if* interface as the template argument. Because of this, the test can be reused with other designs with a different bus interface, by plugging in an appropriate

transactor for the bus interface. The *rw_task_if* interface can be made even more general and reusable by using a template, for example by putting the width of address and data into template parameters instead of having a fixed value.

The code for the RTL design uses the standard SystemC RTL modeling style.

```
class design : public pipelined_bus_ports {
        …
public:
        SC_CTOR(design) {
                ...
                SC_THREAD(addr_phase);
                SC_THREAD(data_phase);
        }
        void addr_phase() { while (1) ... }
        void data_phase() { while (1) ... }
};
```

The design implementation contains the same set of signals for the pipelined interface, with two C++ threads to respond to the two phases of the pipeline.

The netlist connection for the transaction-based verification methodology uses the user-defined channel connection for the transactor:

```
int sc_main(int argc, char *argv[ ]) {
        ...
        // The module and channel structures in this simulation
        test t ("t"); // the test
        rw_pipelined_transactor tr ("tr"); // the transactor
        design duv ("duv"); // the design under verification
        sc_clock clk ("clk",20,0.5,0,true); // a clock

        // The signals to connect the structures
        sc_signal < bool > rw;
        sc_signal < bool > addr_req;
        …

        // Connecting the signals and transactors to
        // the ports of the modules
        t.transactor = tr;
        tr ( clk.signal(), rw, addr_req, ... );
        duv ( clk.signal(), rw, addr_req, ... );

        // Start the simulation
        sc_start(10000);
        ...
}
```

Strictly speaking, this arrangement relies on the ability to drive the same signal (sc_signal) from more than one thread. When multiple tests or multiple dynamic threads are calling the same transactor methods in a transactor, the signals will be driven by different threads. However, because there is only one driver (either the transactor or the design) for each signal, it is not necessary to use resolved signals. Assignments from different threads are serialized by the synchronization within the transactor. In this use model, we would like the values to be updated with the last-assignment-wins semantic, which happens to be how *sc_signal*s currently work in SystemC 2.0. Therefore, we propose extending the SystemC standard to allow sc_signal to be explicitly configured to support this use model.

On the other hand, using a resolved signal will give you better error-detection, but it would require more code and leads to slower performance. For example, when a misbehaving design tries to drive a signal while a transactor is also driving it, collisions can be detected with a resolved signal. However, when a resolved signal is used, code must be added to set the signal to high-impedance when it is no longer being driven by the current thread. A resolved signal must be a logic bit or a logic vector, so 2-state data types and arithmetic cannot be used directly, and explicit conversion must be made to an arithmetic type.

3.2 Modeling Dynamic Concurrency

Pipelined protocols, split protocols and other scenarios can be conveniently modeled using dynamic concurrency. Therefore, a facility to handle dynamic threads would be a useful extension to SystemC. There is currently an example in the examples directory SystemC 2.0.1 reference implementation that implements a prototype of dynamic thread facility (see $SYSTEMC/examples/systemc/forkjoin). The file *sc_fork.h* in the example directory defines *sc_spawn_method()* and *sc_spawn_function()*, which are dynamic spawning enhancements that are not part of the SystemC 2.0 standard, but rather implemented on top of it. Both functions actually use a pool of threads which are statically declared at the beginning of simulation. We have not included this facility in the Verification standard, but we would like to use it in this example to illustrate how dynamic concurrency can be used in verification.

Using *sc_spawn_method*, a test generating concurrent activities can be implemented as shown in the following code. Three processes are spawned in the example below: Two read tasks that need to be synchronized (i.e. we need to wait until both tasks complete before moving on) and a write task that can run as long as it needs to (perhaps indefinitely):

```
void test::main() {
        …
        // Simple concurrent tests
        rw_task_if::addr_t addr[ 3 ]; addr[ 0 ] = 0; addr [ 1 ] = 1; addr[2] = 2;
        rw_task_if::data_t data[ 3 ]; data[2] = 4;

        sc_join_handle readHandle1 = sc_spawn_method( &data[ 0 ], transactor[ 0 ], &rw_task_if::read, &addr[ 0 ]);
        sc_join_handle readHandle2 = sc_spawn_method( &data[ 1 ], transactor[ 0 ], &rw_task_if::read, &addr[ 1 ]);
                // The only reason sc_join_handle is needed is if we want to synchronize the task later.
                // Since in our example we don't need to synchronize the write, we can cast the return
                // value of the sc_spawn_method() call below to void
        sc_spawn_method((void*)0, transactor[0], &rw_task_if::write, &addr[2], &data[2]);

        // Synchronize the two read tasks; the write task keeps going
        sc_process_join(readHandle1);
```

```
        sc_process_join(readHandle2);
        cout << "received data : " << data[ 0 ] << ","  << data[ 1 ] << "\n";
}
```

This code generates two read transactions in parallel, so that they exercise the pipeline. The *sc_spawn_method* function is similar to *create()* in PThreads, creating a new C++ thread and executing the method specified in the third argument for the object in the second argument. The first argument is a pointer to the object in which the return value is stored, and the last argument is the address of argument to the method. Note that *sc_spawn_method* can take between zero to four arguments to be passed to the corresponding method (in the example above, *&addr[2]* and *&data[2]* are the arguments to the *rw_task_if::write()* method).

The SystemC Verification Working Group has discussed a more comprehensive set of APIs to support dynamic threads, but because of the overlap with the current activities in the Language Working Group we have not reached a conclusion and therefore are not able to include a specification in this document. Long term, the Language Working Group needs to provide a truly dynamic thread spawning system. When they do, it is likely that the API will differ from the one shown here, but it will probably include similar features.

3.3 Transaction Manipulation and Recording

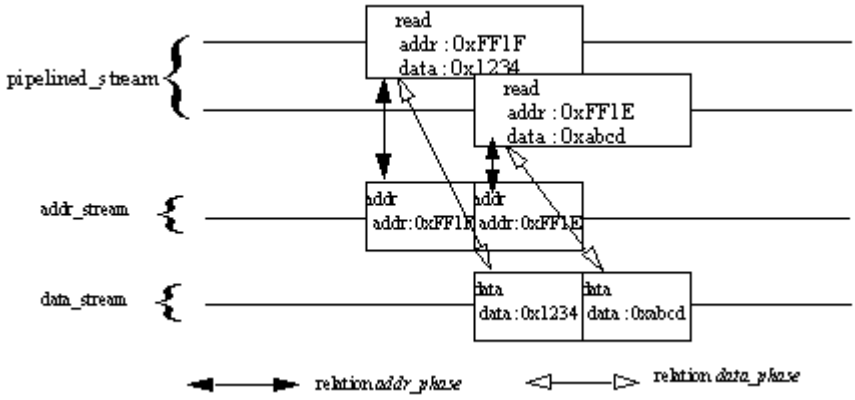
The job of a transactor is to convert a transaction as modeled by a function call into signal-level communication, and vice versa. For example, the *read* method is implemented as follows:

```
data_t rw_pipelined_transactor::read( const addr_t * addr) {
        addr_phase.lock();
        bus_addr = *addr;
        addr_req = 1;
        wait ( addr_ack->posedge_event() );
        addr_req = 0;
        wait ( addr_ack->negedge_event() );
        addr_phase.unlock();

        data_phase.lock();
        wait ( data_rdy->posedge_event() );
        data_t data = bus_data.read();
        wait ( data_rdy->negedge_event() );
        data_phase.lock();

        return data;
}
```

This method translates a call to the *read()* method into a series of signal activities according to the specific protocol at the signal-level interface. Because a pipelined bus is used, two mutexes, *addr_phase* and *data_phase*, are used for the two phases. Both of them grant access in a first-come-first-served manner. At the beginning of the address phase, the corresponding mutex is locked so that there is at most one address communication on the bus at any given time. At the end of the address phase, the corresponding mutex is unlocked so that, although the data phase has not finished yet, another call to this transactor can still start its address phase. The three methods, *posedge_event(), negedge_event(),* and *read(),* are provided by SystemC to

denote a positive edge transition of a signal, a negative edge transition of a signal, and access to the value in a port. The procedure *wait()* in SystemC suspends the thread of execution until the event specified in the argument occurs.

A read operation with this pipelined protocol generates three conceptual transactions, a read transaction and two sub-transactions corresponding to the two phases of the pipelined protocol. These transactions are captured in the following diagram:



Using the SystemC Verification Standard specified later in this document, you can capture this information by adding the transaction recording code into the transactor:

```
class rw_pipelined_transactor : public pipelined_bus_ports, public rw_task_if {
public:
        scv_tr_stream pipelined_stream;
        scv_tr_stream addr_stream;
        scv_tr_stream data_stream;
        scv_tr_generator< addr_t, data_t > read_gen;
        scv_tr_generator< addr_t, data_t > write_gen;
        scv_tr_generator< addr_t > addr_gen;
        scv_tr_generator< data_t > data_gen;
        SC_CTOR(my_transactor) :
                pipelined_stream(“pipelined_stream”),
                addr_stream(“addr_stream”),
                data_stream(“data_stream”),
                read_gen(“read”,pipelined_stream, “addr”,”data”),
                write_gen(“write”,pipelined_stream,”addr”,”data”),
                addr_gen("addr",addr_stream,"addr"),
                data_gen("data",data_stream, "data") { ... }
        virtual data_t read( const addr_t * addr) {
                address_phase.lock();
                scv_tr_handle h = read_gen.begin_transaction(*addr);
                scv_tr_handle h1 = addr_gen.begin_transaction(*addr, “addr_phase”, h);
                ...// address phase
                addr_gen.end_transaction(h1);
                addr_phase.unlock();
```

```
            data_phase.lock();
            scv_tr_handle h2 = data_gen.begin_transaction("data_phase",h);
            ...// data phase
            data_gen.end(h2,data);
            read_gen.end_transaction(h, data);
            data_phase.unlock();
            return data;
        }
};
```

This code creates three transaction streams in the database and creates four transaction generators, *read_gen, write_gen, addr_gen*, and *data*_gen, corresponding to the four types of transactions in this protocol. During execution of the read method, the methods *begin_transaction()* and *end_transaction()* start and terminate transactions respectively. The address and the data are recorded as attributes of these transactions.

The SystemC Verification Standard relies on a static extension of the data types *addr_t* and *data_t* to access and record the values of the transaction attributes, as described in Section 4.

3.4 Constrained Randomization

The job of a test is to create interesting stimulus to exercise the design. Apart from directed tests, constrained random tests are also very important. The SystemC Verification Standard contains a set of APIs to support constrained randomization.

For example, random addresses for the read operation can be created by using a simple *next()* call to a smart pointer object:

```
scv_smart_ptr< rw_task_if::addr_t >  addr;
for (int k = 0; k<100; k++) {
        addr->next();
        cout << "data for address " << *addr << " is " << transactor[ 0 ]-> read ( addr->get_instance() ) << end;
}
```

Similarly, random write requests for our *rw_pipelined_transactor* transactor in Section 4 can be generated by the following code:

```
scv_smart_ptr < rw_task_if::write_t > write;
for (int i=0; i<3; i++) {
        write->next();
        transactor->write( write->get_instance() );
        cout << "send data : " << write->data << endl;
}
```

The *scv_smart_ptr* template has a use model that is very similar to a C/C++ pointer. Fields and methods are accessed by the overloaded operator->. The *next()* method generates a new random value, and the field access returns a reference to the extension of the *data* field.

Expressions and constraints can be created using *scv_smart_ptr* as well. For example, the following code specifies an address range and a relationship between the address and the data:

```
class write_constraint : virtual public scv_constraint_base {
public:
        scv_smart_ptr< rw_task_if::write_t > write;
        SCV_CONSTRAINT_CTOR(write_constraint) {
                SCV_CONSTRAINT( write->addr() < 0x00FF );
                SCV_CONSTRAINT( write->addr() != write->data() );
        }
};
```

Declaring the constraints as classes allows the constraints to be processed once for high-speed randomization. It also allows an object-oriented way to manage constraints, using hierarchy and inheritance.

Biased randomization using weights can be performed by using a bag. A bag is similar to the concept of a set in mathematics, except that it can contain duplicated objects. For example, generating the value "1" 40% of the time and the value "2" 60% of the time can be achieved using the following code:

```
scv_smart_ptr<int> data;
scv_bag<int > distribution;
distribution.push( 1, 40);
distribution.push( 2, 60);
data->set_mode(distribution);
for (int i=0; i<3; i++) {data->next(); process(data); }
```

3.5 Miscellaneous

The SystemC Verification Standard also includes other features that facilitate functional verification, such as HDL connection, special data structures, exception handling, and debugging.

The API for HDL connection allows replacing the SystemC design with an HDL design in Verilog or VHDL. The *sc_signal* signals at the signal-level port of the transactor can be connected to the HDL signals in the HDL design through the following connection calls:

```
scv_connect(rw, "top.rw", SCV_OUT);
scv_connect(addr_req, "top.addr_req", SCV_OUT);
```

The following code illustrates how verification models can use the exception-handling class to report IP-specific exceptions:

```
#define BAD_DATA "BAD_DATA"
rw_task_if::data_t rw_pipelined_transactor::read(* rw_task_if::addr_t * addr) {
        if (bad_data)
                SCV_REPORT_ERROR(BAD_DATA,
                        "bad data is detected within read() of rw_pipelined_transactor");
}
```

In this example, a report is generated through the standard API when bad data is detected within the task. As a result, the library and other associated tools can use these messages to improve debugging and analysis.

The following example illustrates the debugging abstract interface during a C++ debugger session. By presenting the abstract information in a simple method call, the user does not have to go through the internal data members of various objects to determine their current state.

```
gdb> call packet.show()
src : 0x1000
dest : 0x00FE
payload : 0xabcd
```

4 Manipulation of Arbitrary Data Types

The SystemC Verification Standard uses a technique called data introspection to enable the manipulation of arbitrary data types. It allows a library routine to extract information from data objects of arbitrary data types, regardless of whether it is a C/C++ built-in type, a SystemC built-in type, a user-specified composite type (struct), or a user-specified enumeration. Similar techniques can be found in several articles on C++ [4,5,6,7,8]. The original suggestion for the data introspection technique used in SCV based on C++ partial template specialization came from Grzegorz Jakacki [11]. Before we dive into the details in sections 4.1, etc., let's look at the motivation and requirements first.

For example, the *rw_task_if* interface shown in Section 3.1 uses a SystemC data class, *sc_uint<64>,* and a composite type, *write_t*. We would like a C++ library (or a user) to be able to extract information from these types and manipulate them without having to modify the source code.

The data introspection facility introduced later in this section provides a standard abstract interface, *scv_extensions_if*, from which a data object can be analyzed and manipulated. However, while traditional C++ libraries require the user to use a similar interface class as the base class of their composite type, we prefer to use partial template specialization to attach this interface to the data objects. This style supports a wider range of data types. It enables import of legacy code without modification, and allows the same pieces of code to work on built-in types such as int, library types such as *sc_uint<>,* and composite types such as a user-defined packet type with multiple fields.

This standard abstract interface provides the following functionality so that a piece of code can manipulate a data object without explicit type information at compile-time:

- type information extraction
- value access and value assignment
- randomization
- callback registration

This facility can be considered as a C++ version of the Verilog PLI standard. The ability to handle arbitrary data types enables the import of legacy code and facilitates the reuse of the same code for multiple libraries. It is a crucial basic building block for constrained randomization, variable recording, and transaction attribute recording. This section will describe the API and some simple examples; its usage is also shown in the corresponding sections on the constrained randomization API and recording API.

This facility includes the following classes and templates:

- The *scv_extensions_if* abstract interface: This abstract interface enables the manipulation of arbitrary data types without compile-time type information.
- The *scv_extensions* template: Data objects are extended to support the abstract interface through partial template specialization of this template
- The *scv_shared_ptr* template: This template enables sharing of data objects among multiple threads, with reference counting to perform automatic memory management.
- The *scv_smart_ptr* template: This template combines *scv_extensions* and *scv_shared_ptr* to implement dynamic extensions that require instance-specific auxiliary data, such as randomization and callback handling.

This facility utilizes partial template implementation, which is supported by gcc 2.95, aCC 3.31, and Sun Compiler Version 7. It does not yet work on Visual C++ (we have tried it on version 6 and .net). The SystemC Verification Working Group's consensus is that we will continue to rely on this partial template specialization technique, and trust that Visual C++ will eventually support it. An internet posting from Microsoft Corporation

[17] has indicated that this issue is one of the first things they will fix in subsequent releases of Visual C++. The following paragraph is a direct quote from the posting:

> *Many modern C++ libraries—including the C++ standard library—require template partial specialization. Because our shipping compiler is lacking here, these libraries either don't work at all with Visual C++, or work with crippled semantics. You can expect that this is one of the first things we'll fix in subsequent releases of Visual C++.*

We have noticed that while many downloads of SystemC have been installed on Windows for evaluation purposes, most production work is still performed on Unix. We may support a subset of data types that use full template specialization on Windows so that our customers can still try out the Verification Standard on Windows.

4.1 Manipulating Data Objects Without Compile-Time Information

Manipulation of data objects without compile-time information is supported through the *scv_extensions_if* abstract interface. The postfix *"_if"* indicates that this is an abstract interface class with C++ abstract methods and without member variables.

The set of abstract methods in *scv_extensions_if* is partitioned into multiple components. This organization will make it easy to add more extensions in the future. The current specification contains the following components:

- *component util*: Basic utility methods
- *component type*: Methods to extract type information
- *component rw*: Methods to read and write to the data object, its fields, and its array elements
- *component rand*: Methods for randomization-related operations; for details about its usage, please see Section 5.
- *component callbacks*: Methods for callback registration (e.g. value change); for details about its usage, please see Section 6.

While conceptually these methods are part of *scv_extensions_if*, in the implementation of these components, each component can be declared as a separate class in a linear inheritance hierarchy, and we use *scv_extension_util_if, scv_extension_type_if, etc.* to illustrate the organization of *scv_extensions_if*. Although the actual implementation might use a different architecture, the suggested architecture with multiple components enables the addition of new components without requiring users to modify their test bench code (the code needs to be recompiled though). Components can be removed and added by using a macro trick: The interface of each component is declared with a macro representing its base class. After the interface is defined, the component undefines the macro and puts itself into the macro. The following example illustrates this trick:

```
template<>
class scv_extension_util_if: public SCV_EXTENSION_BASE {
        ...
};
#undef SCV_EXTENSION_BASE
#define SCV_EXTENSION_BASE scv_extension_util_if
```

Using this linear arrangement instead of using multiple inheritances to derive *scv_extensions_if* from these components, the overhead of virtual table pointers is kept to a minimum (c.f. [11]). The user of the Verification

Standard does not need to know this implementation detail; it is provided here to illustrate the benefit of using data introspection to extend data types.

The *scv_extensions_if* class also contains the debugging interface, *scv_object_if*, as discussed in Section 1.

The abstract methods in these components are described in the following tables:

scv_extension_util_if	Description

virtual bool has_valid_extensions() {
  return false;
}
	This method returns true if the extension object is a valid extension of the corresponding data object. This information is useful in the cases when the user forgot to create the template specialization for a composite type or an enumeration type.

virtual bool is_dynamic() {
  return false;
}
	This method returns true if the extension object supports dynamic extensions. See Section 4.3 for a discussion on dynamic extensions.

virtual void set_name(const char *) = 0;	This method sets the name for the data object. It is an error if this method is called more than once for a specific object.

virtual const char * get_name() {
  return NULL;
}
	This method returns the name of the data object.

virtual const char * get_short_name() {
  return NULL;
}
	This method returns the short name of the data object. If the data object is a field within a record, the short name is the field name. If the data object is an element within an array, the short name is of the form "[*n*]", where *n* is the index corresponding to this element.

scv_extension_type_if	Description
virtual const char * get_type_name() const = 0;	This method returns a string corresponding to the type name in C++.

enum data_type {
  BOOLEAN,
  ENUMERATION,
  INTEGER,
  UNSIGNED,
  FLOATING_POINT_NUMBER,
  BIT_VECTOR,
  LOGIC_VECTOR,
  FIXED_POINT_INTEGER,
  UNSIGNED_FIXED_POINT_INTEGER,
  RECORD,
  POINTER,
  ARRAY,
  STRING
);
	This enumeration represents the possible types of the data object.

virtual data_type get_type() const = 0;	This method returns the object type as captured in the enumeration.
virtual int get_bitwidth() const = 0;	This method returns the number of bits for the object type.
bool is_bool() const;	This method returns true if type is BOOLEAN.
bool is_enum() const;	This method returns true if type is ENUMERATION.
virtual int get_enum_size() const = 0;	This method returns the number of elements in the enumeration. It returns 0 if this extension is not an

	enumeration.
virtual void get_enum_detail(list<const char *>&, list<int>&) const = 0;	This method sets the arguments to the string representation and the integer representation of the enumeration. It clears the lists if this extension is not an enumeration.
bool is_integer() const;	This method returns true if type is INTEGER.
bool is_unsigned() const;	This method returns true if type is UNSIGNED.
bool is_bit_vector() const;	This method returns true if type is BIT_VECTOR.
bool is_logic_vector() const;	This method returns true if type is LOGIC_VECTOR.
bool is_fixed() const;	This method returns true if type is FIXED_POINT_INTEGER.
bool is_unsigned_fixed() const;	This method returns true if type is UNSIGNED_FIXED_POINT_INTEGER.
bool is_floating_point_number() const;	This method returns true if type is FLOATING_POINT_NUMBER.
bool is_record() const;	This method returns true if type is RECORD.
virtual int get_num_fields() const = 0;	This method returns the number of fields in a record. It returns 0 if this extension is not a record.
virtual scv_extensions_if * get_field(unsigned) = 0;	This method returns the extension for a specific field in a record. It returns 0 if the field does not exist or if this extension is not a record. This method treats the composite data object as a flat object, without distinguishing where a field comes from within the class hierarchy.
virtual const scv_extensions_if * get_field(unsigned) const = 0;	This method returns the extension for a specific field in a record. It returns 0 if the field does not exist or if this extension is not a record. This method treats the composite data object as a flat object, without distinguishing where a field comes from within the class hierarchy.
bool is_pointer() const;	This method returns true if type is POINTER.
virtual scv_extensions_if * get_pointer() = 0;	This method returns the extension of the object that is pointed to by this extension. It returns 0 if this extension is not a pointer.
virtual const scv_extensions_if * get_pointer() const = 0;	This method returns the extension of the object that is pointed to by this extension. It returns 0 it this extension is not a pointer.
bool is_array() const;	This method returns true if type is ARRAY.
virtual int get_array_size() const = 0;	This method returns the number of elements in the array. It returns 0 if this extension is not an array.
virtual scv_extensions_if * get_array_elt(int) = 0;	This method returns the extension for a specific array element. It returns 0 if the array element is invalid or if this extension is not an array.
virtual const scv_extensions_if * get_array_elt(int) const = 0;	This method returns the extension for a specific constant array element. It returns 0 if the array element is invalid or if this extension is not an array.
bool is_string() const;	This method returns true if type is STRING.

The *data_type* enumeration has the following mapping:

- BOOLEAN: bool.

- ENUMERATION: user-defined enumeration.
- INTEGER: char, short, int, long, long long, sc_int, sc_bigint.
- UNSIGNED: unsigned char, unsigned short, unsigned int, unsigned long, unsigned long long, sc_uint, sc_biguint.
- FLOATING_POINT_NUMBER: float, double.
- BIT_VECTOR: sc_bit, sc_bv.
- LOGIC_VECTOR: sc_logic, sc_lv.
- FIXED_POINT_INTEGER: sc_fixed.
- UNSIGNED_FIXED_POINT_INTEGER: sc_ufixed
- RECORD: user-defined struct and class.
- POINTER: T*
- ARRAY: T[n]
- STRING: string, sc_string.

More methods might be added to extract the parameters of *sc_fixed* and *sc_ufixed*. The non-virtual methods in this component are basically methods built on top of other virtual methods.

scv_extension_rw_if	Description
virtual void assign(arg) = 0;	A virtual assignment operator. This method is actually a series of overloaded *assign()* methods, each taking an argument of a different type. The argument arg can be one of the following types: bool, char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, double, const string&, const sc_string&, const sc_bv_base&, and const sc_lv_base&.
virtual void get_value(arg&) const = 0;	A virtual value access operator. This method is actually a series of overloaded *get_value()* methods, each taking an argument of a different type. The argument arg can be one of the following types: bool, char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, double, string, sc_string, sc_bv_base, and sc_lv_base.
virtual bool get_bool() const = 0;	An alternative to *get_value()* without an argument.
virtual long long get_integer() const = 0;	An alternative to get_value() without an argument.
virtual unsigned long long get_unsigned() const = 0;	An alternative to get_value() without an argument.
virtual double get_double() const = 0;	An alternative to get_value() without an argument.
virtual sc_string get_string() const = 0;	An alternative to get_value() without an argument.

It is an error if the type of assignment or value access is incompatible with the type of the underlying data object.

scv_extension_rand_if	Description
enum mode_t {	This enumeration represents the possible styles of

RANDOM, SCAN, RANDOM_AVOID_DUPLICATE, DISTRIBUTION };	value generation.
virtual void enable_randomization() = 0;	This method turns on randomization for this data object. This method is used typically for a field or an array element within a composite type. This method affects randomization of an object that is part of a complex constraint.
virtual void disable_randomization() = 0;	This method turns off randomization for this data object.
virtual bool is_randomization_enabled() const = 0;	This method returns true if randomization is not turned off for this data object.
virtual void next() = 0;	This method assigns a new value to the data object according to the associated value generation mode, if randomization has not been turned off.
virtual void set_random(scv_shared_ptr<scv_random>) = 0;	This method assigns a new random stream to a data object. See Section 5.1 for details.
virtual scv_shared_ptr<scv_random> get_random() const = 0;	This method returns the random stream associated with a data object.
virtual scv_expression form_expression() const = 0;	This method creates a basic expression from the data object for future evaluation and manipulation.

The mode_t enumeration has the following meanings:

- RANDOM: Uses a uniform distribution across the range of all legal values (low overhead).
- SCAN: Keeps a history and starts with the smallest legal values (medium overhead).
- RANDOM_AVOID_DUPLICATE: Keeps a history and avoids duplicated values until all legal values have been generated; when all legal values have been generated, resets history (high overhead).
- DISTRIBUTION: Takes a user-specified (step-like) distribution, and uses it to bias the randomization process (overhead depends on the distribution).

scv_extension_callbacks_if	**Description**

enum callback_reason {
VALUE_CHANGE,
DELETE
};
	This enumeration represents the possible reasons for executing a callback.
	VALUE_CHANGE: the callback is executed because a new value has been assigned to the data object.
	DELETE: the callback is executed because the data object has been deleted.

typedef int callback_h;	This component has an associated type for the callback handle. The reference implementation will implement it as an integer. This handle can be used to remove a callback after it has been registered.
virtual callback_h register_cb(void (*f)(scv_extensions_if&, callback_reason)) = 0;	This method registers a simple callback function. Multiple callback functions can be registered and they will be executed in the order in which they

	were registered.

template<typename arg_type> callback_h register_cb(
  void (*f)(scv_extensions_if&, callback_reason, arg_type),
  arg_type arg
);
	This template method registers a callback function with an extra argument in a type-safe manner. Multiple callback functions can be registered and they will be executed in the order in which they were registered.

virtual void remove_cb(callback_h) = 0;	This method removes an existing callback.

The *scv_extensions_if* interface is an interface derived from this list of component interfaces. The *scv_extensions_if* interface also contains the debugging interface discussed in Section 1. This interface can be used to recursively traverse every field and array element in nested composite types, as shown in the following example:

```
void print_fields(scv_extensions_if * e) {
        switch(e->get_type()) {
        case scv_extensions_if::RECORD :
                { for (int i=0; i < e->get_num_fields(); ++i) { print_fields(e->get_field(i)); } }
        case scv_extensions_if::ARRAY :
                { for (int i=0; i < e->get_array_size(); ++i) { print_fields(e->get_array_elt(i)); }
        case ...
        }
}
```

4.2 Defining the Extensions for Data Types

The data introspection facility depends on partial template specialization of a template called *scv_extensions* to extend data objects with the abstract interface *scv_extensions_if*. Each specialization of the *scv_extensions* template implements the *scv_extensions_if* interface in a way appropriate to the type in the template parameter. The target list of data types supported by the data introspection facility is shown in the following table.

Data Type	Partial Template Specialization
bool	class scv_extensions<bool>
char	class scv_extensions<char>
short	class scv_extensions<short>
int	class scv_extensions<int>
long	class scv_extensions<long>
long long	class scv_extensions<long long>
unsigned char	class scv_extensions<unsigned char>
unsigned short	class scv_extensions<unsigned short>
unsigned int	class scv_extensions<unsigned int>
unsigned long	class scv_extensions<unsigned long>
unsigned long long	class scv_extensions<unsigned long long>
float	class scv_extensions<float>
double	class scv_extensions<double>
string	class scv_extensions<string>
pointer	class scv_extensions<T*>
array	class scv_extensions<T[N]>

sc_string	class scv_extensions<sc_string>
sc_bit	class scv_extensions<sc_bit>
sc_logic	class scv_extensions<sc_logic>
sc_int	template<int W> class scv_extensions< sc_int<W> >
sc_uint	template<int W> class scv_extensions< sc_uint<W> >
sc_bigint	template<int W> class scv_extensions< sc_bigint<W> >
sc_biguint	template<int W> class scv_extensions< sc_biguint<W> >
sc_bv	template<int W> class scv_extensions< sc_bv<W> >
sc_lv	template<int W> class scv_extensions< sc_lv<W> >
sc_fixed	template<int W, int I, sc_q_mode Q, sc_o_mode O, int N> class scv_extensions< sc_fixed<W,I,Q,O,N> >
sc_ufixed	template<int W, int I, sc_q_mode Q, sc_o_mode O, int N> class scv_extensions< sc_ufixed<W,I,Q,O,N> >

These template specializations include appropriate operators so that they behave as if they are the underlying data objects. For example, most of them include the operators +=, -=, *=, /=, %=, ^=, etc., and the specializations for integer types include operators ++, --, <<=, and >>=. The specializations for SystemC types include the corresponding methods in the underlying object, such as *to_int64()*. These operators are required to support value change callbacks, etc. Implicit conversion to the underlying data type is used whenever possible.

Similar to *read()* and *write()* in a SystemC port (sc_port), these extension classes also have *read()* and *write()* to get around the implicit conversion problem. In most cases, the C++ compiler can perform the implicit conversion automatically. In the cases when the compiler cannot deduce the right conversion, *read()* and *write()* can be used. Another method called *get_instance()* returns a non-constant pointer to the underlying object; this method must be used in conjunction with another method *trigger_value_change_cb()* to make sure value change callbacks are executed correctly.

Data introspection is useful only for normal data types, such as integers, structs, and arrays. It does not make sense to use the data introspection API on channels such as sc_signal, especially when they have a specific semantic associated with the object, instead of being a simple data object. One could imagine a future API in SystemC that allows one to traverse through the design structure, get a handle to a FIFO, and then use the data introspection to make queries about the current *values* stored within the FIFO, but this would be a separate API.

Defining the Extensions for User-Specified Types

In order to support user-specified composite types, the user needs to provide a partial template specialization of *scv_extensions* for the specific composite type in the test bench. This process can be automated through a simple Perl script, which takes a description like the following:

```
struct packet_t {
        sc_uint<8> addr;
        sc_uint<12> data;
};
```

and generates the appropriate partial template specialization:

```
SCV_EXTENSIONS(packet_t) {
```

```
public:
        scv_extensions< sc_uint<8> > addr;
        scv_extensions< sc_uint<12> > data;
        SCV_EXTENSIONS_CTOR(packet_t) {
                SCV_FIELD(addr);
                SCV_FIELD(data);
        }
};
```

It is easiest to create extensions for classes containing only public data. However, it is also possible to create an extension for a class containing private data. Consider this class declaration:

```
class restricted_t {
public:
        sc_uint<8> public_data;
private:
        sc_uint<8> private_data;
};
```

One can create an extension for this class by omitting the private data:

```
SCV_EXTENSIONS(restricted_t) {
public:
        scv_extensions< sc_uint<8> > public_data;
        SCV_EXTENSIONS_CTOR(restricted_t) {
                SCV_FIELD(public_data);
        }
};
```

One can create an extension that includes the private data by making the extension class a friend of the user type:

```
class restricted_t {
friend class scv_extensions<restricted_t>;
public:
        sc_uint<8> public_data;
private:
        sc_uint<8> private_data;
};

SCV_EXTENSIONS(restricted_t) {
public:
        scv_extensions< sc_uint<8> > public_data;
        scv_extensions< sc_uint<8> > private_data;
        SCV_EXTENSIONS_CTOR(restricted_t) {
                SCV_FIELD(public_data);
                SCV_FIELD(private_data);
        }
};
```

Similarly, an extension can be created on an enumeration:

```
enum instruction_t { ADD, SUB = 201 };
SCV_ENUM_EXTENSIONS(instruction_t) {
public:
        SCV_ENUM_CTOR(instruction_t) {
                SCV_ENUM(ADD);
                SCV_ENUM(SUB);
        }
};
```

Similar to the existing SystemC SC_MODULE macro, the SCV_EXTENSIONS and SCV_ENUM_EXTENSIONS macros create appropriate classes for the user-specified composite type and user-specified enumeration type. While the corresponding base classes are hidden from the users, for convenience, we refer to the conceptual base class as the *scv_extensions_base* template in the UML diagram described later in this section.

The related macros in the SystemC Verification Standard to facilitate this process are summarized in the following table:

The scv_extensions Macros	Description
SCV_EXTENSIONS(type_name)	This macro is similar to SC_MODULE(). It defines the extension class for the composite type identified by *type_name.*
SCV_EXTENSIONS_CTOR(type_name)	This macro is similar to SC_CTOR(). It defines the constructor for the extension class of the composite type identified by *type_name*.
SCV_EXTENSIONS_BASE_CLASS(base_type_name)	This macro declares *base_type_name* as the base class of the class to be extended. It must be instantiated within the block after SCV_EXTENSIONS_CTOR.
SCV_FIELD(field_name)	This macro declares a field identified by *field_name*.
SCV_ENUM_EXTENSIONS(type_name)	This macro is similar to SC_MODULE(). It defines the extension class for an enumeration identified by *type_name*.
SCV_ENUM_CTOR(type_name)	This macro is similar to SC_CTOR(). It defines the constructor for the extension class of the enumeration identified by *type_name*.
SCV_ENUM(enum_element_name)	This macro declares an enumeration element identified by *enum_element_name*.

4.3 Accessing the Static Extensions of Data Objects

The methods in the abstract data introspection interface can be partitioned into two categories: static extensions and dynamic extensions.

- **Static extensions**

  A static extension method within the abstract interface *scv_extensions_if* can be used to determine the specific type of an object, read from and write to the object, discover the record fields of the object, and discover the array elements of the object, without referring to the C++ header that defines the data type. The methods in the interface for *scv_extension_type_if* and *scv_extension_rw_if* are static extension methods.

- **Dynamic extensions**

  A dynamic extension method within the abstract interface *scv_extensions_if* adds and manipulates auxiliary data associated with a data object, which permits addition of instance-specific information to the object. This enables registration of callbacks, configuration for randomization, and building an expression tree from the data objects, without changing the C++ header that defines the data type.

Static extensions do not require auxiliary data to be associated with the data object, so they can be used for any data object without special use model. The functions to access the static extensions of any data object are captured in the following table.

Accessing Static Extensions	Description
template<typename T> scv_extensions<T> scv_get_extensions(T&);	This function returns an extension to a data object.
template<typename T> const scv_extensions<T> scv_get_const_extensions(const T&);	This function returns an extension to a constant data object.[1]

The returned object can be passed to any routine that takes a reference (or a pointer) of *scv_extensions_if* type as an argument. It is a run-time error if the dynamic extension methods of the returned object are used, because the returned object is a temporary wrapper around the data object. The auxiliary data stored in the extra storage will be lost when the temporary wrapper goes out of scope. Another call to *scv_get_extensions* will not be able to recover the same data. It might seem possible to store the auxiliary data in some global registry; however, it is difficult to maintain the registry for data objects that come and go, because two objects in different scopes can have the same pointer value when the memory is reused.

Static extensions can be used to extract information for value recording, as shown in the following code:

```
// The function my_code needs the C++ header for packet_t to instantiate p.
void my_code() { packet_t p; tool_A(p); }

// The utility tool_A can be designed for a generic type instead of the specific type packet_t
// This template needs the header for packet_t to compile.
template <typename T> void tool_A(const T& p) {
        scv_extensions<T> ext = scv_get_extensions(p);
        cout << "Argument p has " << ext.get_num_fields() << " fields." << endl;
```

[1] In order to overcome an ambiguity error in HP's aCC compiler, we have to use a different function name, *scv_get_const_extensions*, instead of overloading *scv_get_extensions* for obtaining an extension from a constant object.

```
        if (ext.is_integer()) ext.assign( rand() ); // basic assignment of values
        tool_B(ext);
};


// The implementation of a proprietary function can be hidden with the use of scv_extensions_if
// This function does not need the header for packet_t to compile.
void tool_B(scv_extensions_if& p) {
        if (p.is_integer()) cout << p.get_integer() << endl;
};
```

4.4 Accessing the Dynamic Extensions of Data Objects

In order to support dynamic extensions, we must be able to attach auxiliary data to a specific data object instance. In this section, we introduce two templates *scv_shared_ptr* and *scv_smart_ptr*. The *scv_shared_ptr* template enables sharing of data objects among multiple C++ threads by performing automatic memory management. The *scv_smart_ptr* template combines *scv_shared_ptr* and *scv_extensions* to attach auxiliary data to a data object. The *scv_smart_ptr* subsumes the functionality in *scv_shared_ptr*, but *scv_smart_ptr* is more costly in both memory space and performance, so it should be used only when dynamic extensions are needed.

Sharing Data Objects Among Multiple Threads

Typically, simple enhancements to arbitrary data types can be achieved through the use of templates. For example, the template *auto_ptr* from the standard C++ library is a smart pointer that points to a heap-based object. When this template goes out of scope, it also deletes the heap-based object.

In the SystemC multi-thread environment, where multiple threads can share the same heap object, *auto_ptr* is not very useful. The boost library from www.boost.org contains a template to enable reference-counted objects and is currently proposing to include this in the standard C++ library [10]. The SystemC Verification Standard includes a similar template, called *scv_shared_ptr*. The goal is to match this template with the one in the boost library, so that when the standard C++ library is extended to support shared pointers, it will be easy to re-package *scv_shared_ptr* to use the standard C++ library without asking SystemC users to change their code in any significant way. If possible, *scv_shared_ptr* will just be a typedef for *boost::shared_ptr,* and we will include the files directly from the boost library in the reference implementation of the Verification Standard.

The definition of the *scv_shared_ptr* template is shown in the following table. It uses reference counting to detect when the heap-based object should be deleted. This class is designed to behave as if it were a C/C++ pointer.

The scv_shared_ptr Template Class	Description
template< typename T> class scv_shared_ptr;	A reference-counted pointer for a heap-based object of type T.
scv_shared_ptr(T * core = 0);	A constructor to convert the argument to a shared pointer. A typical usage is: scv_shared_ptr p(new int());
scv_shared_ptr(const scv_shared_ptr&);	A copy constructor.
bool compare(const scv_shared_ptr& other) const;	This method returns true if this shared pointer points at the same object as the other pointer.
bool isNull() const	This method returns true if this shared pointer is null.

scv_shared_ptr& operator=(const scv_shared_ptr&);	The assignment operator.
bool operator!() const;	This operator returns true if the pointer is NULL. (see reference [5])
T& operator * () const;	This operator returns a reference to the heap-based object.
T * operator -> () const;	This operator returns a pointer to the heap-based object.
friend bool operator==(const scv_shared_ptr<T>&, const scv_shared_ptr<T>&);	This operator returns true if two shared pointers refer to the same heap-based object.
friend bool operator!=(const scv_shared_ptr<T>&, const scv_shared_ptr<T>&);	This operator returns true if two shared pointers refer to two different heap-based objects.

This class is especially useful with dynamic threads. For example, you can use a shared pointer in the following situation:

```
scv_shared_ptr<packet_t> p(new packet_t());
… // set up the packet…
sc_spawn_method(object,&object_t::method, p);
return;
```

In this code, a packet is created on the heap, and a new thread is spawned to process the object on the heap. Because memory management is performed automatically by *scv_shared_ptr*, users don't have to wait for the spawned task to finish before letting the stack variable *p* go out of scope. The packet will be deleted automatically when all related *scv_shared_ptr* instances go out of scope. For details, please refer to Meyers' discussion of smart pointers and reference counting [5].

This shared pointer concept forms the basis for adding extra storage space during data introspection on arbitrary data objects, as discussed below.

Attaching Auxiliary Data to Data Objects

In order to support dynamic extensions with auxiliary data, an enhancement of the *scv_shared_ptr* template is used. The enhanced template is called *scv_smart_ptr*, which combines the *scv_extensions* object and the data object using *scv_shared_ptr*. Instead of instantiating the data object directly, a smart pointer to a heap object is instantiated. This smart pointer also instantiates an extension object in the heap. Using the methods in the smart pointer and its abstract interface, the same auxiliary data associated with the data object can be accessed across simulation time and multiple threads and won't be lost.

The *scv_smart_ptr* template implements the *scv_smart_ptr_if* abstract interface (and the debugging interface discussed in Section 1). Their APIs are defined in the following tables:

The scv_smart_ptr_if Abstract Interface	Description
virtual scv_extensions_if * get_extensions_ptr() = 0;	This method returns a pointer to the associated extension of the heap-based data object.

virtual const scv_extensions_if * get_extensions_ptr() const = 0;	This method returns a constant pointer to the associated extension of the heap-based data object.

The scv_smart_ptr Template Class	Description
template<typename T> class scv_smart_ptr;	A smart pointer for data object of the type T.
scv_smart_ptr(T* heap_obj = 0, const char * name = 0);	This constructor creates a smart pointer with the supplied heap-based object. The heap-based object must be associated with a smart pointer only once using this constructor. Use the copy constructor to create additional smart pointers to the same heap-based object. If no heap-based object is supplied, a new one will be created using the default constructor. Note that this semantic is slightly different than *scv_shared_ptr*.
scv_smart_ptr(const scv_smart_ptr&);	The copy constructor.
scv_smart_ptr& operator = (const scv_smart_ptr&);	The assignment operator.
scv_extensions<T>& operator * () const;	This operator returns a reference to the associated extension for the data object.
scv_extensions<T> * operator -> () const;	This operator returns a pointer to the associated extension for the data object.
scv_expression operator () () const;	This operator creates a basic expression from the data object for future evaluation and manipulation.
const T& read() const;	This method returns the value of a data object.
viod write(const T&);	This method updates the value of a data object.

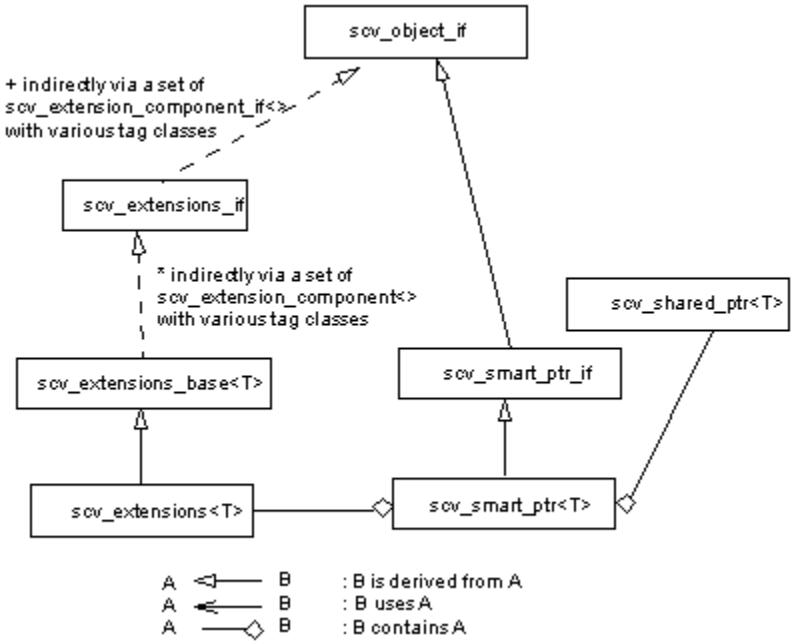
Conceptually, a smart pointer contains two shared pointers, one pointing to the data object and one pointing to the associated extension object. The *scv_smart_ptr* template is designed to behave as if it were a C/C++ pointer. The following example illustrates the use model:

```
void my_procedure (scv_smart_ptr<packet_t> p) {
        // The body is written as if this procedure is void my_procedure(packet_t *).
        cout << "my address : " << p->addr << endl;
        cout << "my data : " << p->data << endl;
        if (p->data == 0) p-> data = rand();
        cout << "the whole packet : " << *p << endl;
        p->data.set(3); // set() is a method on sc_uint.
};
```

A value change callback can be registered before *my_procedure()* is called:

```
scv_smart_ptr<packet_t> p;
p->register_cb(my_callback_function);
my_procedure(p);
```

The following UML-like diagram illustrates the relationship among various classes in the data introspection facility.



5 Randomization, Constraints, and Weight Specifications

Constrained random tests are an important element in a state-of-the-art verification environment. Using the data introspection facility discussed previously, constrained randomization can be performed on arbitrary data types. This section describes the corresponding APIs and the semantic for these features.

5.1 Seed and Random Stream Management

In order to support advanced seed management and enable independent random stream operations, we propose a *scv_random* class, as a replacement for *rand()* and *srand()* from the standard C library. The *scv_random* class uses an object-oriented paradigm to enable reproducibility in many use models.

An instantiation of the *scv_random* class gives the user an independent stream of random *unsigned* integer values. It can take an explicit seed from the user, or extract a seed from the seed associated with the current process thread. By default, it uses the same algorithm as *jrand48()* from the standard C library, but it can be configured to use *rand()* or a user-specified algorithm. The interface for the *scv_random* class is shown in the following table:

The scv_random Class (Global Configuration)	Description

enum value_generation_algorithm {
RAND,
RAND32,
RAND48,
CUSTOM
};
	This enumeration represents the different randomization algorithm:

typedef unsigned int (*alg_func)(unsigned long long& next);	The type for a custom randomization algorithm.
static void set_default_algorithm(value_generation_algorithm alg = RAND 48, alg_func custom_alg = 0);	This static method sets the base algorithm for new *scv_random* objects. The argument *custom_alg* is ignored unless the first argument is CUSTOM. If the first argument is CUSTOM and *custom_alg* is 0, RAND 48 is used.
static void set_global_seed(unsigned long long = 1);	This static method sets the global seed from which the seeds for new *scv_random* objects will be calculated based on the corresponding thread names.
static void print_initial_seeds(const char * filename);	This static method prints the initial seed information to the specified file.
static void print_initial_seeds(ostream&);	This static method prints the initial seed information to the specified stream.
static void print_current_seeds(const char * filename);	This static method prints the current seed information to the specified file.
static void print_current_seeds(osteram&);	This static method prints the current seed information to the specified stream.

static void seed_monitor_on(bool retrieve, const char * filename);	This static method turns seed monitor on. If retrieve is true, seed information is loaded into the registry and will be used to initialize subsequent scv_random objects by name matching. If retrieve is false, future seed information in stored in the specified file until seed_monitor_off() is called. If this method is executed twice without seed_monitor_off(), a warning is generated and the previous seed monitor will be turned off.
static void seed_monitor_on(bool retrieve, const char * monitorName, FILE * file);	This static method is the same as the previous seed_monitor_on() method, except that it stores the seed information in a user-created file pointer. Other code can use the same file to store other information. The monitor name is used to prefix any seed information, and the user must make sure that the other information in the file does not use the same prefix.
static void seed_monitor_off();	This static method turns off the seed monitor.

The *scv_random* class allows the user to specify which algorithm to use for random value generation. The semantic for the different mode are:

- RAND: Uses the re-entrant version of the C *rand()* algorithm, i.e. *rand_r()*. If the implementation of *rand_r()* in the C library of a platform generates only 16-bit integers, *scv_random::next()* will generate only 16 bit integers. Similarly randomization using data introspection will generate 16-bit integer for int and unsigned int. Randomization on other data types will give non-uniform probability distribution and some values may never be generated.
- RAND32: Uses the re-entrant version of the C *rand()* algorithm, i.e. *rand_r()*. If the implementation of *rand_r()* in the C library of a platform generates only 16-bit integers, scv_random will it twice to make sure every bit gets randomly set.
- RAND48: Uses the *jrand48()* algorithm for uniform unsigned random streams.
- CUSTOM: Uses a custom randomization algorithm specified through *set_default_algorithm()* for global configuration or *set_algorithm()* for specific random streams.

RAND is typically used for comparison or evaluation purposes, when a SystemC test bench needs to generate the same series of values as a C++ test bench using *rand()*. RAND32 is faster then RAND48, but *rand()* is know to generate values with a non-uniform distribution in many platforms.

The scv_random Class (per-instance)	**Description**
scv_random(const char * name = "<anonymous>");	This constructor creates a random stream with a specific name. A seed is generated from the global seed and the corresponding thread name.
scv_random(unsigned long long seed);	This constructor creates a random stream with an explicit seed.
scv_random(const char * name, unsigned lnog long seed);	This constructor creates a random stream with a specific name and an explicit seed.
scv_random(const scvg_random& other,	The copy constructor.

const char * name = "<anonymous>", unsigned lnog long seed = 0);	
unsigned int next();	This method generates the next random number.
unsigned long long get_initial_seed() const;	This method returns the initial seed from which this object was created.
void set_current_seed(unsigned long long) const;	This method sets the current seed from which the next random number will be generated
unsigned long long get_current_seed() const;	This method returns the current seed from which the next random number will be generated.
void set_algorithm(value_generation_algorithm = RAND48, alg_func algorithm = 0);	This method changes the algorithm from which future random numbers will be generated.

The *scv_random* class also contains the debugging interface discussed in Section 1. The following code shows the basic usage of *scv_random*:

```
scv_random gen("gen", 200);
cout << gen.next(); // print a random unsigned integer value
```

Users can explicitly control how many random streams they want in their testbenches by instantiating *scv_random* objects and associating them to *scv_smart_ptr* objects through the *set_random()* method in the data introspection interface (and other objects that support randomized behavior, such as *scv_bag*, described in Section 5.4)

The assignment of seeds to *scv_random* objects centers around reproducibility in several aspects.

Global Seed

The randomization facility has one global seed that the user can manipulate during elaboration time through the method *scv_random::set_global_seed()*.If the user does not provide the global seed explicitly, the default seed is 1.

Generating a Unique Seed for Each Process Thread

Because different SystemC implementations might have different scheduling orders among the C++ threads, this facility uses the hierarchical name of each process thread to transform the global seed to a unique seed for each thread. In doing so, provided that the standard defines a consistent hierarchical name convention for each process thread (and dynamic thread) across multiple SystemC implementations, the values generated from the randomization would be *independent* of the order in which the threads are executed.

One advantage of transforming the global seed into a unique seed per thread is that when a new process thread is added to an existing test bench (such as a new monitor module), the behavior of the existing thread will not change.

In order to support this use model, a new API is added to extract the hierarchical name of each process thread.

Supporting Function	Description
const char * scv_thread_unique_name(	This function returns the hierarchical name of the

const sc_thread_handle);	process thread.

Instantiating *scv_random* Objects

An instance of the *scv_random* class is automatically assigned a seed, based on the global seed. If it is a variable within a module, the hierarchical path of the module and its position compared to other *scv_random* objects within that module uniquely determine its seed. If it is a variable generated on-the-fly by a process thread, the hierarchical name of the thread and its position with the code of the thread also uniquely determine its seed.

As usual, instead of relying on the library to manage the seeds, users can override this assignment by explicitly setting the seed of the *scv_random* objects.

Seed File Manipulation

Using the thread-based assignment of seeds, similar behavior is maintained in the test bench even when threads are added or removed from your existing test bench. In the advanced cases in which the user wants to change the behavior of part of the test bench, while maintaining a similar behavior for the other part of the test bench, a seed file can be generated from a simulation run, and the user can manually change some of the seeds in the file, and rerun the simulation with the new seed file.

By composing the hierarchical name and the name that was specified for the *scv_random* object, the verification library can look up the initial seed that is used for each uniquely named *scv_random* object from the seed file. As a result, even when a new *scv_random* object is added to existing code within a thread, as long as this new *scv_random* object has a different name, the existing *scv_random* objects will be assigned the same seeds from the seed file so that the values generated in the original code will remain the same.

5.2 Basic Randomization

Data objects of arbitrary data types can be randomized through the use of *scv_smart_ptr*. For example, a random value for an *sc_uint<8>* can be generated using the following code:

```
scv_smart_ptr< sc_uint<8> > data;
data->next();
```

By default, *scv_smart_ptr* instantiates an internal *scv_random* object to perform randomization. The same *scv_random* object can also be shared among smart pointers by calling the method *set_random()*.

Similarly, it can randomize arrays and structs:

```
struct packet_t {
        int data;
        int array[10];
};
...
scv_smart_ptr< packet_t > p;
p->data.next(); // generate a random value and assign it to the data field
p->array[3].next(); // generate a random value and assign it to the array element with index 3
p->next(); // generate random values for all fields and all array elements
```

```
p->data.disable_randomization();
p->next(); // generate random values for all fields and all array elements, except for the data field.
```

These methods are defined in the data introspection interface for *scv_extension_rand_if*, in Section 4.1.

The randomization facility allows different modes of value generation. Section 1.1 talks about randomization using constraints, and Section 5.4 talks about randomization using weights and distributions. In summary, the randomization facility can be configured in several aspects:

- Randomization can be turned on and off using the methods *enable_randomization()* and *disable_randomization()*. The default is on.
- The modes of value generation are specified as the enumeration *scv_extensions_if::mode_t* in the data introspection facility.When randomization is on, the default value generation mode is RANDOM. and the user can change the mode by calling *set_mode()* with the desired mode as argument. If a bag is supplied as the argument to *set_mode(),* the DISTRIBUTION mode is set. If the methods *keeponly()* and *keep_out()* are called, the DISTRIBUTION mode is also set. For details about DISTRIBUTION mode and these methods, please see Section 5.4.
- The method *reset_distribution()* can be used to remove the existing distribution from a data object. If the mode before this call is DISTRIBUTION, it is changed to RANDOM.

Mode changes do not affect whether randomization is turned on or not. The only method that turns on randomization is *enable_randomization(),* and the only method that turns off randomization is *disable_randomization()*.

5.3 Constraint Specification and Constrained Randomization

Constraints are specified through derived classes of the *scv_constraint_base* class. An example is shown below:

```
class write_constraint : virtual public scv_constraint_base {
public:
        scv_smart_ptr< rw_task_if::write_t > write;
        SCV_CONSTRAINT_CTOR(write_constraint) {
                SCV_CONSTRAINT( write->addr() < 0x00FF );
                SCV_CONSTRAINT( write->addr() != write->data() );
        }
};
```

When constraints get complicated, it might be difficult to debug unsatisfiable constraints. During the SystemC Verification Working Group meetings, our conclusion is that the manual way of debugging is to “print the constraints and stare at the screen” to determine that they are unsatisfiable. EDA vendors can create tools to improve upon this, but the debugging solution does not need to be in the standard.

The standard does not require support of constraints on floats or doubles.

A constraint is derived from the *scv_constraint_base* class; the data to be randomized is specified as *scv_smart_ptr* class variable(s). The basic components of an expression can be created from *scv_smart_ptr* objects through *operator()()*, which can then be composed into more complicated expressions by using the following operators:

- Arithmetic operators +, -, *
- Relational operators ==, !=, >, >=, <, <=

- Logical operators !, &&, ||

In general, *operator()()* is used to create expressions that can be analyzed or evaluated at the later point. Both the scv_extensions classes and the scv_smart_ptr templates from the data introspection facility implement this operator.

Expressions are captured in the *scv_expression* class, with methods described in the following table:

The scv_expression Class	Description

enum operatorT {
EMPTY, EXTENSION, INT_CONSTANT,
UNSIGNED_CONSTANT, DOUBLE_CONSTANT,
SC_SIGNAL, EQUAL, NOT_EQUAL,
GREATER_THAN, LESS_THAN,
GREATER_OR_EQUAL, LESS_OR_EQUAL,
AND, OR, NOT, PLUS, MINUS, MULTIPLY
};
	This enumeration represents the possible types of expressions.

scv_expression(const scv_expression&);	The copy constructor.
scv_expression(int);	This constructor converts an integer to an expression.
scv_expression(unsigned);	This constructor converts an unsigned integer to an expression.
scv_expression(long long);	This constructor converts an long long to an expression.
scv_expression(unsigned long long);	This constructor converts an unsigned long long to an expression.
scv_expression(double);	This constructor converts a double to an expression.
template<typename T> static scv_expression create_reference(const sc_signal_in_if<T>& sig);	This virtual constructor converts an sc_signal object into an expression. When the expression is evaluation later in the simulation (or is analyzed by the constraint solver), the latest value at that point of the simulation will be used.
template<int W> static scv_expression create_constant(const sc_int<W>&);	This virtual constructor converts the value in a variable to an expression. The value at the time when the expression is constructed will be used.
template<int W> static scv_expression create_constant(const sc_uint<W>&);	This virtual constructor converts the value in a variable to an expression. The value at the time when the expression is constructed will be used.
template<int W> static scv_expression create_constant(const sc_bigint<W>&);	This virtual constructor converts the value in a variable to an expression. The value at the time when the expression is constructed will be used.
template<int W> static scv_expression create_constant(const sc_biguint<W>&);	This virtual constructor converts the value in a variable to an expression. The value at the time when the expression is constructed will be used.
template<int W> static scv_expression create_constant(const sc_bv<W>&);	This virtual constructor converts the value in a variable to an expression. The value at the time when the expression is constructed will be used.
friend scv_expression operator==(const scv_expression&, const scv_expression&);	This operator creates an expression with an equality comparison.

friend scv_expression operator!=(
  const scv_expression&,
  const scv_expression&
);
	This operator creates an expression with an inequality comparison.

friend scv_expression operator>(
  const scv_expression&,
  const scv_expression&
);
	This operator creates an expression with a greater than comparison.

friend scv_expression operator<(
  const scv_expression&,
  const scv_expression&
);
	This operator creates an expression with a less than comparison.

friend scv_expression operator>=(
  const scv_expression&,
  const scv_expression&
);
	This operator creates an expression with a greater-than-or-equal-to comparison.

friend scv_expression operator<=(
  const scv_expression&,
  const scv_expression&
);
	This operator creates an expression with a less-than-or-equal-to comparison.

friend scv_expression operator&&(
  const scv_expression&,
  const scv_expression&
);
	This operator creates a conjunction expression.

friend scv_expression operator||(
  const scv_expression&,
  const scv_expression&
);
	This operator creates a disjunction expression.

friend scv_expression operator!(const scv_expression&);	This operator creates a negation expression.

friend scv_expression operator+(
  const scv_expression&,
  const scv_expression&
);
	This operator creates an arithmetic addition expression.

friend scv_expression operator-(
  const scv_expression&,
  const scv_expression&
);
	This operator creates an arithmetic subtraction expression.

friend scv_expression operator*(
  const scv_expression&,
  const scv_expression&
);
	This operator creates an arithmetic multiplication expression.

bool evaluate() const;	This method evaluates the expression as a Boolean predicate.
const char * get_expression_string() const;	This method converts the expression into a string.
operatorT get_operator() const;	This method returns the top-level operator of the expression.

const scv_expression& get_left() const;	This method returns the left operand of the top-level operator, if applicable.
const scv_expression& get_right() const;	This method returns the right operand of the top-level operator, if applicable.
scv_extensions_if * get_extension() const;	This method returns the data introspection extension of the corresponding data object.
long long get_int_value() const;	This method returns the integer value of an integer expression.
unsigned long long get_unsigned_value() const;	This method returns the unsigned integer value of an unsigned integer expression.
double get_double_value() const;	This method returns the double value of an double expression.
void get_value(arg&) const;	A method to access constant values in an expression. This method is actually a series of overloaded *get_value()* methods, each taking an argument of a different type. The argument arg can be one of the following types: bool, char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, double, string, sc_string, sc_bv_base, and sc_lv_base.
sc_signal_in_if * get_signal() const;	This method returns the signal pointer for a signal expression.
void get_extension_list(list<scv_extensions_if*>&) const;	This method sets the argument to the list of data introspection extensions that were referenced in the expression.
void get_signal_list(list<sc_signal_in_if*>&) const;	This method sets the argument to the list of signal objects that were referenced in the expression.

Using *scv_expression*, constraints can be specified as derived classes of *scv_constraint_base*, in conjunction with several macros designed for this purpose. They are summarized in the following table:

The Constraint Macros	Description
SCV_CONSTRAINT_CTOR(constraint_name)	This macro is similar to SC_CTOR(), and it defines the constructor for the constraint identified by *constraint_name.*
SCV_CONSTRAINT(expression)	This macro defines a hard constraint.
SCV_SOFT_CONSTRAINT(expression)	This macro defines a soft constraint.
SCV_BASE_CONSTRAINT(base_constraint_name)	This macro defines a base constraint.

The scv_constraint_base Class	Description
void next();	This method generates a new random value to all members of the constraint class. Randomization of specific fields can be disabled or enabled by using disable_randomization() and enable_randomization() in the data introspection facility.
scv_extension_if::mode_t get_mode() const;	This method returns the current randomization mode.
scv_shared_ptr<scv_random> get_random() const;	This method returns the random stream that is

	attached to the constraint object.
void get_members(list<scv_smart_ptr_if *>&) const;	This method sets the argument to the list of members of the constraint object.
void set_mode(scv_extensions_if::mode_t);	This method sets the randomization mode for all members of the constraint object.
void set_random(scv_shared_ptr<scv_random>);	This method attaches a specific random stream to the constraint object.

The expressions in the series of *SCV_CONSTRAINT* macros within the constructor are merged into a single expression using the conjunction operator, &&, and stored in a static variable corresponding to the class. Using the traditional semantic for a conjunction, the resulting behavior is independent of the order from which the series of SCV_CONSTRAINT macros are declared. Because the constraint information is stored in a static variable for this class, it is processed only once. The expressions in the macros must use *operator()()* on the member smart pointers of the constraint class to relate the constraints to the underlying data objects referred by the member smart pointers. It is an error to assign a different data object to the smart pointer member variable of the constraint class. If a random value is needed for an existing smart pointer, you should call next() on the constraint and then copy the resulting value to the existing smart pointer.

Using this constraint class in randomization is straightforward. The following code generates random data for two writes and prints it to the screen:

```
write_constraint c("write constraint");
for (int i=0; i<2; ++i) {
        c.next();
        cout << *c.write << endl;
}
```

Selected fields can be randomized through the use of the *disable_randomization()* and *enable_randomization()* methods. For example, the following code generates a random value for the field *data* and uses sequential addresses:

```
write_constraint c("write constraint");
c.write->addr->disable_randomization();
for (int i=0; i<2; ++i) {
        c.write->addr = i;
        c.next();
        cout << *c.write << endl;
}
```

Variables can be dynamically enabled or disabled for randomization without limit.

Using *next()* is a good way to randomize multiple *scv_smart_ptr* objects and multiple fields in the same *scv_smart_ptr* object within the constraint. However, if only one field needs to be randomized, *next()* can be called directly for the specific field, which avoids using *disable_randomization()*. For example, the following code behaves the same way as the previous code:

```
write_constraint c("write constraint");
for (int i=0; i<2; ++i) {
        c.write->addr = i;
```

```
        c.write->data->next();
        cout << *c.write << endl;
}
```

The constraints specified by the macro *SCV_CONSTRAINT* are hard constraints; if the constraint solver cannot find a legal value that satisfies these hard constraints, an error is reported. Soft constraints can also be specified using the macro *SCV_SOFT_CONSTRAINT*. If the constraint solver cannot find a legal value that satisfies both soft and hard constraints, a warning is reported, and the solver will generate a value with respect to the hard constraints only, while ignoring the soft constraints. If the constraint solver still cannot find a legal value (ignoring the soft constraints), an error is reported, and the solver will generate a value while ignoring all constraints.

Because all constraints in a constraint class are ignored if a single (hard) constraint cannot be satisfied, it may be advisable to keep unrelated constraints in separate classes. Smaller constraint classes may make it easier to determine why a set of constraints cannot be satisfied.

It is not possible to disable part of a constraint expression; however, the same effect can be achieved by introducing a guard variable (with randomization disabled) to the relevant part of the expression. When the guard variable is set to false, the related part of the expression is effectively disabled.

The class-based constraint specification facility also supports generation of constraints using sequential code. For example, a constraint can be specified for each element of an array, as shown in the following code:

```
class complex_constraint : virtual public scv_constraint_base {
public:
        scv_smart_ptr< int > data_array[2];
        SCV_CONSTRAINT_CTOR(complex_constraint) {
                for (int i=0; i<2; ++i) {
                        SCV_CONSTRAINT(data_array[ i ]() < 10);
                }
        }
};
```

Because the constraints are captured in class declarations, class inheritance can be used to create a hierarchy of constraints. In the following example, the previous two constraints are merged into one with a new constraint specifying that the data in the write request cannot be the same as the first element in the data array from the other constraint.

```
class hierarchical_constraint : public write_constraint, public complex_constraint {
public:
        SCV_CONSTRAINT_CTOR (hierarchical_constraint) {
                SCV_BASE_CONSTRAINT(write_constraint);
                SCV_BASE_CONSTRAINT(complex_constraint);
                SCV_CONSTRAINT ( write->data() != data_array[ 0 ]() );
        }
};
```

While this constraint code focuses on manipulation of constraint objects, sometimes it is easier to write code with the data as the primary focus. The *use_constraint()* method applies a constraint to a data object.

```
extern void my_select_constraint( scv_smart_ptr<rw_task_if::write_t>& data);
void my_test() {
        scv_smart_ptr<rw_task_if::write_t> data;
        my_select_constraint(data);
        my_tvm.my_task(data);
}
void my_select_constraint(scv_smart_ptr<rw_task_if::write_t>& data) {
        if (...) {
                write_constraint c("c");
                data.use_constraint(c.write);
        } else {
                hierarchical_constraint c("c");
                for (int i=0; i<2; ++i) c.data_array[i]->next();
                data.use_constraint(c.write);
        }
}
```

Non-abstract methods in scv_extensions<T>	Description
void use_constraint(scv_smart_ptr<T>&);	This method assigns a constraint to a data object. The argument is expected to be a smart pointer within a constraint class. Otherwise, an error report is generated.

5.4 Weight Specification and Biased Randomization

While a constraint specifies the range of legal values, a weight specification biases the random value generation process so that some values are generated more often than others. This facility is captured in the enumeration *scv_extensions_if::mode_t*, representing the different modes from which values can be generated.

Distributions are specified using the concept of a bag, which represents either a collection of weighted values or a collection of weighted ranges (i.e., a bag of pairs). A bag is similar to a set, except that it can contain duplicated elements. For weighted ranges, we use the *pair* template from STL, which is a quick way to create a struct with two fields. The API for a bag is defined in the following table:

The scv_bag Class	Description
template<typename T> class scv_bag;	A bag with objects of type T.

scv_bag(
  const char * name = "<anonymous>",
  unsigned long long seed = 0
);
	This constructor creates an empty bag with the specified name and the specified seed for randomization. If *seed* is 0, a seed is generated from the global seed and the corresponding thread name.

scv_bag(
  const scv_bag& other,
  const char * name = "<anonymous>",
  unsigned long long seed = 0
);
	The copy constructor.

scv_bag& operator=(const scv_bag&);	The assignment operator.

void push(const T&, int num = 1);	This method adds the specified number of objects with the specified value to the bag.
void remove(const T&, bool all_copies = true);	This method removes one or all of the objects with the same value as the argument.
void clear();	This method empties the bag.
const T& peek_random();	This method selects a random object from the bag and returns its value.
int size() const;	This method returns the number of objects in the bag.
int distinct_size() const;	This method returns the number of distinct objects in the bag.
bool empty() const;	This method returns true if the bag is empty.
void set_random(scv_shared_ptr<scv_random>);	This method attaches the specified random stream to the bag.
friend bool operator==(const scv_bag&, const scv_bag&,);	Equality comparison.
friend bool operator==(const scv_bag&, const scv_bag&);	Inequality comparison.

For example, to specify a distribution in which the value "1" is generated 60% of the time and the value "2" is generated 40% of the time, the following code can be used:

```
scv_bag<int> bag;
bag.push(1,60);
bag.push(2,40);
scv_smart_ptr<int> data;
data->set_mode(bag);
for (int i=0; i<2; ++i) { data->next(); process(data); }
```

The set_mode() method allows the user to choose among four modes of value generation. For a data object with a non-composite type, a distribution can be supplied as the argument to select the DISTRIBUTION mode. When next() is executed for this data object, a random value is generated with respect to the supplied distribution, without invoking the constraint solver, i.e., any constraint expressed with this data object will be ignored in this process.

When next() is executed on a composite type or a constraint object with multiple smart pointers, some of the fields or smart pointers may be configured with the distribution mode, and the others may be configured in another mode. An example is attached as follows:

```
class c_t : public scv_constraint_base {
public:
    scv_smart_ptr<int> a;
    SCV_CONSTRAINT_CTOR(c_t) {
        SCV_CONSTRAINT( a() < 1 );
    }
};
```

```
void my_test() {
        c_t c("c");
        c.next(); // generate a value for “a” between INT_MIN and 0 inclusive

        scv_bag<int> b;
        b.add(10); b.add(11);
        c.a->set_mode(b);
        c.next(); // generate a value for “a” among { 10, 11 } with an error report about “a()<1” is made.
}
```

In this example, the constraint "a()<1" is checked after value generation from the distribution, and an error report is generated. The final value of *a* retains the value generated from the distribution. The user can change this semantic by overloading the *next()* method in their constraint classes.

When next() is executed on a composite type or a constraint with multiple smart pointers, the values are generated in two steps. The first step is to pick values for all member fields or member smart pointers with the distribution mode turned on. The values are picked from the distribution directly without consulting the related Boolean constraints. If there are other fields without the distribution mode turned on, a second step is taken to analyze the Boolean constraint and generate a constrained random value for them, using the values generated in the previous step, i.e., as if randomization has been turned off via disable_randomization() for those data objects in step 1. For example:

```
class c_t : public scv_constraint_base {
public:
        scv_smart_ptr<int> a;
        scv_smart_ptr<int> b;
        SCV_CONSTRAINT_CTOR(c_t) {
                SCV_CONSTRAINT( a() < b() && a() < 5 );
        }
};
void my_test() {
        c_t c("c");
        c.next(); // generate values for both a and b according to the Boolean constraint "a() < b() && a() < 5".

        scv_bag<int> dist;
        dist.add(2); dist.add(4);
        c.a->set_mode(dist);
        c.next();
                // step 1: generate a value for a among { 2, 4 } and the Boolean constraint "a() < b() && a() < 5" is ignored.
                // step 2: use the value generated in step 1 for “a”, and invoke the constraint solver to solve the
                //       Boolean constraint "a() < b() && a() < 5", and create a random value for “b”, while
                //       keeping the same value for “a”.
```

```
        scv_bag<int> dist2;
        dist2.add(2); dist2.add(11);
        c.a->set_mode(dist2);
        c.next();
                // step 1: generate a value for a among { 2, 11 } and the Boolean constraint "a() < b() && a() < 5"
                //      is ignored. Let's assume 11 is selected.
                // step 2: use the value generated in step 1 for a, and invoke the constraint solver to solve the
                //      Boolean constraint "a() < b() && a() < 5". In this case, since the value selected for “a”
                //      in step 1 violates the Boolean constraint, no legal value for “b” can be found to satisfy
                //      the Boolean constraint and an error report is made. A unconstrained value is selected
                //      for “b” in this case.
}
```

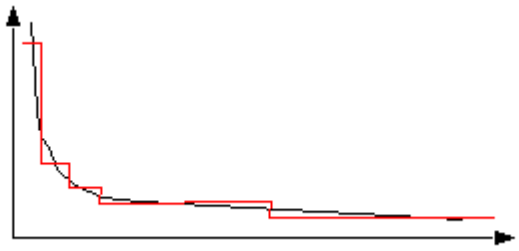
The weights on a range of values can be specified easily. For example, generating the range [0,1] 40% of the time and [2,10] 60% of the time can be achieved using the following code:

```
scv_smart_ptr<int> data;
scv_bag< pair< int,int> > distribution;
distribution.push( pair<int,int>(0,1), 40);
distribution.push( pair<int,int>(2,10), 60);
data->set_mode(distribution);
for (int i=0; i<3; i++) {data->next(); process(data); }
```

In this example, the *next()* method will select a range according to the weights, and then select a value from the range using a uniform probability distribution. As a result, while the chance of having some value within the range [2,10] is higher than that for some value within the range [0,1], the chance of having the value 10 is much smaller than that for the value 0. Effectively, the values 40 and 60 reflect the area under the range.

If the ranges within a distribution overlap each other, a warning report is generated when *set_mode()* is executed.

Using weights on ranges, a generic distribution, such as an exponential distribution, can be approximated by a step-like distribution, as shown in the following diagram:



Using this step-like distribution, a bag of pairs can be created and used to bias the randomization process:

```
scv_bag< pair<int,int> > bag;
bag.push(pair<int,int>(1,3), 100);
bag.push(pair<int,int>(4,10), 30);
bag.push(pair<int,int>(11,20), 20);
bag.push(pair<int,int>(21,80), 80);

...
scv_smart_ptr<int> data;
data->set_mode(bag);
for (int i=0; i<2; ++i) { data->next(); process(data); }
```

The *set_mode()* method is described in the following table:

Non-abstract randomization interface in scv_extensions<T>	Description
void set_mode(scv_extensions_if::mode_t);	This method sets the value generation mode. If the mode is DISTRIBUTION, the distribution from previous set_mode() is used. If none has been specified, an error report is generated, and the mode RANDOM is used.
void set_mode(const scv_bag<T>&);	This method sets the value generation mode to DISTRIBUTION, and uses the supplied bag with weights on individual values as the distribution to generate the value.
void set_mode(const scv_bag< pair<T,T> > &);	This method sets the value generation mode to DISTRIBUTION, and uses the supplied bag with weights on ranges as the distribution to generate the value.

The data introspection facility also includes several methods for specifying a simple distribution without a bag. They are listed in the following table. They are especially useful when the test bench wants to generate a value from a simple range.

Non-abstract randomization interface in scv_extensions<T>, where T is a non-composite type.	Description
void keep_only(const T&);	This method modifies the current distribution from other keep_only()s and keep_out()s to include only the supplied value.
void keep_only(const T& lowerbound, const T& upperbound);	This method modifies the current distribution from other keep_only()s and keep_out()s to include only the supplied range.
void keep_only(const list<T>&);	This method modifies the current distribution from other keep_only()s and keep_out()s to include only the supplied list of values.
void keep_out(const T&);	This method modifies the current distribution from other keep_only()s and keep_out()s to exclude the supplied value.
void keep_out(const T& lowerbound, const T& upperbound	This method modifies the current distribution from other keep_only()s and keep_out()s to exclude the supplied range.

);

void keep_out(const list<T>&);	This method modifies the current distribution from other keep_only()s and keep_out()s to exclude the supplied list of values.
void reset_distribution();	This method sets the attached distribution (from keep_only()s, keep_out()s, or set_mode()) to an empty bag, and sets the value generation mode to RANDOM

The set of keep_only and keep_out provides a convenient way to specify a distribution without creating an explicit bag. The calls to keep_only and keep_out are cumulative, and are combined using a simple conjunction semantics.

When these methods are executed, any previous distribution supplied by the set_mode() method is removed, and the value generation mode is automatically set to DISTRIBUTION, and constrained randomization will be turned off for this data object. If a new distribution is provided as a bag via set_mode() after these calls are executed, the new distribution will replace the distribution created through these keep_only and keep_out calls.

Some examples are included as follows:

```
scv_smart_ptr<int> i;
i->keep_only(0,4);
i->keep_out(2);
i->next(); // generate a value among { 0, 1, 3, 4 }

scv_bag<int> b;
b.add(2); b.add(7);
i->set_mode(b);
i->next(); // generate a value among {2,7}

i->reset_distribution();
i->next(); // generate a value between INT_MIN and INT_MAX.
```

6 Variable and Transaction Recording

In order to effectively debug a simulation run, visualization of events and activities is very important. A test bench must be able to record appropriate data into a database to support visualization. The same database can also be used to perform coverage analysis. This section describes the API from which the user can control what information is recorded in the database.

Two kinds of recordings have been considered. Because value transitions in variables can be recorded using value-change callbacks, only the callback registration API in the data introspection facility is included in the SystemC Verification Standard. The focus of the Verification Working Group discussion was on transaction recording, which we felt is the suitable level of abstraction for recording activities in a test bench.

6.1 Variable Recording

The values of a variable across time can be recorded into a database using the VCD facility in SystemC 2.0. However, it is more efficient to associate variable recording to value-change callbacks. In SystemC 2.0, a value-change event on an *sc_signal* object can be performed using code similar to the following example:

```
class my_module : public sc_module {
public:
        sc_inout< bool > sig1;
        SC_CTOR(my_module) {
                SC_METHOD(sig1_callback);
                sensitive << sig1;
        }
        void sig1_callback() { cout << "The value of sig1 has been changed to : " << sig1 << endl; }
};
```

Using the data introspection facility described earlier, value-change callbacks can also be performed on a data object by using *scv_smart_ptr*.

```
class my_module : public sc_module {
public:
        scv_smart_ptr<int> fsm_state;
        SC_CTOR(my_module) {
                fsm_state->register_cb(fsm_state_callback);
        }
        void fsm_state_callback(scv_extensions_if& data, scv_extensions_if::callback_reason r) {
                if ( r == scv_extensions_if::VALUE_CHANGE) {
                        cout << "The FSM state has been changed to : " << data << endl;
                        write_to_the_database(data);
                }
        }
};
```

Using value-change callbacks in a data object is different from using value-change callbacks in a signal. Using data introspection, value-change callbacks can be registered in a data object of any type, and the callbacks will be executed whenever an assignment to the data object is performed, regardless of whether the new value is the same as the old value or not. A value-change callback in a signal is executed after the events in a delta cycle are processed. So, if a signal has value 1, and a process assigns a value 0 and then a value 1 to the signal within the same delta cycle, the callback will not be executed. If a data object with dynamic extensions has value 1, and a process assigns a value 0 and then a value 1 to the data object within the same delta cycle, the callback will be executed twice. If an action should be taken only when a different value is assigned to the data object, the callback function could be written to store the previous value and to ignore invocations when the new value is the same as the previous value.

6.2 Transaction Recording

Simulation activity in a testbench is best recorded at the transaction level. Transaction recording is the act of recording timing information and attribute information associated with transactions into the database. This information can be used to visualize simulation activities, debug, perform coverage analysis, and do other tasks. Similar to the way synchronization (such as sc_semaphore) is organized in SystemC 2.0, the SystemC Verification Standard uses a two-layer approach:

- Manual transaction recording: One set of core APIs in the standard with *sufficient expressiveness* for various styles of transaction recording.
- Automatic/assisted transaction recording: Multiple sets of convenience APIs, built on top of manual transaction recording, each *simplifying the use model* in a specific style of transaction recording.

The current specification contains the manual transaction recording API. We are still discussing various ways of doing automatic or assisted transaction recording. The SystemC Verification Standard probably needs to include several specific styles that are commonly found in existing application domains, but the important part is to make sure that manual transaction recording is expressive enough to support various non-standard transaction recording styles.

Several styles of performing automatic or assisted transaction recording are summarized in the appendix. Because we have not reached a conclusion, the information in the appendix is provided for reference and archive, and to stimulate new ideas. The appendix should not be considered as part of the SystemC Verification Standard.

6.2.1 The Architecture

The manual transaction recording API contains three major classes, scv_tr_db, scv_tr_stream, and scv_tr_generator. These classes are independent of the actual database format that is used in the simulation. Callback registrations are used to connect a specific database to the transaction recording facility. The main reason for using callbacks is to address the following requirements:

- Support multiple databases in a single simulation.
- Enable proprietary databases to connect to any SystemC implementation.
- Be generic enough to support multiple styles and degrees of automation.
- Avoid unnecessary processing at simulation run-time.

Partitioning the facility into three classes provides a flexible architecture to support different styles of transaction recording. The responsibilities of the three classes are summarized in the following table:

Class	Description	Analogy	Strategy
scv_tr_db	A transaction database containing a collection of transaction streams.	A directory containing multiple files.	Multiple instances represent multiple databases in a single simulation.
scv_tr_stream	A transaction stream containing a collection of related transactions.	A file containing a collection of records.	Each module or channel may use zero, one, or multiple streams to group the transactions that are being generated.
scv_tr_generator	A transaction generator for a specific transaction type, containing information such as the transaction type names, and attribute names.	A form for entering records of a specific record type, asking the user for information about individual fields.	Compile-time type checking, optimization, and preprocessing can be performed with minimal overhead in the creation of individual transactions.

The *scv_tr_db* objects are typically (but not necessarily) instantiated within *sc_main*. These objects let users open and close a transaction recording database, and suspend and resume transaction recording.

After instantiating the database objects, transaction streams and their associated transaction generators can be created for each database, typically (but not necessarily) at elaboration time in the simulation, and typically as member variables of a module or a channel.

```
class rw_pipelined_transactor : public pipelined_bus_ports, public rw_task_if {
public:
        scv_tr_stream rw_stream;
        scv_tr_generator< addr_t, data_t> read_gen;
        scv_tr_generator< addr_t, data_t> write_gen;
        SC_CTOR(rw_pipelined_transactor) :
                rw_stream("my_transactor"),
                read_gen("read",rw_stream,"addr","data"),
                write_gen("write",rw_stream,"addr","data") { ... }
        …
```

```
};
```

The *scv_tr_stream* objects are the primary means to group related transactions into the same area within your database; a stream can have overlapping transactions. You can either create a transaction stream directly as a class variable in your transactor, or pass the pointer of a transaction stream to a transactor during the elaboration phase.

The *scv_tr_stream* constructor takes a string as the first argument, which will be used as the name for this stream. The second argument is a string identifying the kind of stream to be created. You can use this argument to tag the stream as a stream in a transactor, in a test, or in other scenarios. The final argument is the database from which this stream will be created. If no database is provided, the default database will be obtained from the global database configured via the static method *set_default_db()*.

After instantiating the transaction streams, information about the kind of transactions in a stream can be specified by using the *scv_tr_generator* template class. The template takes two optional template arguments; the first argument is the type of attribute that (if specified) must be provided when a transaction is first initiated; and the second argument is the type of attribute that (if specified) must be provided when a transaction is terminated. Apart from the attribute types specified in the template arguments, special attributes can be added to the transaction through the transaction handle, although it is typically slower to do so.

The arguments to the constructor of the template are the type names of the transactions, the transaction stream to be recorded on, and the optional string names of the two optional attributes. If you create two *scv_tr_generator* objects with exactly the same template parameters and the same constructor arguments, the two objects refer to the same underlying core to generate the same kinds of transactions.

Once the generators are instantiated, a transaction can be created by calling appropriate methods in a generator. The actual API to create transactions is described in a later section. Because string names are processed during the construction of the generator, the library does not have to process the names again when individual transactions are created. The types of the begin attributes and the end attributes are provided through template parameters, enabling performance optimization with respect to the attributes that use template specialization.

The actual code to record the information into a specific database can be connected to this facility by registering a series of callbacks. Using this callback mechanism, a text-based database is provided with the reference implementation. Individual tool vendors can configure the reference implementation and their proprietary implementation to record to a different database just by changing or adding related callbacks.

This architecture satisfies the requirements that were listed earlier in this section:

- Multiple databases can be created by instantiating multiple *scv_tr_db* objects
- These four classes are independent of the actual database format that is used in the simulation. Connection to a specific database format (open or proprietary) can be established through the use of callback registrations, using static methods such as *register_class_cb()*.
- This architecture provides a flexible use model and supports different styles and different degree of automation. For example:
  - One or more transaction streams (*scv_tr_stream*) can be instantiated as class variables within the same module or channel. They can also be shared among multiple modules or channels;
  - One or more transaction generators (*scv_tr_generator*) can be instantiated as class variables within the same module or channel. They can also be instantiated on-the-fly, right before a transaction is recorded. The former style is efficient, but the latter style allows more automation.
  - A transaction stream (*scv_tr_stream*) can have more than one associated transaction generator (*scv_tr_generator*) with different transaction types and different attribute types. It also supports the use model with specialized transaction streams that contain only one specific transaction type with fixed attribute types.

- The use of *scv_tr_generator* enables transactions to be generated efficiently without manipulating transaction type strings, attribute names, and other common information about the transaction types. The use of template parameters to specify attribute types in *scv_tr_generator* enables further optimization through partial template specialization.

The following tables describe the API for the *scv_tr_db* class and *scv_tr_stream* class. The callback registration methods are the main API for these classes. While similar callback registration methods are included in the generator class, the generator class also includes the methods to generate and manipulate transactions, so the API table for the generator class will be presented later in the section instead of here.

While EDA vendors can use these callbacks to connect the SystemC test bench to their proprietary database, the SystemC user only need to learn about how to construct these objects. These classes also contain the debugging interface discussed in Section 1.

These classes utilize a typedef defined in global scope:

```
typedef long scv_tr_relation_handle_t;
```

The scv_tr_db Class	Description
scv_tr_db(const char * db_name, sc_time_uint = SC_FS);	This constructor creates a database with the specified name and the specified time scale.
static void set_default_db(scv_tr_db *);	This static method sets the default database for subsequent stream creations.
static scv_tr_db * get_default_ db();	This static method returns the database configured via *set_default_db()*.
void set_recording(bool);	This method turns recording on and off (the default is on).
bool get_recording() const;	This method returns true if recording is turned on for this database.

enum callback_reason {
CREATE,
DELETE,
SUSPEND,
RESUME
};
	This enumeration represents the situations in which callbacks are executed.
	CREATE: When a database is created.
	DELETE: When a database is deleted.
	SUSPEND: When the recording to a database is suspended.
	RESUME: When the recording to a database is resumed.

typedef int callback_h;	The *scv_tr_db* class has an associated type for the callback handle. The reference implementation will implement it as an integer. This handle can be used to remove a callback after it has been registered.
typedef void callback_function(scv_tr_db&, callback_reason, void * user_data);	This type defines the callback function that can be registered.
scv_tr_relation_handle_t create_relation(const char *relation_name) const;	Create a new relation that can be established between two transactions. If a relation with the specified name has already been created, return the handle to that relation.

const char *get_relation_name(scv_tr_relation_handle_t relation_handle) const;	Get the name of the relation with the specified handle. Return NULL if the handle does not refer to a valid relation.

static callback_h register_class_cb(
  callback_function *,
  void * user_data = 0
);
	This method registers a callback to be executed whenever one of the situations represented by *callback_reason* happens. The callback function should examine the *callback_reason* that is provided through the argument and take appropriate action.

static void remove_class_cb(callback_h);	This method removes a callback from the database.

The scv_tr_stream Class	**Description**

scv_tr_stream(
  const char * stream_name,
  const char * stream_kind,
  scv_tr_db * database
  = scv_tr_db::get_default_db()
);
	This constructor creates a transaction stream with the specified name in the specified database. The string for *stream_kind* describes what kind of stream it is. For example, you can use “transactor” for streams within any transactor.

const char * get_stream_kind() const;	This method returns the stream kind string supplied in the argument of the constructor.
scv_tr_db * get_tr_db() const;	This method returns the database associated with this stream.

enum callback_reason {
  CREATE,
  DELETE
};
	This enumeration represents the situations in which the callbacks are executed.
	CREATE: When a stream is created.
	DELETE: When a stream is deleted.

typedef int callback_h;	The *scv_tr_stream* class has an associated type for the callback handle. The reference implementation will implement it as an integer. This handle can be used to remove a callback after it has been registered.

typedef void callback_function(
  scv_tr_stream&,
  callback_reason,
  void * user_data
);
	This type defines the callback function that can be registered.

static callback_h register_class_cb(
  callback_function *,
  void * user_data = 0
);
	This method registers a callback to be executed whenever one of the situations represented by *callback_reason* happens. The callback function should examine the *callback_reason* that is provided through the argument and take appropriate action.

static void remove_class_cb(callback_h);	This method removes a callback from the stream.

6.2.2 Generating Transactions

The design of the transaction generator is tightly related to the definition of transactions:

- A transaction is a collection of attribute values that are stored as a group in a transaction recording database.
- Each transaction has a specific type (identified by a string), a specific start time, and a specific end time,

- The attributes are classified as begin attributes, end attributes, and special attributes. These attributes are all optional. The value of the begin attribute needs to be specified when you create a transaction, and the value of the end attribute needs to be specified when you terminate a transaction. Special attributes can be added as necessary between the begin time and the end time.
- A transaction can be related to another transaction. A relationship can be specified using a string identifier, explicitly provided by users.
- Transactions can have a collection of built-in attributes that are automatically set by the implementation. These include *begin_time*, *end_time*, and others. From the user point of view, a test bench does not have to know that these attributes exist.

After considering various scenarios that may need transaction recording, we have identified the following requirements for an API for the transaction generators:

- Able to record transactions at the exact time it starts and finishes (e.g. in a stimulus generator).
- Able to record transactions that started and/or finished in the past (e.g. in a monitor).
- Support overlapping transactions (for pipelined protocols and split protocols)

Recording Basic Transactions

Basic transactions can be created by the methods begin_transaction() and end_transaction() in the generator. For example, the following code create a read transaction for each execution of the *read()* method.

```
data_t rw_pipelined_transactor::read(const addr_t * addr) {
        scv_tr_handle h = read_gen.begin_transaction(*addr);
        …
        if (special_case) h.record_attribute("special attribute", special_attribute);
        read_gen.end_transaction(h, data);
        return data;
    }
```

This basic usage of the three methods in the generator creates a transaction that begins at the time when *begin_transaction()* is executed, and ends at the time when *end_transaction()* is executed. The separation of the begin attribute and end attribute fits nicely into this method with one argument and one return value. Concurrent execution of the *read()* method will generate overlapping read transactions, and users can add appropriate synchronization among concurrent threads to control how much overlapping is allowed in their test benches.

Because there might be multiple outstanding transactions at a given time, a transaction handle is used to match the calls to *begin_transaction()* and *end_transaction()* so that appropriate transactions can be terminated. After a transaction is terminated, the handle is still valid, allowing it to be used, for example, to specify relationships among transactions, as described later in this section. The run-time information about the transaction is deleted when all handles to the specific transaction go out of scope.

Through the template parameters of the generator, compile-time type-checking can be performed on these methods to make sure the user has passed in begin and end attributes with the correct types. Recording these attributes when the library creates or terminates a transaction allows more optimization than using the special attributes to implement the begin and end attributes.

When additional attributes are recorded using *record_attribute(),* an explicit name must be provided. Each attribute of a transaction is considered to have one valid value. This is the last value recorded for the attribute for a given transaction. If multiple record_attribute calls are made with the same attribute name on a transaction,

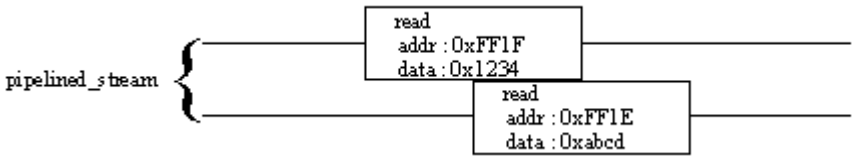
this is equivalent to a glitch. The last value is considered the good value. It is up to the scv_tr database writer implementation whether to store the glitch values or not. However, the last value must be stored for any implementation[1].

All applicable attribute types must have a data introspection extension so that SystemC can extract the right values from the variables and store them into the database. During recording, pointer fields in a composite type are not traversed, so that the size of the attribute is constant across multiple transactions of the same transaction type. In the future, if there is a demand, we may decide to add in the capability to traverse pointers, probably enabled through an optional argument to the generator.

Taking the example in Section 3, we can illustrate how transactions can be recorded in a design with a pipelined interface. A transactor for a pipelined bus with at most two outstanding transactions can be modeled as follows:

```
class rw_pipelined_transactor : public pipelined_bus_ports, public rw_task_if {
public:
        fifo_mutex address_phase;
        fifo_mutex data_phase;
        scv_tr_stream pipelined_stream;
        scv_tr_generator< addr_t, data_t > read_gen;
        scv_tr_generator< addr_t, data_t > write_gen;
        ...
        virtual data_t read(addr_t * addr) {
                address_phase.lock();
                scv_tr_handle h = read_gen.begin_transaction(*addr);
                ...// Address phase
                addr_phase.unlock();
                data_phase.lock();
                ...// Data phase
                read_gen.end_transaction(h, data);
                data_phase.unlock();
                return data;
        }
};
```

If there are two concurrent threads calling *read()* at the same time, the following transactions will be generated in the database.



[1] According to the definition of a transaction, each attribute has only one value. If we store all specified values, together with the exact time such values are recorded, we will incur the full overhead of signal recording, and change our definition of a transaction.

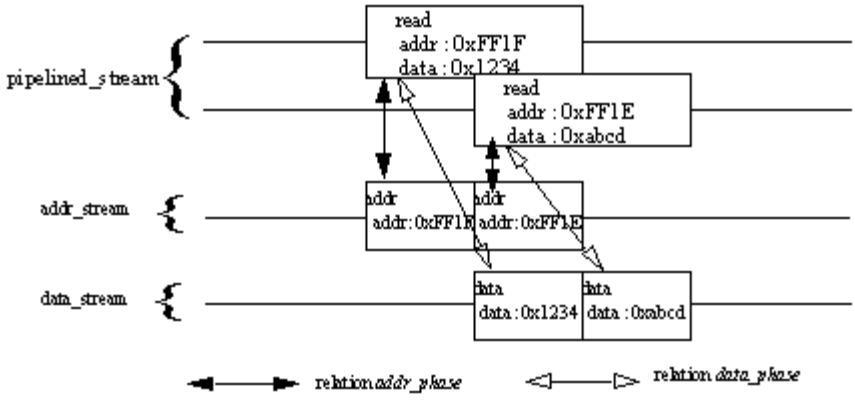
Advanced Transaction Recording

In order to support monitor-style transaction recording, variants of the *begin_transaction()* and *end_transaction()* methods are included for specifying transactions that started and terminated in the past. This variant takes an extra argument to specify the exact time at which the transaction has begun or ended. The value of this extra argument must indicate a time in the past so that the database can be implemented efficiently. It is a run-time error if the supplied time argument is in the future.

The ability to specify transaction relationships is important to facilitate debugging in terms of transactions. It enables recording of causal relationships among the activities in the simulation. For example, the following example creates a transaction for each phase of the pipeline. A transaction relationship is a natural way to link the related transactions together.

```
class rw_pipelined_transactor : public pipelined_bus_ports, public rw_task_if {
public:
        …
        scv_tr_stream addr_stream;
        scv_tr_stream data_stream;
        scv_tr_generator< addr_t > addr_gen;
        scv_tr_generator< data_t > data_gen;
        SC_CTOR(my_transactor)
                : ...
                addr_gen("addr",addr_stream,"addr"),
                data_gen("data",data_stream, "data") { ... }

        virtual data_t read(addr_t * addr) {
                address_phase.lock();
                scv_tr_handle h = read_gen.begin_transaction(*addr);
                scv_tr_handle h1 = addr_gen.begin_transaction(*addr, “addr_phase”, h);
                ...// Address phase
                addr_gen.end_transaction(h1);
                addr_phase.unlock();
                data_phase.lock();
                scv_tr_handle h2 = data_gen.begin_transaction(“data_phase”,h);
                ...// Data phase
                data_gen.end(h2,data);
                read_gen.end_transaction(h, data);
                data_phase.unlock();
                return data;
            }
};
```

This code will generate the following transactions in the database:



The following tables describe the API for the *scv_tr_generator* template class (and its base class *scv_tr_generator_base*) and *scv_tr_handle* class. While EDA vendors can use these callbacks to connect the SystemC test bench to their proprietary database, the SystemC user only needs to learn about how to create transactions. These classes also contain the debugging interface discussed in Section 1.

The scv_tr_generator_base Class	Description
scv_tr_stream * get_tr_stream() const;	This method returns the transaction stream associated with this generator.
const char * get_begin_attribute_name() const;	This method returns the name of the begin attributes.
const char * get_end_attribute_name() const;	This method returns the name of the end attributes.

enum callback_reason {
CREATE,
DELETE
};
	This enumeration represents the situations in which the callbacks are executed.
	CREATE: When a generator is created.
	DELETE: When a generator is deleted.

typedef int callback_h;	The *scv_tr_generator_base* class has an associated type for the callback handle. The reference implementation will implement it as an integer. This handle can be used to remove a callback after it has been registered.
typedef void callback_function(scv_tr_generator&, callback_reason, void * user_data);	This type defines the callback function that can be registered.
static callback_h register_class_cb(callback_function *, void * user_data = 0);	This method registers a callback to be executed whenever one of the situations represented by *callback_reason* happens. The callback function should examine the *callback_reason* that is provided through the argument and take appropriate action.
static void remove_class_cb(callback_h);	This method removes a callback from the generator.

The scv_tr_generator Template Class	Description

template<typename begin_type, typename end_type> class scv_tr_generator;	A transaction generator is a template with two optional parameters, representing the types of the begin attribute and the end attribute.
scv_tr_generator(const char * transaction_type_name, scv_tr_stream&, const char * begin_attribute_name = 0, const char * end_attribute_name = 0);	This constructor creates a generator with the specified name and stream. The names of the begin attribute and the end attribute can be supplied as optional arguments.
scv_tr_handle begin_transaction();	This method creates a new transaction without a begin attribute, starting at the current time.
scv_tr_handle begin_transaction(const begin_type&);	This method creates a new transaction with the specified begin attribute, starting at the current time.
scv_tr_handle begin_transaction(const sc_time& begin_time);	This method creates a new transaction without a begin attribute, starting in the past as specified in the argument.
scv_tr_handle begin_transaction(const begin_type&, const sc_time& begin_time);	This method creates a new transaction with the specified begin attribute, starting in the past as specified in the argument.
scv_tr_handle begin_transaction(const char * relation_name, const scv_tr_handle&);	This method creates a new transaction without a begin attribute, starting at the current time. The two arguments specify a transaction relationship with another existing transaction. By providing this information in the same call as *begin_transaction()* (instead of a separate call to *add_relation()*), more optimization can be achieved.
scv_tr_handle begin_transaction(scv_tr_relation_handle_t relation_handle, const scv_tr_handle&);	This method creates a new transaction without a begin attribute, starting at the current time. The two arguments specify a transaction relationship with another existing transaction. By providing this information in the same call as *begin_transaction()* (instead of a separate call to *add_relation()*), more optimization can be achieved. Identifying the relation by handle rather than by name is still more efficient.
scv_tr_handle begin_transaction(const begin_type&, const char * relation_name, const scv_tr_handle&);	This method creates a new transaction with the specified begin attribute, starting at the current time. The last two arguments specify a transaction relationship with another existing transaction.
scv_tr_handle begin_transaction(const begin_type&, scv_tr_relation_handle_t relation_handle, const scv_tr_handle&);	This method creates a new transaction with the specified begin attribute, starting at the current time. The last two arguments specify a transaction relationship with another existing transaction.

scv_tr_handle begin_transaction(const sc_time&, const char * relation_name, const scv_tr_handle&);	This method creates a new transaction without a begin attribute, starting in the past as specified in the argument. The last two arguments specify a transaction relationship with another existing transaction.
scv_tr_handle begin_transaction(const sc_time&, scv_tr_relation_handle_t relation_handle, const scv_tr_handle&);	This method creates a new transaction without a begin attribute, starting in the past as specified in the argument. The last two arguments specify a transaction relationship with another existing transaction
scv_tr_handle begin_transaction(const begin_type&, const sc_time&, const char * relation_name, const scv_tr_handle&);	This method creates a new transaction with the specified begin attribute, starting in the past as specified in the argument. The last two arguments specify a transaction relationship with another existing transaction.
scv_tr_handle begin_transaction(const begin_type&, const sc_time&, scv_tr_relation_handle_t relation_handle, const scv_tr_handle&);	This method creates a new transaction with the specified begin attribute, starting in the past as specified in the argument. The last two arguments specify a transaction relationship with another existing transaction.
void end_transaction(const scv_tr_handle&);	This method terminates a transaction without an end attribute, ending at the current time.
void end_transaction(const scv_tr_handle&, const end_type&);	This method terminates a transaction with the specified end attribute, ending at the current time.
void end_transaction(const scv_tr_handle&, const sc_time&);	This method terminates a transaction without an end attribute, ending in the past as specified in the argument.
void end_transaction(const scv_tr_handle&, const end_type&, const sc_time&);	This method terminates a transaction with the specified end attribute, ending in the past as specified in the argument.

The scv_tr_handle Class	Description
scv_tr_handle();	The default constructor to create a place-holder for a transaction.
scv_tr_handle(const scv_tr_handle&);	The copy constructor.
scv_tr_handle& operator=(const scv_tr_handle&);	The assignment operator.
scv_tr_stream * get_tr_stream() const;	This method returns the stream associated with this transaction.
scv_tr_generator_base * get_tr_generator() const;	This method returns the generator associated with this

	transaction.
bool is_valid() const;	This method returns true if this handle refers to a valid transaction.
bool is_active() const;	This method returns true if this handle refers to a valid transaction that has not been terminated.
long long get_id() const;	This method returns a integer id for this transaction that is unique throughout the entire simulation
template<typename T> void record_attribute(const char * attribute_name, const T& attribute_value);	This method records a special attribute.
template<typename T> void record_attribute(const T& attribute_value);	This method records a special attribute. It can be used for attribute_value types which have scv_extensions that include the name of the object.
void add_relation(const char * relation_name, const scv_tr_handle&);	This method specifies a transaction relationship between the current transaction and the specified transaction.
void add_relation(scv_tr_relation_handle_t relation_handle, const scv_tr_handle& transaction_handle);	This method specifies a transaction relationship between the current transaction and the specified transaction
scv_extensions_if * get_begin_exts() const;	This method returns the extension of the begin attribute.
scv_extensions_if * get_end_exts() const;	This method returns the extension of the end attribute.
const scv_tr_handle *get_immediate_related_transaction(scv_tr_relation_handle_t * relation_handle_p);	If a related transaction is specified at the beginning of a new transaction, then this method returns the other related transaction and the relation_handle. Else it returns NULL.

enum callback_reason {
BEGIN,
END,
DELETE
};
	This enumeration represents the situations in which the callbacks are executed.
	BEGIN: When a transaction is created.
	END: When a transaction is terminated.
	DELETE: When all related handles for a specific transaction have gone out of scope.

typedef int callback_h;	This class has an associated type for the callback handle. The reference implementation will implement it as an integer. This handle can be used to remove a callback after it has been registered.
typedef void callback_function(scv_tr_handle&, callback_reason, void * user_data);	This type defines the callback function that can be registered.
static callback_h register_class_cb(callback_function *, void * user_data = 0);	This method registers a callback to be executed whenever one of the situations represented by *callback_reason* happens. The callback function should examine the *callback_reason* that is provided through the argument and take appropriate action.
typedef void callback_special_attribute_function(	This type defines the callback function that can be registered for the special attributes.

scv_tr_handle&, const char * attribute_name, scv_extensions_if * attribute_value, void * user_data);	
static callback_h register_special_attribute_cb(callback_special_attribute_function *, void * user_data = 0);	This method registers a callback to be executed whenever a special attribute is specified for a specific transaction.
typedef void callback_relation_function(scv_tr_handle&, scv_tr_handle&, void * user_data, scv_tr_relation_handle_t relation_handle);	This type defines the callback function that can be registered for a transaction relationship.
static callback_h register_relation_cb(callback_relation_function *, void * user_data = 0);	This method registers a callback to be executed whenever a transaction relationship is specified on two specific transactions.
static void remove_class_cb(callback_h);	This method removes a callback from the list of callbacks registered through the static methods for scv_tr_handle.

7 Miscellaneous Supporting Facilities

There are several smaller facilities that we have found useful for supporting test bench creation, verification IP creation, and debugging. They are described in this section.

7.1 HDL Connection

The SystemC Verification Standard enables the use of SystemC to create testbenches. We would like such testbenches to be able to simulate both SystemC designs and HDL designs. As a result, HDL connection in SystemC is an important prerequisite to the Verification Standard. Because it is not currently in the SystemC standard, the SystemC Verification Standard contains a basic API to enable such use models. The intent of this API is to not impose any requirement that cannot be met through standard interfaces for connection to an HDL simulator, such as PLI, VPI, and VHPI.

An elaborated API for HDL connection is out of the scope of the current SystemC Verification Standard specification. Specifically, the current specification does not handle connection to a bit-slice of an HDL signal, a Verilog memory, or a Verilog memory element. It does not allow forcing an HDL resolved signal to a specific value or releasing it from the SystemC code. This specification does not cover the ability of directly instantiating HDL modules in SystemC or vice versa. It merely assumes that the SystemC design hierarchy and the HDL design hierarchy are in the same simulation executable, but neither is directly aware of the other. This API assumes an integration between SystemC and the HDL simulator that performs synchronization at the delta cycle level and that maintains a single global time. Such a tight integration is necessary to handle common design scenarios, such as the case where a register is instantiated in SystemC and a register is instantiated in HDL, and they are cross coupled and are driven by a common clock. All conforming implementations of the *scv_connect()* calls must properly simulate the design just described.

The following table describes the SystemC Verification Standard API for HDL connection:

Basic HDL Connection API	Description

enum scv_hdl_direction {
SCV_INPUT = 1,
SCV_OUTPUT = 2
};
	This enumeration represents the possible directions of the connection
	• SCV_INPUT: HDL is the only driver
	• SCV_OUTPUT: SystemC is the only driver

template < typename T> void scv_connect(sc_signal<T> & signal, const char * hdl_signal, scv_hdl_direction d = SCV_OUTPUT, unsigned hdl_sim_inst = 0);	This function connects an sc_signal object to an HDL signal. The *hdl_sim_inst* argument supports the scenario where multiple HDL simulator instances are combined in one overall simulation. Its value indicates which HDL simulator instance the signal should be connected to. The value 0 indicates the default simulator if there is only one HDL simulator. The value 1 indicates the first instance of a Verilog simulator. The value 2 indicates the first instance of a VHDL simulator. The meaning for values larger then 2 are vendor/simulator dependent.

void scv_connect(sc_signal_resolved& signal, const char * hdl_signal, scv_hdl_direction d = SCV_OUTPUT, unsigned hdl_sim_inst = 0);	This function connects an sc_signal_resolved object to an HDL signal.
template < int W> void scv_connect(sc_signal_rv<W>& signal, const char * hdl_signal, scv_hdl_direction d = SCV_OUTPUT, unsigned hdl_sim_inst = 0);	This function connects an sc_signal_rv object to an HDL signal.

The current specification only support SCV_INPUT and SCV_OUTPUT, but not a bi-directional connection. This is because bi-directional connections cannot be implemented with existing standards such as PLI, and the semantics related to such a connection is unclear. In the future, we may add SCV_INOUT to the *scv_hdl_direction* enumeration with a well-defined semantic.

The connection must be made during the elaboration phase (before *sc_initialize()* or *sc_start()*). A SystemC signal can only be connected to one HDL signal, and similarly a HDL signal can only be connected to one SystemC signal. These APIs are similar to those for out-of-module references in Verilog. The format of the HDL signal string must adhere to the specific HDL language domain within which the HDL signal resides.

- When a connection is made using SCV_OUTPUT, the SystemC signal controls the propagation of values. When the value is assigned to the SystemC signal, the value is propagated to the HDL signal. The behavior is undefined if a new value is assigned to the HDL signal from the HDL description.

  When *sc_signal<T>* is used in SystemC 2.0, there is no resolution of values when multiple SystemC drivers try to write to the SystemC signal; the last assignment wins. When *sc_signal_resolved* or *sc_signal_rv<W>* is used, resolution according to the SystemC semantic is performed before the value is propagated to the HDL signals.

- When a connection is made using SCV_INPUT, the HDL signal controls the propagation of values. When the value is assigned to the HDL signal, the value is propagated to the SystemC signal. It is an error if a new value is assigned to the SystemC signal from the C++ code.

The HDL connection can connect signals with compatible types, as specified in the following tables. If the type is not compatible according to the table, an error should be reported. A warning will be reported if a connection is being made between an unsigned type and a signed type, such as a connection between unsigned bit vector in Verilog and *sc_int<W>* in SystemC.

Verilog	**SystemC**
scalar	sc_bit, sc_logic, sc_signal_resolved.
bit vector of size W (both	sc_bv<W>, sc_lv<W>, sc_int<W>, sc_uint<W>, sc_bigint<W>,

signed and unsigned)	sc_biguint<W> sc_signal_rv<W>.
integer	sc_int<32>

VHDL	SystemC
bit, std_logic, std_ulogic	sc_bit, sc_logic, sc_signal_resolved.
bit vector, std_ulogic_vector of size W, std_logic_vector of size W.	sc_bv<W>, sc_lv<W>, sc_int<W>, sc_uint<W>, sc_bigint<W>, sc_biguint<W>, sc_signal_rv<W>.

In all cases, it is legal to bind a vector of length one to a scalar.

The HDL connection must be performed on these types of HDL signals only. It is an error to connect to a bit-slice of an HDL signal, a Verilog memory, or a Verilog memory element.

In the reference implementation of SystemC without an HDL simulator, these functions will be implemented as place-holders and will not actually do anything. It will be up to individual EDA vendors to integrate this interface to their HDL simulator.

7.2 Sparse Array

The SystemC Verification Standard contains several generic data structures. For example, in Section 5.4, a bag is used to describe a weighted distribution for the randomization facility.

One of the requirements from the Verification Working Group is to support memory modeling in multiple levels of abstraction. Similarly, it is natural for people to ask for other kinds of verification IP. We propose to include a set of frequently-used data structures in the standard to facilitate, for example, modeling of memory as a sparse array. It is the responsibility of IP vendors to provide other models in other abstraction levels.

Because SystemC 2.0 and the SystemC Verification Standard are in C++, users can use the C++ standard template library (or the Microsoft Foundation Classes if they are using Visual C++) for many frequently-used data structures. In order to support memory modeling, the SystemC Verification Standard includes a sparse array, which can be regarded as a memory-efficient representation of a memory at the highest level of abstraction.

The scv_sparse_array Template Class	Description
template<typename I, typename T> class scv_sparse_array;	This template represents a sparse array indexed by an integer. The type of the index is I, and the type of each array element is T. The supported index types are C/C++ integer types and SystemC integer types.
scv_sparse_array::scv_sparse_array(const char * name = 0, const T& default_value = T(), const I& lowerbound, const I& upperbound,);	A constructor with an object name, a default value for the array elements, and the upper and lower bound of the indices.
scv_sparse_array::scv_sparse_array(const scv_sparse_array& other, const char * name = 0);	The copy constructor
scv_sparse_array& scv_sparse_array::operator=(const scv_sparse_array&);	The assignment operator.
const T& scv_sparse_array::operator[ ] (const I& i) const;	This operator returns the element

	corresponding to the index i.
T& scv_sparse_array::operator[ ] (const I& i);	This operator returns the element corresponding to the index i.

7.3 Exception Handling

Exception handling is an important element in verification. The purpose of a test bench is to find bugs in a design, and bugs can be reported as exceptions. Because verification IP also needs to report errors to the user, the SystemC Verification Standard includes an API from which the SystemC implementations, current and future SystemC extensions, verification IP, SystemC designs, and SystemC test benches can report exceptions. In doing so, a SystemC implementation can generate a summary of all exceptions reported through the standard API, and pass it to other tools for analysis.

The basic exception-handling facility already in SystemC version 2.0.1 allows the reference implementation to report errors in a consistent way. The new API presented here extends such an effort to provide a facility that is more configurable and has the ability to report user-specified exceptions.

This facility is captured in a *scv_report* and *scv_report_handler* classes, with the following goals:

- It must be very similar to the existing SystemC 2.0.1 *sc_report* facility, so that it can easily be merged with it with very little impact on existing SystemC code.
- It must be highly configurable so that users can precisely control the actions that the exception facility will undertake when different kinds of exceptions occur.
- It must be precisely configurable to support both C++ style exception actions (throw/catch) as well as C-style exception actions (similar to POSIX “perror()”).
- It must allow any number of other libraries or user models to easily use the same exception API. (As an example, the existing exception API in SystemC 2.0.1 makes this difficult because its message IDs are integers with pre-defined values. Preventing ID clashes between different libraries or models that have been independently developed is thus difficult.) By having all libraries and models use the same exception API, exceptions can be reported to the user in a consistent way.

This new exception handling API will also be available in SystemC 2.1, as *sc_report* and *sc_report_handler*. At that point, the SystemC Verification Library will use the SystemC implementation of the API instead of its own. We will include typedefs so that *scv_report* and *scv_report_handler* can still be used (but they’ll refer to the corresponding SystemC classes).

An occurrence of an exception is hereafter referred to as a *report.* The *scv_severity* enum is used to classify the severity of a report:

The scv_severity enum	Description

enum scv_severity {
SCV_INFO = 0,
SCV_WARNING,
SCV_ERROR,
SCV_FATAL
};
	This enumeration describes the severity of a report.
	• SCV_INFO: The report is informative only.
	• SCV_WARNING: The report indicates a potentially incorrect condition.
	• SCV_ERROR: The report indicates a definite problem during execution.
	• SCV_FATAL: The report indicates a problem which cannot be recovered

	from. By default, the simulation is terminated immediately after reporting a SCV_FATAL report.

It is the job of the exception package to determine what actions to take when a report occurs. For a given report, typically multiple actions are taken. The *scv_actions* type is used to specify most of the actions that the exception API can take for a report. There are several predefined values:

- SCV_UNSPECIFIED: Take the action specified by a configuration rule of lower precedence. (see the description of the *set_action()* methods below about precedence.)
- SCV_DO_NOTHING: Don't take any actions for the report.
- SCV_THROW: Throw a C++ exception that represents the report.
- SCV_LOG: Print the report into the report log, typically a file on disk.
- SCV_DISPLAY: Display the report to the screen, typically by printing to "cout".
- SCV_CACHE_REPORT: Save the report into a cache so that calling code can interrogate it (similar to POSIX *perror*()).
- SCV_STOP: Call *sc_stop*(). No further simulation can be done, but the simulator remains "in control". Simulator GUI stays alive and available. On exit all normal cleanup code, manual destructors, etc. are executed.
- SCV_ABORT: Call *abort*().
- SCV_INTERRUPT: Interrupt simulation if simulation is not being run in batch mode.

Each exception report can be configured to take one or more *scv_actions*. Multiple actions can be specified using bit-wise OR. When SCV_DO_NOTHING is combined with any thing other than SCV_UNSPECIFIED, the bit is ignored by the facility. Please see later discussion for an example.

In addition to the actions specified via *scv_actions*, the exception API also can take two additional actions. The first action is always taken: the *sc_stop_here()* function is called for every report, thus providing users a convenient location to set breakpoints to detect error reports, warning reports, etc. The second action that can be taken is to immediately abort the simulation (via sc_stop()). The stop action is configured via the *stop_after()* method described below, which allows users to set specific limits on the number of reports of various types that will cause simulation to abort.

For the SCV_CACHE_REPORT feature to work reliably in the presence of multiple SystemC threads, it seems necessary that thread-specific storage be used to cache reports. Perhaps a single *sc_attribute* on a process can be used for this purpose.

The SystemC 2.0.1 exception API uses integer IDs to classify various message types of reports. We have decided to switch from an integer ID to a simple literal character string ID in the new exception API, because it is difficult to avoid ID clashes with independently developed libraries and models if integer IDs are used. Using string IDs greatly reduces the chances of a clash and also allows the ID itself to provide useful description and classification information to the user. The new API uses the following typedef for message types:

```
typedef const char * scv_msg_type;
```

In the SystemC 2.0.1 exception package, all of the possible message IDs needed to be pre-registered with the exception package prior to any generation of reports. In the new API, no pre-registration is used since this is unnecessary and it is error-prone if pre-registration is not optional.

The exception API is contained within the *scv_report* and *scv_report_handler* classes.

The scv_report_handler Class	Description

static void set_handler(
  void (*handler)(
  const scv_report&, const scv_actions&
) = 0
);
	Specify an alternate report handler. You can revert to the default handler by passing in 0. If an alternate handler has been specified, *scv_report_handler::report*() calls it after looking up the scv_action based on the msg_type and severity. The alternate handler can pass control back to the default handler by calling *scv_report_handler::default_handler*().

static scv_actions set_actions(
  scv_severity severity,
  scv_actions actions = SCV_UNSPECIFIED
);
	Configure the set of actions to take for reports of the given severity. (Lowest precedence match.) The previous actions set for this severity is returned as the result. SCV_UNSPECIFIED is returned if there was no previous actions set for this severity.

static scv_actions set_actions(
  scv_msg_type msg_type,
  scv_actions actions = SCV_UNSPECIFIED);
	Configure the set of actions to take for reports of the given message type. (Middle precedence match.) The previous actions set for this message type is returned as the result. SCV_UNSPECIFIED is returned if there was no previous actions set for this message type.

static scv_actions set_actions(
  scv_msg_type msg_type,
  scv_severity severity,
  scv_actions actions = SCV_UNSPECIFIED
);
	Configure the set of actions to take for reports having *both* the given message type and severity. (Highest precedence match.) The previous actions set for this message type *and* severity is returned as the result. SCV_UNSPECIFIED is returned if there was no previous actions set for this message type *and* severity

static int stop_after(
  scv_severity severity,
  int limit = -1
);
	Call *sc_stop*() after encountering *limit* number of reports of the given severity. (Lowest precedence match.) If *limit* is set to one, the first occurrence of a matching report will cause the stop action. If *limit* is 0, stop action will never be taken due to a matching report. If *limit* is negative, stop action will never be taken for non-fatal error, and stop action will be taken for the first occurrence of a fatal error. The previous limit for this severity is returned as the result. The stop_after() call will return UINT_MAX in the case where no previous corresponding stop_after() call was made.

static int stop_after(
  scv_msg_type msg_type,
  int limit = -1
);
	Call *sc_stop*() after encountering *limit* number of reports of the given message type. (Middle precedence match.) The previous limit for this message type is returned as the result. If limit is 0, stop action will never be taken due to a matching report. If limit is negative, the limit specified by a lower precedence rule is used. The stop_after() call will return UINT_MAX in the case where no previous corresponding stop_after() call was made.

static int stop_after(
  scv_msg_type msg_type,
  scv_severity severity,
  int limit = -1
);
	Call *sc_stop*() after encountering *limit* number of reports having *both* the given message type and severity. (Highest precedence match.) If limit is 0, stop action will never be taken due to a matching report. If limit is negative, the limit specified by a lower precedence rule is used. The previous limit for this message type and severity is returned as the result. The stop_after() call will return UINT_MAX in the case where no previous corresponding

	stop_after() call was made.
static scv_actions suppress(scv_actions actions);	Suppress specified actions for subsequent reports regardless of configuration and clears previous calls to *suppress()*. The return value is the actions that were suppressed prior to this call.
static scv_actions suppress();	Restore default behavior by clearing previous calls to *suppress()*. The return value is the actions that were suppressed prior to this call.
static scv_actions force(scv_actions actions);	Force specified actions to be taken for subsequent reports in addition to the actions specified in the current configuration and clears previous calls to *force()*. The return value is the actions that were forced prior to this call
static scv_actions force();	Restore default behavior by clearing previous calls to *force()*. The return value is the actions that were forced prior to this call.
static scv_actions get_new_action_id();	Return an unused *scv_actions* value. Returns a different value each time it is called (returns SCV_UNSPECIFIED if no more unique values are available). Used when establishing user-defined actions, interpreted by a non-default report handler.
static const char *get_log_file_name();	Return the log file name currently in effect.
static void set_log_file_name(const char *name);	Set the log file name. The log file name in effect when the first log entry is made determines the actual name of the log file. Once the log file has been created, subsequent calls to *set_log_file_name*() are ignored.
static void report(scv_severity severity, scv_msg_type msg_type, const char * msg, const char *file, int line);	Generate a report instance, which will cause the exception package to take the appropriate actions based on the current configuration.
static void default_handler(const scv_report& report, const scv_action& action);	Generate a report instance, but always use the default report handler (ignore calls to *set_handler*()). This method is used in alternate message handlers to pass messages on to the default handler.
static const scv_report* get_cached_report();	Return pointer to cached report available for the current process if one is available.
static void clear_cached_report();	Clear cached report for the current process (if any).

The *force()* and *suppress()* methods provide a brute-force way to override the current configuration. For example, *force(SCV_LOG)* could be called during debugging to cause all reports to be logged regardless of the current configuration. As another example, "*scv_actions* prev = *suppress(); suppress(*prev | *SCV_THROW);*" could be called by code that is not C++ throw-safe when it starts execution, and then *suppress(*prev*)* would be called when it completes execution. To avoid affecting other threads that are executing other code, the above *suppress()* calls would need to be made without any intervening *wait()* statements.

If no calls to *stop_after*(SCV_FATAL, ..) are made, or if no *stop_after*(SCV_FATAL, ..) rules are currently in effect, then the default behavior of the exception API will be to call *sc_stop*() after the occurrence of the first fatal error. When the exception API is to take an abort action, it will first take all other actions that are in effect for the report (e.g. SCV_LOG, SCV_DISPLAY) and then it will abort without performing a throw action.

Once *set_handler*() is called to establish a separate handler for one or more reports, none of the usual processing for these reports (logging, fatal error counting, etc.) takes place, unless the new handler calls default_handler().

The scv_report Class	Description
scv_severity get_severity() const;	Get severity of a report object.
const char* get_msg_type() const;	Get message type of a report object.
const char* get_msg() const;	Get message contents of a report object.
const char* get_file_name() const;	Get file name that generated report object.
int get_line_number() const;	Get line number that generated report object.
sc_time get_time() const;	Get the simulation time that report object was generated.
const char* get_process_name() const;	Get the process name that generated the report object.

When a report is logged to a file, the current simulation time and current process name will automatically be included within the report, similar to the *sc_report_handler::compose_message()* method within the existing SystemC 2.0.1 API.

An implementation of the SystemC Verification Standard can decide what the actions are initially set to. For example, an implementation may implement the system defaults as follows:

```
// These four constants are globally visible:
const scv_actions SCV_DEFAULT_INFO_ACTIONS    = SCV_LOG;
const scv_actions SCV_DEFAULT_WARNING_ACTIONS = SCV_LOG | SCV_DISPLAY;
const scv_actions SCV_DEFAULT_ERROR_ACTIONS
        = SCV_LOG | SCV_DISPLAY | SCV_CACHE_REPORT | SCV_THROW;
const scv_actions SCV_DEFAULT_FATAL_ACTIONS
        = SCV_LOG | SCV_DISPLAY | SCV_CACHE_REPORT | SCV_THROW;
void scv_report_handler::initialize() {
        // actions for each severity level must be present in initialize()
        set_actions(SCV_INFO,    SCV_DEFAULT_INFO_ACTIONS);
        set_actions(SCV_WARNING, SCV_DEFAULT_WARNING_ACTIONS);
        set_actions(SCV_ERROR,   SCV_DEFAULT_ERROR_ACTIONS);
        set_actions(SCV_FATAL,   SCV_DEFAULT_FATAL_ACTIONS);
        // other actions for messages reported from the SystemC Verification
        // library may also be specified here.
        set_actions("SCV_CONFIGURATION_REPORT", SCV_LOG | SCV_DISPLAY);
}
```

The following macros are globally visible as part of the standard and should be used to generate reports:

```
#define SCV_REPORT_INFO(msg_type, msg) \
```

```
        scv_report_handler::report( SCV_INFO, msg_type, msg, __FILE__, __LINE__ )
#define SCV_REPORT_WARNING(msg_type, msg) \
        scv_report_handler::report( SCV_WARNING, msg_type, msg, __FILE__, __LINE__ )
#define SCV_REPORT_ERROR(msg_type, msg) \
        scv_report_handler::report( SCV_ERROR, msg_type, msg, __FILE__, __LINE__ )
#define SCV_REPORT_FATAL(msg_type, msg) \
        scv_report_handler::report( SCV_FATAL, msg_type, msg, __FILE__, __LINE__ )
```

The following example illustrates how the exception API might be custom configured and how reports are generated. Note that message types are best captured within one or more header files, where they are declared using #define macros. This technique insures that strings representing message types are only declared once and that any typos that might occur when message types are specified in the *SCV_REPORT_** macros are caught by the compiler.

```
const char *SCV_RPT_INVALID_THREAD_HANDLE = "SCV invalid thread handle";
const char *PCI_RPT_PROTOCOL_EXCEPTION = "PCI Protocol Exception";
const char *PCI_RPT_PROTOCOL_READ_RETRY = "PCI Read Retry";
sc_main() {
        // Custom configure the exception package:
        scv_report_handler::stop_after(SCV_ERROR, 10);
        scv_report_handler::set_actions(PCI_RPT_PROTOCOL_READ_RETRY, SCV_DO_NOTHING);
        // PCI_RPT_PROTOCOL_READ_RETRY reports will now be completely ignored…
        sc_start(1, SC_MS);
        scv_report_handler::set_actions(PCI_RPT_PROTOCOL_READ_RETRY, SCV_DISPLAY);
        // PCI_RPT_PROTOCOL_READ_RETRY reports will now be displayed to the screen
        sc_start(1, SC_MS);
        scv_report_handler::set_actions(PCI_RPT_PROTOCOL_READ_RETRY);
        // SCV_REPORT_INFO(PCI_RPT_PROTOCOL_READ_RETRY, …) reports will now
        // be configured to SCV_UNSPECIFIED. Therefore, a lower precedence
        // rule applies and the actions in SCV_DEFAULT_INFO_ACTIONS will take
        // effect for PCI_RPT_PROTOCOL_READ_RETRY . Note that we do not go back
        // to the previous SCV_DO_NOTHING action for PCI_RPT_PROTOCOL_READ_RETRY.
        sc_start(1, SC_MS);
}
void foo() {
        if (thread_handle.in_use())
                SCV_REPORT_ERROR(SCV_RPT_INVALID_THREAD_HANDLE,
                        "Thread handle is invalid because it is already in use");
```

```
}
void bar() {
        sc_time max_time(500, SC_NS);
        if (...)
                SCV_REPORT_WARNING(PCI_RPT_PROTOCOL_EXCEPTION,
                        "PCI burst read exceeded max time limit of " + max_time.to_string());


        if (...)
                SCV_REPORT_INFO(PCI_RPT_PROTOCOL_READ_RETRY,
                        "PCI read retry at time " + sc_time_stamp().to_string());
}
```

The following example illustrates how reports using *SCV_CACHE_REPORT* actions can be accessed:

```
void c_style_example() {
        // POSIX perror() style exception:
        scv_report_handler::clear_cached_report();
        execute_my_routine();
        if (scv_report* rp = scv_report_handler::get_cached_report()) {
                cout << rp->get_msg() << endl;
        }
}
```

The following example illustrates how reports using *SCV_THROW* actions can be accessed:

```
void cpp_style_example() {
        // C++ style exception:
        try {
                execute_my_routine();
        } catch (sc_exception e) {
                cout << e.get_report().get_msg() << endl;
        }
}
```

The C++ exception handling style may be preferable to the POSIX perror() style over the long term. At the moment, the current Verification Standard prototype supports the POSIX *perror()* style exception handling only, primarily because it is difficult to create a library that is completely throw-safe. The current SystemC 2.0.1 release does rely on *throw* for handling errors. The presence of *scv_shared_ptr* template makes it a little bit easier to create throw-safe code, but it is still not foolproof.

The goal is to make the reference implementation for the SystemC Verification Standard throw-safe, but if it is not throw-safe, this fact can be documented. It will be up to individual EDA vendors to decide whether to support C++-style exceptions in their implementations of the Verification Standard and in other SystemC libraries and models.

8 Verification Features Not Addressed by This Specification

Some verification features and requirements were discussed during the VWG meetings that are not incorporated into this specification. A short summary about them is recorded in this section. The descriptions of them are not yet formulated with sufficient details to be proposed as part of a standard. More investigation is needed to clarify the requirements from various parties and to design an appropriate API to address them.

Concurrency and Complex Synchronization

A proposal for concurrency and complex synchronization has been submitted by Cadence to the SystemC Verification Working Group (VWG). However, during the VWG discussion, we realized that this proposal is similar to the current activities in the SystemC Language Working Group (LWG), with some interesting differences. Although the goal of the LWG is to enhance SystemC to support embedded software, the actual requirements [13] are very similar to the VWG requirements [4]. As a result, we have decided to defer the specific proposal in this area until we can determine a consistent API that supports both verification and embedded software.

Interface Introspection

We have discussed an extension of the data introspection facility to handle interface [14,15]. However, we have not reached a conclusion as to whether to propose it as part of the SystemC Verification Standard. A short description is included in the appendix for your references. This is not part of the proposal; it is included there for reference and archive purpose and to stimulate new ideas. Because we have not discussed it in details or agree upon the API, if we decided to include interface introspection in a later proposal, it may take a different form.

Assisted Transaction Recording

As discussed in the section on transaction recording, we are looking into different ways of automating or assisting the process of transaction recording. Several styles have been considered [12,15], but we have not reached a conclusion about which styles of automation should be part of the standard. The appendix contains several examples for your references. This is not part of the proposal; it is included there for reference and archive purpose and to stimulate new ideas. Because we have not discussed it in details or agree upon the API, if we decided to include assisted transaction recording in a later proposal, it might take a different form.

Transaction Retrieval API

Apart from recording transactions into a database during simulation, we have briefly discussed the API for retrieving transactions from a database during simulation. From a verification standpoint, it might make sense to write a SystemC model that could open the database and read the stored transactions, for the purposes of replaying it or for the purposes of generating other transactions. However, any proposed API for retrieving transactions from databases would be independent and would not affect the current API, and defining such an API may take a fair amount of time & effort. Because the existing spec provides enough capabilities to enable creation of most verification IP and testbenches, we have decided to defer discussion on this topic.

Non-temporal and Temporal Assertions

Several assertion efforts have been discussed. Cadence's TestBuilder team has temporal assertion capability in the TestBuilder 1.3 release [1]. The Accellera Formal Verification Technical Committee has voted and selected Sugar 2.0 [16] (which includes a Verilog dialect and a VHDL dialect) as the standard language for assertion-based verification in April 2002. The SystemC standard for assertion must adhere to Sugar 2.0 and implements a subset of the Sugar 2.0 standard in a C++ dialect. We have not designed the API for this aspect of the standard yet.

Coverage

We have briefly discussed the requirement for collecting coverage information in a test bench. In the simplest case - just collecting the coverage information and presenting it to the user - it is probably a tool issue to be solved by tool vendors instead of part of the proposal. In the more advanced cases of using coverage information to dynamically change the test bench, a standard is probably needed. Because it represents a facility that is independent from the facility in this proposal, we have postponed the discussion on this topic.

Temporal Constraints

A simple temporal constraint can typically be translated into a process thread with state variables and non-temporal constraints. The values at a specific time can be generated according to the value of the state variable. In the complex cases where a value generated at an earlier time might cause a conflict (no legal value can be found) at a later time, sequential ATPG is required to generate the sequence of values to satisfy the temporal constraint. As such, the technology is not mature enough for us to create a standard interface to specify temporal constraints for such a sequential ATPG solver. Temporal constraints can also be regarded as a twin to temporal assertions.

9 References

1. C. Norris Ip and Stuart Swan, *Using Transaction-Based Verification in SystemC*, Cadence Design Systems, Inc. This is a white paper available at http://www.systemc.org.
2. TestBuilder releases 0.92 to 1.3, document and source code available at http://www.testbuilder.net.
3. SystemC releases 1.0 and 2.0, document and source code available at http://www.systemc.org.
4. Adam Rose*, SystemC Functional Verification Requirement Spreadsheet*, SystemC Verification Working Group, 2001 & 2002.
5. Scott Meyers, *More Effective C++,* Addison-Wesley, 1996. For information about smart pointers, see item 28, smart pointers and item 29, reference counting.
6. C. Norris Ip, Stuart Swan, and Jasvinder Singh. A *Prototype Implementation for Data Introspection and Constraint Specification*, Cadence Design Systems, February 2002. This is a prototype with source-code circulated among the VWG members.
7. Nathan C. Myers, *Traits: a new and useful template technique*, C++ Report, June 1995. The paper is available at http://www.cantrip.org/traits.html.
8. Thaddaeus Frogley, *An introduction to C++ traits*, Overload #43. The paper is available at http://thad.notagoth.org/cpptraits_intro.
9. John Maddock and Steve Cleary, *C++ Type traits*, Dr Dobb's Journal, October 2000. The paper is available at http://www.boost.org/libs/type_traits/c++_type_traits.htm.
10. *the Boost type traits library*. Information about this library is available at http://www.boost.org/libs/type_traits.
11. Grzegorz Jakacki, *Extensions*, China Integrated Circuit Design Center, September 21, 2001. This is a proposal submitted to the SystemC Language Working Group.
12. C. Norris Ip, Bill Paulsen, and John Rose. *Transaction Recording : an elaboration of section 4.3 of the SystemC Verification Proposal*, Cadence Design Systems. This is a document circulated among VWG members, May 16, 2002.
13. Johan Cockx, *Requirements for Software Modeling in SystemC 3.0*, IMEC, Version 1.2, July 8, 2002. This is a document circulated among LWG members.
14. Mike Meredith, *Interface Introspection Examples*, Forte Design Systems, May 21, 2002. This is a document circulated among the VWG members.
15. Mike Meredith, Steve Sutherland, Andrew Fairley, and Sean Dart. *Proposal of Technology for Incorporation in SystemC Verification Facility*, Forte Design Systems, April 16, 2002 This is a document circulated among VWG members.
16. Cindy Eisner and Dana Fisman, *Sugar 2.0 : An Introduction*, IBM Haifa Research Laboratory and Weizmann Institute of Science, Rehovot, Israel, May 2002. This is a tutorial available at http://img.cmpnet.com/eedesign/2002/may/sugar_tutorial.pdf.
17. Bobby Schmidt, Partial Template Implementation, Microsoft Corporation, July 9, 2002. This is an internet posting available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndeepc/html/deep07092002.asp.
18. Thorsten Groetker, Stan Liao, Grant Martin, Stuart Swan, “System Design with SystemC”. Book available at www.systemc.org -> Products and Solutions -> Books.

Appendix A: Requirements Summary

Topics in the VWG requirement list

Requirement Identifiers	Brief Description	VWG Discussion Summary	Section that Addresses these Requirements
R1 (1-8)	general requirement about look and feel and applicability in different application domains	---	subjective judgment for the overall proposal
R2 (1-6)	general requirement about reproducibility and seed controls in a multi-thread environment	---	Section 5.1
R3 (1-d)	non-temporal constraint specification for the purpose of generating a random value (aspects regarding expressive power)	discussed a possible API during the introspection prototype presentation on Feb 20, 2002	Section 1.1
R4 (1-5)	non-temporal constraint specification for the purpose of generating a random value (aspects regarding ease of use and efficient implementation)	discussed a possible API during the introspection prototype presentation on Feb 20, 2002	Section 1.1
R5 (1-6)	Weight specification for biased random value generation	identify potential subtle interaction with non-temporal constraints during the constraint solving process	Section 5.4
R6 (1-c)	functional coverage measurement and on-the-fly feedback to the test bench	need more investigation; defer to later discussion	Section 8
R7 (1-2)	non-temporal and temporal assertions for error detection	need more investigation; defer to later discussion.	Section 8
R8 (1-5)	transaction recording to generate a database from which other tools can analyze simulation activities in terms of transactions	several email discussions around January 2002, and a possible API was circulated on March 20, 2002.	Section 6.2
R9 (1-4)	automatic translation of a description in a transactor description language to an executable C++ transactor	the technology seems to be premature and should be considered as an API independent of the verification standard - a possible prototype might may be distributed with SystemC	Not Applicable
Ra (1-a)	Modeling a transactor as an adaptor between a transaction-level test and a signal-level design	an example using the current SystemC 2.0 standard has been circulated; it seems that no new API is needed except for transaction recording	Section 3.1
Rb (1-7)	the ability to handle arbitrary data types in various facilities	discussed a possible API during the introspection prototype presentation on Feb 20, 2002	Section 4
Rc (1-a)	general multi-thread support, including dynamic thread manipulation, complex synchronization, and sensitivity on transactions	need to defer this proposal until we can determine its relationship to the LWG activities on embedded software	Not discussed
Rd (1)	miscellaneous (memory modeling)	---	Section 7.2

Summary for the general requirement

Requirement Identifier	Brief Description	Proposal
R11	look and feel	we believe the current proposal has the SystemC look and feel
R12	multiple language support	not considered / not ready to send proposal yet
R13	protocol hierarchy	communication refinement in SystemC 2.0
R14	complex buses	transactor modeling style discussed in Section 3
R15	combine split arbitrate	transactor modeling style
R16	dynamic threads and complex scheduling	proposal deferred to a later time
R17	verification patterns	we believe the general facility proposed here is sufficient to support many verification patterns
R18	communication types	we believe the general facility proposed here is sufficient for modeling complex communication patterns

Summary for transactor modeling

Requirement Identifier	Brief Description	Proposal	Use Case Scenarios (taken from the rw_pipelined_transactor example in this section)
Ra1	verification at transaction level	adapting communication refinement in SystemC 2.0	class rw_pipelined_transactor : public rw_task_if, public rw_bus_ports { ... };

Ra2	transaction-level test cases should be reusable	adapting communication refinement in SystemC 2.0	class test : public sc_module {
			public:
			sc_port<rw_task_if> transactor;
			...
			};

Ra3	separation of test case and transactors	object-oriented programming	separation of the test bench into two classes: rw_pipelined_transactor and test;
Ra4	translation from a parameterized transaction command to wire-level protocol	adapting communication refinement in SystemC 2.0	class rw_pipelined_transactor implementing the abstract interface rw_task_if by manipulating the signals in class rw_pipelined_bus_ports
Ra5	plug and play with RTL (HDL simulators?)	Basic HDL connection	discussed in Section 7.1
Ra6	pipelined transactions	Dynamic threads + thread synchronization	data_t rw_pipelined_transactor::read(addr_t * addr) { ... }
Ra7	split transactions	Dynamic threads + thread synchronization	data_t rw_pipelined_transactor::read(addr_t * addr) { ... }
Ra8	single command	dynamic thread + thread synchronization	data_t rw_pipelined_transactor::read(addr_t * addr) { ... }
Ra9	multi-thread	Static and dynamic threads	multiple process threads can each call read()/write() on the same or multiple instances of rw_pipelined_transactor object. A single process thread can spawn multiple dynamic threads to execute read()/write() concurrently.
Raa	multi-language	basic HDL connection	discussed in Section 7.1

Summary for the supports of arbitrary data types

Requirement Identifier	Brief Description	Proposal	Use Case Scenarios
Rb1	supporting arbitrary data types	template specialization of *scv_extensions<T>*	---

Rb2	information extraction from data objects in order to generate random values	abstract interface class *scv_extensions_if*	void generate_random_value
			(scv_extensions_if * data) {
			...
			cout << data->get_bit_width();
			...
			}

Rb3	information extraction from data objects in order to record values	abstract interface class *scv_extensions_if*	void record_value
			(scv_extensions_if* data) {
			...
			cout << data->get_integer();
			...
			};

Rb4	Value change callbacks on data objects	template wrapper class *scv_smart_ptr<T>*
		(see also Section 6.1)
			class my_module : public sc_module {
			public:
			scv_smart_ptr<fsm_state_t> state;
			SC_CTOR(my_module) {
			state.register_cb(...);
			...
			}
			};

Rb5	Find design element in hierarchy	explicit class member access
		(or through a registry)
			int sc_main(...) {
			...
			design.module1 .fsm_state.register_cb(...);
			...
			}

Rb6	peek/poke (no scope rule that limits access)	use *sc_signal_rv* or enable write from multiple threads in *sc_signal*
		(current reference implementation prints an error when sc_signal is used with *DEBUG_SYSTEMC* turned on)
			int sc_main(...) {
			...
			top.module1.port_data = 1;
			...
			}

Rb7	Block a process	not included in this proposal	---

Summary for constrained randomization

Requirement Identifier	Brief Description	Proposal	Use Case Scenarios
R21	reproducibility	global seed, thread-based seeds, scv_random, and seed files	---
R21	unique per thread (for automatically generated seeds)	generation of a seed for each thread by a deterministic transformation from the global seed using the thread name	---
R23	independent seeding	scv_random	scv_random gen1("gen1",999); scv_random gen2("gen2",1001);
R24	global configuration of seed management	static methods in scv_random	scv_random::set_global_seed(999);
R25	unique default seeds	generation of default seeds from the unique seed in the current thread.	scv_random gen1("gen1");

R26	ability to range over list of objects	weighted distribution on the list of object	scv_bag<int> b; b.push(1);
			b.push(2);
			scv_smart_ptr<int> data;
			data->set_mode(b);

			data->next();
R 31	arithmetic and logical constraints	scv_expression	write->addr() != write->data()
R32	if/else in constraints	convert to Boolean implication	(!write->addr() == 0xFF) || write->data() < 1000
R33	grouping constraints into groups with guard	scv_extensions_if::use_constraint()	see example in main text with my_select_constraint().
R34	specifying ranges as constraints	convert to comparison operator	0 < a() && a() < 100
R35	classifying variables into variables with fixed values and variables with values to be randomly generated	scv_extensions_if:: disable_randomization()	scv_smart_ptr<my_record> data; data->field1.disable_randomization(); data->next();
R36	class-based constraint specification	base class scv_constraint_base	class write_constraint : virtual public scv_constraint_base { ... }
R37	using scope in constraints	class variable in a constraint derived from scv_constraint_base	class write_constraint ... { public: scv_smart_ptr<int> data; ... }
R38	using inheritance in constraints	class inheritance in C++	class hierarchical_constraint : public write_constraint, public complex_constraint { ... }
R39	Ranging over lists	weighted distribution on the list	scv_bag<int> b; b.push(1); b.push(2); scv_smart_ptr<int> data; data->set_mode(b); data->next();
R3a	Constraining the size of a variable-sized data object	use a fixed size object with invalid elements to mimic a variable-sized object	---
R3b	constrained randomization should be simple to use	using scv_smart_ptr as if it was a real C pointer with randomization methods	class write_constraint ... { ...} void my_test() { write_constraint c("c"); c.next(); cout << *c.data << endl; }
R3c	seed control in constrained randomization	set_random()	scv_shared_ptr<scv_random> rand(new scv_random("rand",1999)); write_constraint c("c"); c.set_random(rand);
R3d	supports basic types and composite types in constraints	scv_smart_ptr	scv_smart_ptr<int> data1; scv_smart_ptr<ethernet_packet> data2;
R41	efficiency, scale, and complexity for constraint solver	class-based constraint specification enables pre-processing with low run-time overhead	class write_constraint : virtual public scv_constraint_base { ... }
R42	low user intervention	not a language issue. The proposal API assumes a fully autonomous constraint solver. Individual vendors may provide configuration methods for the user to configure the solver.	---
R43	order changing for solver	not a language issue. The proposed API assumes the solver does not depend on the order of constraint specifications. It uses a simple	---

		conjunction semantic for the list of constraints. Individual vendors may manipulate the constraint expression tree to rearrange the order.	
R44	debug in the case of deadlock	not a language issue. The proposed API assumes the solver eventually returns. Individual vendors may provide a solver that fails sometimes, and in those cases, trigger the exception handling mechanism mentioned in Section 4.5.2 Individual vendors may also provide configuration methods for the user to adjust the behavior.	---
R45	single run	the class-based constraint specification enables the expression to be processed once only using static variables within the class.	---
R51	weighting value ranges within variable domain	scv_bag<pair<T,T>>	---
R52	weighting single values within variable domain	scv_bag<T>	---

Summary for weight specifications

R53	varying weights over time	create an SC_METHOD or an SC_THREAD, and change the weighting over time using set_mode(scv_bag)	---
R54	generating values according to a user-defined distribution	scv_bag<pair<T,T>>	---
R55	weighting object within a set of objects	scv_bag<T>	---
R56	dynamic editing of the list in the weight specification	create an SC_METHOD or an SC_THREAD, and change the weighting over time using set_mode(scv_bag)	---

Summary for transaction recording

Requirement Identifier	Brief Description	Proposal	Use Case Scenarios
R81	playback from transaction recording database	not considered	---
R82	generated error within transactions	create another transaction to indicate error and link it to the current transaction	---
R83	transaction definition	see Section 6.2.2	---

R84	transaction recording	*scv_tr_db, scv_tr_stream, scv_tr_generator, and scv_tr_handle*	h = read_tr.begin_transaction(addr);
			...
			read_tr.end_transaction(h, data);

R85	Linking	*scv_tr_handle::add_relation()*	h = read_tr.begin_transaction(addr);
			h1 = addr_tr.begin_transaction(addr);
			h.add_relation("addr_phase",h1);

Summary for memory modeling

Requirement Identifier	Brief Description	Proposal	Use Case Scenarios
Rd1	modeling memory at multiple levels of abstraction	not a language requirement; it seems more like an application-specific library to be provided by a IP vendor	*scv_sparse_array* can be used as the highest level of abstraction for a memory

Appendix B: API convention

Name Space and Prefix

The name space "*systemc_verification_library*" and the prefix "*scv_*" are used in the Verification Standard to avoid name conflict.

Interface classes vs. Base classes

The SystemC Verification Standard contains several interface classes, such as *scv_extensions_if* and s*cv_object_if*. These interface classes use *"_if*" as postfix to indicate that they contain C++ abstract methods and that they do not contain member variables. A derived class from an interface class is expected to implement the abstract methods. On the other hand, the name convention for a base class is to have a postfix "_base", such as the *scv_extensions_base* template in the introspection facility. Typically a base class contains member variables and implements a set of methods common to all classes derived from this base class.

String

The type "*const char *"* is used in most places for strings, except in some functions or methods, which might return "*sc_string*". The guideline as discussed in LWG and VWG is summarized as follows:

- When a function or method uses a string argument, the type "const char *" is used. The function/method must copy the string if it intends to retain the string after the function returns.

- When a function or method returns a string:

    - It is returned as "const char *" if the content of the string stays the same throughout the lifetime of the simulation (for function and static method) or the lifetime of the object (for non-static method).
    - It is returned as "sc_string" if the content of the string may change during the lifetime of the simulation (for functions and static methods) or the lifetime of the object (for non-static methods).

Callbacks

Callbacks are registered:

- globally to a facility through methods with the name *register_class_cb()*, and
- locally to an object through methods with the name *register_cb()*.

Callbacks can be removed by calling a method with the name *remove_cb()*. The handle for the callbacks is declared as a type within the related class; the type has the name *callback_h*. Most callbacks use an enumeration within the related class to capture the possible reasons for which the callbacks are executed; the enumeration has the name *callback_reason*.

Appendix C: The Complete Code for the Overview Example

```
// this code compiles and runs with our latest prototype for this specification

#include "scv.h"
#include "fifo_mutex.h"

class rw_task_if : virtual public sc_interface {
public:
  typedef sc_uint<8> addr_t;
  typedef sc_uint<8> data_t;
  struct write_t {
    addr_t addr;
    data_t data;
  };

  virtual data_t read(const addr_t*) = 0;
  virtual void write(const write_t*) = 0;
};

SCV_EXTENSIONS(rw_task_if::write_t) {
public:
  scv_extensions<rw_task_if::addr_t> addr;
  scv_extensions<rw_task_if::data_t> data;
  SCV_EXTENSIONS_CTOR(rw_task_if::write_t) {
    SCV_FIELD(addr);
    SCV_FIELD(data);
  }
};

class pipelined_bus_ports : public sc_module {
public:
  sc_in< bool > clk;
  sc_inout< bool > rw;
  sc_inout< bool > addr_req;
  sc_inout< bool > addr_ack;
  sc_inout< sc_uint<8> > bus_addr;
  sc_inout< bool > data_rdy;
  sc_inout< sc_uint<8> > bus_data;

  SC_CTOR(pipelined_bus_ports)
    : clk("clk"), rw("rw"),
      addr_req("addr_req"),
      addr_ack("addr_ack"), bus_addr("bus_addr"),
      data_rdy("data_rdy"), bus_data("bus_data") {}
};

class rw_pipelined_transactor
  : public rw_task_if,
    public pipelined_bus_ports {

  fifo_mutex addr_phase;
```

```
  fifo_mutex data_phase;

  scv_tr_stream pipelined_stream;
  scv_tr_stream addr_stream;
  scv_tr_stream data_stream;
  scv_tr_generator<sc_uint<8>, sc_uint<8> > read_gen;
  scv_tr_generator<sc_uint<8>, sc_uint<8> > write_gen;
  scv_tr_generator<sc_uint<8> > addr_gen;
  scv_tr_generator<sc_uint<8> > data_gen;

public:
  rw_pipelined_transactor(sc_module_name nm) :
      pipelined_bus_ports(nm),
      addr_phase("addr_phase"),
      data_phase("data_phase"),
      pipelined_stream("pipelined_stream"),
      addr_stream("addr_stream"),
      data_stream("data_stream"),
      read_gen("read",pipelined_stream,"addr","data"),
      write_gen("write",pipelined_stream,"addr","data"),
      addr_gen("addr",addr_stream,"addr"),
      data_gen("data",data_stream,"data")
  {}
  virtual data_t read(const addr_t* p_addr);
  virtual void write(const write_t * req);
};

rw_task_if::data_t rw_pipelined_transactor::read(const rw_task_if::addr_t* addr) {
  addr_phase.lock();
  scv_tr_handle h = read_gen.begin_transaction(*addr);

  scv_tr_handle h1 = addr_gen.begin_transaction(*addr,"addr_phase",h);
  wait(clk->posedge_event());
  bus_addr = *addr;
  addr_req = 1;
  wait(addr_ack->posedge_event());
  wait(clk->negedge_event());
  addr_req = 0;
  wait(addr_ack->negedge_event());
  addr_gen.end_transaction(h1);
  addr_phase.unlock();

  data_phase.lock();
  scv_tr_handle h2 = data_gen.begin_transaction("data_phase",h);
  wait(data_rdy->posedge_event());
  data_t data = bus_data.read();
  wait(data_rdy->negedge_event());
  data_gen.end_transaction(h2);
  read_gen.end_transaction(h,data);
  data_phase.unlock();

  return data;
}

void rw_pipelined_transactor::write(const write_t * req) {
  scv_tr_handle h = write_gen.begin_transaction(req->addr);
  // ...
  write_gen.end_transaction(h,req->data);
}
```

```
class test : public sc_module {
public:
  sc_port< rw_task_if > transactor;
  SC_CTOR(test) {
    scv_thread::spawn("test::main",this,&test::main);
    //    SC_THREAD(main);
  }
  void main();
};

class write_constraint : virtual public scv_constraint_base {
public:
  scv_smart_ptr<rw_task_if::write_t> write;
  SCV_CONSTRAINT_CTOR(write_constraint) {
    SCV_CONSTRAINT( write->addr() < 0x00ff );
    SCV_CONSTRAINT( write->addr() != write->data() );
  }
};

inline void process(scv_smart_ptr<int> data) {}

inline void test::main() {
  // simple sequential tests
  for (int i=0; i<3; i++) {
    rw_task_if::addr_t addr = i;
    rw_task_if::data_t data = transactor->read(&addr);
    cout << "received data : " << data << endl;
  }

  scv_smart_ptr<rw_task_if::addr_t> addr;
  for (int i=0; i<3; i++) {

    addr->next();
    rw_task_if::data_t data = transactor->read( addr->get_instance() );
    cout << "data for address " << *addr << " is " << data << endl;
  }

  scv_smart_ptr<rw_task_if::write_t> write;
  for (int i=0; i<3; i++) {
    write->next();
    transactor->write( write->get_instance() );
    cout << "send data : " << write->data << endl;
  }

  scv_smart_ptr<int> data;
  scv_bag<int> distribution;
  distribution.push(1,40);
  distribution.push(2,60);
  data->set_mode(distribution);
  for (int i=0;i<3; i++) { data->next(); process(data); }
}

class design : public pipelined_bus_ports {
  list< sc_uint<8> > outstandingAddresses;
  list< bool > outstandingType;
  sc_uint<8>  memory[128];

public:
```

```
  SC_HAS_PROCESS(design);
  design(sc_module_name nm) : pipelined_bus_ports(nm) {
    for (int i=0; i<128; ++i) { memory[i] = i; }
    SC_THREAD(addr_phase);
    SC_THREAD(data_phase);
  }
  void addr_phase();
  void data_phase();
};

inline void design::addr_phase() {
  while (1) {
    while (addr_req.read() != 1) {
      wait(addr_req->value_changed_event());
    }
    sc_uint<8> _addr = bus_addr.read();
    bool _rw = rw.read();

    int cycle = rand() % 10 + 1;
    while (cycle-- > 0) {
      wait(clk->posedge_event());
    }

    addr_ack = 1;
    wait(clk->posedge_event());
    addr_ack = 0;

    outstandingAddresses.push_back(_addr);
    outstandingType.push_back(_rw);
    cout << "received request for memory address " << _addr << endl;
  }
}

inline void design::data_phase() {
  while (1) {
    while (outstandingAddresses.empty()) {
      wait(clk->posedge_event());
    }
    int cycle = rand() % 10 + 1;
    while (cycle-- > 0) {
      wait(clk->posedge_event());
    }
    if (outstandingType.front() == 0) {
      cout << "reading memory address " << outstandingAddresses.front()
         << " with value " << memory[outstandingAddresses.front()] << endl;
      bus_data = memory[outstandingAddresses.front()];
      data_rdy = 1;
      wait(clk->posedge_event());
      data_rdy = 0;

    } else {
      cout << "not implemented yet" << endl;
    }
    outstandingAddresses.pop_front();
    outstandingType.pop_front();
  }
}

int sc_main (int argc , char *argv[]) {
```

```
  scv_tr_db db("my_db");
  //  scv_tr::set_global_db(&db);

  // create signals
  sc_clock clk("clk",20,0.5,0,true);
  sc_signal< bool > rw;
  sc_signal< bool > addr_req;
  sc_signal< bool > addr_ack;
  sc_signal< sc_uint<8> > bus_addr;
  sc_signal< bool > data_rdy;
  sc_signal< sc_uint<8> > bus_data;

  // create modules/channels
  test t("t");
  rw_pipelined_transactor tr("tr");
  design duv("duv");

  // connect them up
  t.transactor(tr);

  tr.clk(clk);
  tr.rw(rw);
  tr.addr_req(addr_req);
  tr.addr_ack(addr_ack);
  tr.bus_addr(bus_addr);
  tr.data_rdy(data_rdy);
  tr.bus_data(bus_data);

  duv.clk(clk);
  duv.rw(rw);
  duv.addr_req(addr_req);
  duv.addr_ack(addr_ack);
  duv.bus_addr(bus_addr);
  duv.data_rdy(data_rdy);
  duv.bus_data(bus_data);

  // run the simulation
  sc_start(1000000);

  return 0;
}
```

Appendix D: Dynamic Concurrency

There is currently an example in the examples directory SystemC 2.0.1 reference implementation that implements a prototype of dynamic thread facility (see $SYSTEMC/examples/systemc/forkjoin). To use this facility, the *.cpp files in the *forkjoin* library need to be compiled and linked into user's SystemC simulation. Also the *sc_fork.h* file will need to be included by SystemC source files that use the facility.

The file *sc_fork.h* in the example directory defines *sc_spawn_method()* and *sc_spawn_function()*, which are dynamic spawning enhancements that are not part of the SystemC 2.0 standard, but rather implemented on top of it. Both functions actually use a pool of threads that are statically declared at the beginning of simulation.

Both functions return an *sc_join_handle* object that can be used to later synchronize the spawned method. The syntax of *sc_spawn_method()* and *sc_spawn_function()* are as follows:

```
sc_join_handle sc_spawn_method(&returnValue, &object, ClassName::MethodName, &arg1, &arg2);

sc_join_handle sc_spawn_function(&returnValue, FunctionName, &arg1, &arg2);
```

If the method being spawned does not return a value, "(void*)0" should be used in place of "&returnValue".

Each of the above methods is overloaded to take any number of arguments to supply to the spawned function between zero and four.

```
class MyClass{
        // …
        int myMethod(const char *arg1, unsigned int arg2);
        // …
};  // class MyClass


void mySysCProc()  // lives inside a  subclass of sc_module and is declared to be SC_THREAD(mySysCProc)
{
MyClass *pMyObject = new MyClass();
        while(true)
        {
                int retVal;
                const char arg1 = "Data";
                unsigned int arg2 = 7;

                sc_join_handle myHandle = sc_spawn_method(&retVal, pMyObject, MyClass::myMethod, &arg1,
&arg2);
        }
} // mySysCProc()
```

The *forkjoin* library uses static thread pool that needs to be declared before threads can be spawned. Ideally this should be done in *sc_main()*:

```
#include <systemc.h>
#include "sc_fork.h"
```

```
// sc_main() initializes simulation run, runs the simulation, and then cleans up before exit
int sc_main(int argc, char *argv[])
{
        // Create the global thread pool with a limit of 10 threads
        thread_pool::init(10);

        // Rest of sc_main() implementation goes here…

        // Return the thread pool to prevent a memory leak
        thread_pool::destroy();

        // Other sc_main() clean up code goes here

}  // sc_main()
```

Using the API provided by this example, a test generating concurrent activities can be implemented as shown in the following code. Three processes are spawned in the example below: Two read tasks that need to be synchronized (i.e. we need to wait until both tasks complete before moving on) and a write task that can run as long as it needs to (perhaps indefinitely):

```
#include <systemc.h>
#include "sc_fork.h"
class MyTest : public sc_module
{
        // …
        SC_HAS_PROCESS(MyTest);
        void testCase1()
        {
                // Simple concurrent tests
                rw_task_if::addr_t addr[ 3 ]; addr[ 0 ] = 0; addr [ 1 ] = 1; addr[2] = 2;
                rw_task_if::data_t data[ 3 ]; data[2] = 4;

                sc_join_handle readHandle1 = sc_spawn_method( &data[ 0 ], transactor[ 0 ], &rw_task_if::read,
        &addr[ 0 ]);
                sc_join_handle readHandle2 = sc_spawn_method( &data[ 1 ], transactor[ 0 ], &rw_task_if::read,
        &addr[ 1 ]);
                // The only reason sc_join_handle is needed is if we want to synchronize the task later.
                // Since in our example we don't need to synchronize the write, we can cast the return
                // value of the sc_spawn_method() call below to void
                (void)sc_spawn_method((void*)0, transactor[0], &rw_task_if::write, &addr[2], &data[2]);

                // Synchronize the two read tasks; the write task keep going
                sc_process_join(readHandle1);
                sc_process_join(readHandle2);
                cout << "received data : " << data[ 0 ] << ","  << data[ 1 ] << "\n";
```

```
    } // testCase1()
    // …

    MyTest(const sc_module_name name) : sc_module(name) // …
    {
        // …
        SC_THREAD(testCase1);
        //…
    } // MyTest()

}; // class MyTest
```

This code generates two read transactions in parallel, so that they exercise the pipeline. The *sc_spawn_method* function is similar to *create()* in PThreads, creating a new C++ thread and executing the method specified in the third argument for the object in the second argument. The first argument is a pointer to the object in which the return value is stored, and the last argument is the address of argument to the method. Note that *sc_spawn_method* can take from zero to four arguments to be passed to the corresponding method (in the example above, *&addr[2]* and *&data[2]* are the arguments to the *rw_task_if::write()* method).

The SystemC Verification Working Group has discussed a more comprehensive set of APIs to support dynamic threads, but because of the overlap with the current activities in the Language Working Group we have not reached a conclusion and therefore are not able to include a specification in this document. Long term, the Language Working Group needs to provide a truly dynamic thread spawning system. When they do, it is likely that the API will differ from the one shown here, but it will probably include similar features.

Appendix E: Debugging

Debugging is a crucial component in any verification effort. Because SystemC is a C++ library, a C++ debugger can be used for debugging. However, a C++ debugger does not understand the built-in concepts that are presented in SystemC 2.0 and the SystemC Verification Standard.

The SystemC 2.0 standard includes a base class called *sc_object*, which implements some common functionality that a custom tool for SystemC can use to manipulate objects in SystemC. For example, it enables assignment of attributes to SystemC objects. While we might use the same base class for the objects in the SystemC Verification Standard, in some simple objects such as a handle for a transaction, it might be unreasonable to have the overhead of maintaining the full capability of *sc_object*.

We also realized that one of the problems in debugging data objects with non-C++ built-in types in a traditional C++ debugger is to extract appropriate information from data internal to a data object. Therefore, the SystemC Verification Standard includes a lightweight abstract interface to facilitate this process. It is up to the actual class to implement this interface when it is appropriate.

Conceptually (and a possible strategy for SystemC 3.0), we are partitioning the existing sc_object base class into two classes, a lightweight abstract interface called sc_object_if and a base class called sc_object_base derived from sc_object_if.

The scv_object_if Abstract Interface	Description

virtual const char * get_name() const {
  return NULL;
}
	This method returns the instance name of the object.

virtual const char * kind() const {
  return NULL;
}
	This method returns a string that is unique to each class.

virtual void print (
  ostream& o,
  int details,
  int indent
) const {}
	This method prints the current values on the output stream. If the *details* argument is nonzero, the method displays more information, typically displaying substructures if any. A positive value specifies how many levels of detail to print. A negative value causes the method to print all available levels of detail. The *indent* argument specifies the number of spaces that prefix each line.

virtual void show(
  int details,
  int indent
) const { print(cout,-1,0); }
	This method prints the current values to the screen. This method is designed to be executed within a debugger, such as gdb. The default implementation is sufficient most of the time, but a derived class can override the default implementation if necessary.

static void set_debug_level(
  const char * facility,
  Int level = -1
);
	This static method turns on debugging messages for a specific facility.

virtual int get_debug() const { return 0; }	This method returns the debugging level for a specific class.
virtual void set_debug(int) {}	This method sets the debugging level for a specific class.

The derived classes of the scv_object_if abstract interface must implement two static methods called “int get_debug()” and “void set_debug(int)”, but the body of their implementation can be left empty. During our discussion in the VWG meetings, we have considered using data introspection to extract the appropriate information from a data object. However, it was brought up that the extraction would most likely be done in a debugger such as gdb, and using templated functions such as *scv_get_extensions()* would not be possible. The lightweight abstract interface can be executed in a debugger as a simple function call.

The first two methods provide a quick summary of what the object is. The *show()* and *print()* methods are used for displaying information about the object. The *show()* method is designed to be used within a debugger. For example, if *packet* is a variable in the thread being debugged, the following session shows how its value can be extracted without going through the internal data of the *packet* class.

```
gdb> call packet.show()
src : 0x1000
dest : 0x00FE
payload : 0xabcd
```

Typically, the print() methods (and similarly for show()) will follow this form:

```
virtual void print(ostream& o, int details, int indent) {
        // print basic information
        ...
        // print details
        if ( details != 0 ) {
                int newdetails = ( details > 0 ) ? details-1 : details;
                for ( each substructure s ) s.print(o,newdetails,indent+2);
        }
}
```

The static method and the last two methods in the previous table are designed to configure the library to print out debugging messages during simulation. This is useful in tracing the operation of a misbehaving test bench. The set_debug() method turns on debugging information for the specific class, not just the current object. The set_debug_level() method turns on tracing for a specific facility, such as threads, etc. Different vendors can design different sets of facility name strings for turning on debugging information for different kinds of functionality. You can specify the level of details to be generated by using the integer argument level. If it is -1, debug output for the related facility is turned off. If it is 0, all available debug output is turned on. Positive values control the verbosity of the debug messages (larger values produce more outputs).

The classes that derive from the common debugging interface are:

Classes that should derive from the common debugging interface
scv_extensions_if
scv_smart_ptr_if
scv_random
scv_expression
scv_constraint_base
scv_tr_db
scv_tr_stream

scv_tr_generator
scv_tr_handle
scv_report

Appendix F: Interface Introspection

During the VWG meetings, we have discussed extending the data introspection facility to include interface introspection [14]. While it seems to be useful for assisted transaction recording, we have not determine whether it should be included in the Verification Standard, nor have we worked out the details in the API. This appendix captures a short example as discussed in the email discussion. This facility is not part of the proposal; it is included here for reference and archive and to stimulate new idea. Because we have not discussed it in details or agree upon the API, if we decided to include interface introspection in a later proposal, it might take a different form.

Using the data introspection facility in this proposal, information about a data type such as sc_uint< N > is captured in scv_extensions< sc_uint < N > >. Similarly, the interface introspection extracts information about a C++ class with public methods. For example :

```
// An application specific interface
class my_interface {
public:
        virtual int myf1( char a, short b )=0;
        virtual long myf2( int a )=0;
};
// A class that implements the interface
class my_implementation : public my_interface {
        virtual int myf1( char a, short b ) { return 0; }
        virtual long myf2( int a ) { return 0; }
};
```

An extension of these interfaces can be captured in an interface introspection facility, using macro instantiation such as the following:

```
// Define a specialization for the interface
SCV_II( my_interface, 2 )
SCV_II_FUNCTION_2( 0, myf1, int, char, short )
SCV_II_FUNCTION_1( 1, myf2, long, int )
SCV_II_END( my_interface )
```

A different macro is needed for a method with different numbers of arguments, or without a return type. It is not clear whether this series of macro instantiations can be generalized to a form similar to the data introspection extension specification, using a class structure to group related macros.

When these macros are specified appropriately for the interface, an extension object can be generated for the interface. This extension object will implement an abstract interface from which details about a method can be extracted. For example, the following code takes a pointer to an extension object that corresponds to a specific method in an interface. It extracts the name of the return value, the name of the method, and name of the parameters to the method.

```
void print_function( const interface_introspection_if::function_description *d ) {
        cout << d->return_value->name() << " ";
        cout << d->name() << "( ";
        for ( int i=0; i<d->number_of_parameters; i++ ) {
                if ( i>0 )
                        cout << ", ";
                cout << (char *)d->parameters[i]->name() << " param" << i;
        }
        cout << " )";
}
```

The following code extracts the list of methods in a class:

```
void print_interface( const interface_introspection_if *ii ) {
        cout << "class " << ii->name() << " : public sc_interface { \n";
        for ( int i=0; i<ii->number_of_functions(); i++ ) {
                print_function( ii->function(i) );
                cout << ";\n";
        }
        cout << "\n};";
}
```

The following code shows how the interface extension object can be constructed from a specific object, where *get_ii()* returns an interface introspection extension of the argument, similar to *scv_get_extensions()* in the data introspection facility:

```
int main() {
        …
        // accessing information about an interface
        my_implementation m;
        const interface_introspection<my_interface> &ii = get_ii(&m);
        cout << "my_implementation has " << get_ii(&m).number_of_functions() << " functions.\n";
        cout << "\n\nmy_implementation is:\n";
        print_interface( &get_ii(&m) );
        cout << "\n";
        …
}
```

Appendix G: Assisted Transaction Recording

We have spent a significant amount of time discussing automatic or assisted transaction recording in our VWG meetings. This appendix summarizes some of the discussion.

G.1 Providing Observability and Controllability of Communication Through Watchable Channels

This section summarizes the discussion of watchable and watcher classes with respect to automatic transaction recording. The paragraphs in italics are direct quote from the document distributed by Forte [15]. The other paragraphs might contain personal opinions from individual members during the VWG discussions, so they might differ from the original document.

Goals : A fundamental requirement for a verification infrastructure is to provide observability and controllability of activity within and around the design. In SystemC 2.0 a central organizing principal is the representation of activity as communication between modules through channels. We believe that a primary goal of the SystemC verification standard must be to provide observability and controllability of communication through channels.

This observability and controllability will be used to provide many capabilities to the verification user including :

- *The ability to generate constrained random activity on a channel*
- *The ability to log activity on a channel for later analysis*
- *The ability to create verification entities that passively observe activity on a channel for purposes such as measurement of functional coverage, and validation of correct behavior.*

Secondary goals include:

- *Minimizing the amount of additional work that must be done by the end user to support verification*
- *Minimizing the extent to which the needs of verification intrude on the designer of a channel. This means minimizing the amount of modification that must be made to a channel to enable verification*
- *Permitting the decision of whether to log activity on a channel to be made as late as possible.*

We have developed a set of classes that demonstrate one way these goals may be achieved. [...]

The goals listed above describe the need of an assisted transaction recording style that simplifies the use of transaction recording in a channel. The VWG members believe it can be layered on top of the current manual transaction recording facility in the current proposal.

The description in the original document relies on a base class from which the corresponding channel should be derived from. The base class contains a general-purpose channel-oriented callback infrastructure. According to the document, the benefit is:

The benefit of building the logging capability on top of a general-purpose channel-oriented callback infrastructure, is that it simplifies the process of adding logging to a channel, as well as permitting a rich set of solutions to be developed in addition to logging. In this proposal the general callback infrastructure is implemented in "watcher" class, scv_watcher, and a "watchable" class, scv_watchable. Objects that derive from scv_watcher are able to register for callbacks from objects that derive from scv_watchable. The ability to derive functionality from a generic watcher extends beyond the basic logging capability to include such things as debuggers, functional watchers in the testbench and higher-level system modeling.

The watchable classes and watcher classes are defined as follows:

Watchable classes : SystemC provides standard mechanisms for one entity to anonymously learn of activity in another entity. These are the event, and the ability to have a process that is sensitive to changes of value of a set of signals. Sensitivity to signals is not available for non-signal channels. So it does not extend easily to the general case. Events are used for inter-thread synchronization in user code and in channel classes, but because there is no data associated with events, they don't provide a convenient basis on which to build a system that performs anonymous logging and watching of channel activity.

We will introduce the scv_watchable class as a base class that provides general-purpose registration and notification services. It is tailored to the requirements of the logging and watching of transaction activity on channels, but it is designed as a general-purpose class.

An scv_watchable object generates events that are delivered asynchronously to all of the clients (scv_watchers) who are registered to receive them. Events are delivered to clients by means of an immediate function call rather than one that is deferred until the end of the cycle, as it is the case with an sc_event. This is important because multiple important transaction level events can occur in one simulation delta. This mechanism ensures that clients cannot miss these events.

An scv_watchable allows watching clients to anonymously register to receive a subset of the possible event types. Each scv_watcher-derived class is responsible for generating the events, but is not responsible for keeping track of which clients are registered. The set of possible event types is fixed. It is designed to serve a wide variety of channel types by observing that the basic function of any channel is for information to be provided by one entity and received by others. […]

Watcher classes : If we've got watchables, we need watchers. The scv_watcher class provides the basis for all classes that are clients of scv_watchable. Only scv_watcher classes may register to receive events from scv_watchable classes. […]

The example of logging from the original document is shown below:

```
class txconsumer : public sc_module, public xf1, public scv_watchable<xf1*> {
public:
        sc_port<xf1, 0> m_input_port;
        txconsumer( sc_module_name name ) : sc_module (name), m_input_port(*this) {}
        ~txconsumer() {}

        // implementation of the receive function
        void receive( int a, int * d)  {
```

```
                notify_read_start();
                cerr << "In txconsumer::receive at " <<
                        sc_get_curr_simcontext()->time_stamp() << "\n";
                wait(10, SC_NS);
                NOTIFY_READ_END_IF(xf1)->receive(a,d);
        }
        // implementation of the transmit function {
                notify read_start();
                cerr << "In txconsumer::transmit at " <<
                        sc_get_curr_simcontext()->time_stamp() << "\n";
                wait( 10, SC_NS);
                NOTIFY_READ_END_IF(xf1)->transmit( a, d );
        }
        [ ...]
};
```

The watcher and watchable classes must rely on extending the data introspection mechanism in this proposal to describe the functions of an interface. However, while the document claims that *this combined with a generic callback* infrastructure will allow almost transparent, anonymous, automatic logging, the example shown in the document relies on intrusive addition of notify_read_start() and NOTIFY_READ_END_IF() in the channel implementation, which is similar to *begin_transaction()* and *end_transaction()* in the current manual transaction recording specification. However, it does simplify the use model by eliminating the need to explicitly instantiate the transaction streams and the transaction generator (at the expenses of requiring the interface introspection declaration for the channel interface)

Even with full support of interface introspection, it is unlikely to be able to eliminate the manual addition of a call to specify the start time and attribute of a transaction. It was brought up in the VWG meeting that sometime a channel has to synchronize multiple tasks, and the time at which a channel method is called might not correspond to the conceptual start of a transaction; so the *begin_transaction()* method may need to be called after the channel method gains access to a certain resource. Furthermore, sometime it is not appropriate to record some of the arguments to the channel methods as attributes of the corresponding transactions.

The communication between the watchable classes and the watcher classes seems to lead to a generic facility that supports more than just transaction recording. For example, it might provide the mechanism for a transactor to communicate to a golden model. The VWG meetings have not discussed this possibility independent of transaction recording.

The original document also describes an scv_msg class, which has not been discussed in the VWG meetings.

G.2 Examples of Different Styles of Assisted Transaction Recording

During the discussion of automatic and assisted transaction recording, several examples illustrating different styles of automation were distributed. This section summarizes five examples that were described in a document from Cadence [12]. These styles of assisted transaction recording can be implemented as a layer on top of the current manual transaction recording API:

a) Hiding transaction stream and transaction generator in a transactor (similar to the watchable classes described in D.1)
b) A non-intrusive class wrapper in the connection
c) A non-intrusive method wrapper in a transactor
d) A non-intrusive method wrapper in a test
e) Simple macros in a transactor method implementation

Each of these examples has its pros and cons. Some of them are better thought out than others. Example A is very similar to the watchable class idea in the previous section. Example B is a completely non-intrusive solution that works on existing transactor IPs. The remaining ones are mostly from earlier investigation and it seems that in practice, the styles in A and B are more useful and addresses different needs that may arise in different situation.

A) hiding a transaction stream and transaction generator in a transactor

```
#include <systemc.h>
#include <vwg.h>
// a thin assisted transaction recording layer
class scv_watchable_interface : public sc_interface {
        scv_tr_stream stream;
public:
        template<typename T>
        scv_tr_handle begin_transaction(const string& transaction_name, const T& begin_attr) {
                scv_tr_generator<T> gen(transaction_name,stream);
                // the optional end-attribute is not used.
              return gen.begin_transaction(begin_attr);
        }
        template<typename T>
        void end_transaction(scv_tr_handle& h, const T& end_attr) {
                h.get_tr_generator_base().end_transaction(h,end_attr);
        }
        template<typename T>
        void record_attribute(scv_tr_handle& h, const sc_string& name, const T& attribute) {
```

```
                h.get_tr_generator_base().record_attribute(h,name,attribute);
        }
};

// how a user will specify what transaction to record and what attributes to record.
class my_transactor_if : public scv_watchable_interface {
public:
        virtual int read(int addr) = 0;
};
class my_transactor : public my_transactor_if {
public:
        virtual int read(int addr) {
                scv_tr_handle h = begin_transaction("read",addr);
                ...
                if (addr < 0xFF) {
                        record_attribute(h, "special_attr",3);
                }
                ...
                end_transaction(h,data);
                return data; }
};
```

B) non-intrusive class wrapper in the connection

```
#include <systemc.h>
#include <vwg.h>
// ----------------------------------------
// a thin assisted transaction recording layer
// - implementation is too long to put into this document. But it has been done.
// - an anchor to determine the right begin time has also been done to handle
//   concurrent calls to the transactor methods.
// ----------------------------------------
template<typename T> class scv_atr_base { ... }
template<typename T> class scv_atr_extensions : public scv_atr_base<T> {}
template<typename T> scv_atr_extensions<T> scv_get_atr(T& a) { ... }

// ----------------------------------------
```

```
// how a user will specify what transaction to record and what attributes to record.
// ----------------------------------------
class my_transactor_if : virtual public sc_interface {
public:
        virtual void write(int addr, int data) = 0;
};
 template<>
class scv_atr_extensions<my_transactor_if> : public scv_atr_base<my_transactor_if> {
        my_transactor_if * _instance;
public:
        virtual void write(int addr, int data) {
                // deferrable_begin_transaction() is similar to
                // begin_transaction, except that it examines whether the wrapper has been
                // configure to begin the transaction right away or after certain resource
                // is obtained. (A later example shows the use of “anchor” to achieve delayed
                // begin of the transaction
                scv_tr_handle h = deferrable_begin_transaction(addr);
                _instance->write(addr,data);
                end_transaction(h,data);
        }
};
class my_transactor : public my_transactor_if {
public:
        virtual void write(int addr, int data) { ... }
};
class my_test : ... {
public:
        sc_port<my_transactor_if> port_1;
};
int sc_main(...) {
        my_test t("t"); // a test which knows nothing about transaction recording
        my_transactor tr("tr"); // a transaction which knows nothing about transaction recording
        t.port_1(scv_get_atr(tr));
         scv_get_atr gets a wrapper around “tr” to do transaction recording
};
```

This approach is a non-intrusive approach that works with existing tests and transactors. The change is in the connection. In sc_main(), when the test is connected to the transactor, the template function scv_get_atr() is called to instantiate the transaction-recording wrapper around the transactor.

This can be extended to handle transactor methods that must obtain a resource before starting a transaction, through the use of an anchor:

```
template<> class scv_atr_extensions<my_pipelined_transactor>
: public scv_atr_extensions<my_transactor_if> {
public:
        virtual scv_begin_anchor get_begin_anchor(const sc_string& task) {
                return ((my_pipelined_transactor*) get_instance())->addr_mutex;
        }
};
```

C) non-intrusive method wrapper in a transactor

```
#include <systemc.h>
#include <vwg.h>
// ----------------------------------------
// a thin layer for assisted transaction recording
// ----------------------------------------
// and other overload version for different method
// signatures (1 arg, 2arg, ..., with or without return type,
// etc.)
template<typename transactor_t, typename T1, typename T2>
void scv_atr(scv_tr_stream& s,
          const string& transaction_name, transactor_t& transactor,
          void (*transactor_t::method)(T1,T2), T1 arg1, T2 arg2) {
        scv_tr_generator<T1,T2> gen(transaction_name, s);
        scv_tr_handle h = gen.begin_transaction(begin_attr);
        transactor.(*method)(arg1,arg2);
        gen.end_transaction(h,end_attr);
};
// ----------------------------------------
// how a user will specify what transaction to record and what attributes to record.
// ----------------------------------------
class my_transactor_if : public sc_interface {
public:
        virtual void write(int addr, int data) = 0;
```

```
};
class my_transactor : public my_transactor_if {
        scv_tr_stream my_stream;
        fifo_mutex addr_phase;
        fifo_mutex data_phase;
public:
        SC_CTOR(my_transactor) : my_stream("my_transactor") {}
        void write(int addr, int data) {
                addr_phase.lock(); // determine exact begin time
                // the only line for transaction recording
                 scv_atr(my_stream,"write", this,&my_transactor::write_core,addr,data);
        }
private:
        void write_core(int addr, int data) {
                // perform actual protocol manipulation
                // ...
                addr_phase.unlock();
                data_phase.lock();
                // ...
                data_phase.unlock();
        }
};
```

D) non-intrusive method wrapper in a test

```
#include <systemc.h>
#include <vwg.h>
// ----------------------------------------
// a thin assisted transaction recording layer
// ----------------------------------------
// and other overload version for different method
// signatures (1 arg, 2arg, ..., with or without return type,
// etc.)
template<typename transactor_t, typename T1, typename T2>
void scv_atr(scv_tr_stream& s,
            const string& transaction_name, transactor_t& transacor,
            void (*transactor_t::method)(T1,T2), T1 arg1, T2 arg2) {
```

```
        scv_tr_generator<T1,T2> gen(transaction_name, s);
        scv_tr_handle h = gen.begin_transaction(begin_attr);
        transactor.(*method)(arg1,arg2);
        gen.end_transaction(h,end_attr);
};
// ----------------------------------------
// how a user will specify what transaction to record and what attributes to record.
// ----------------------------------------
class my_transactor_if : public sc_interface {
public:
        virtual void write(int addr, int data) = 0;
};
class my_test : public sc_module {
        scv_tr_stream stream;
public:
        sc_port<my_transactor_if> port_1;
        SC_CTOR(my_test) : stream("my_test") { SC_THREAD(main); }
        void main() {
                int addr, data;
                // instead of calling port_1->write(addr,data),
                scv_atr(stream,"write", port_1[0], &my_transactor_if::write,addr,data);
        }
};
```

E) simple macros in a transactor method implementation

```
#include <systemc.h>
#include <vwg.h>
// ----------------------------------------
// a thin layer for assisted transaction recording
// ----------------------------------------
#define SCV_TRANSACTION_DATA \
        scv_tr_stream my_stream;
#define SCV_BEGIN_TRANSACTION(transaction_name,begin_attr_type, begin_attr) \
        scv_tr_generator<begin_attr_type> gen(# transaction_name, my_stream); \
        scv_tr_handle h = gen.begin_transaction(begin_attr);
#define SCV_RECORD_ATTRIBUTE(attr) \
```

```
        gen.record_attribute(h,# attr, attr);
#define SCV_END_TRANSACTION(end_attr) \
        gen.record_attribute(h, “end-attribute”, end_attr); \
        gen.end_transaction(h);

// ----------------------------------------
// how a user will specify what transaction to record and what attributes to record.
// ----------------------------------------
class my_transactor_if : public sc_interface {
public:
        virtual void write(int addr, int data) = 0;
};
class my_transactor : public my_transactor_if {
        SCV_TRANSACTION_DATA;
        fifo_mutex addr_phase;
        fifo_mutex data_phase;
public:
        SC_CTOR(my_transactor) {}
        void write(int addr, int data) {
                addr_phase.lock();
                SCV_BEGIN_TRANSACTION("write",int,addr);
                // ...
                addr_phase.unlock();
                data_phase.lock();
                // ...
                data_phase.unlock();
                SCV_END_TRANSACTION(data);
        }
};
```