Chapter 15
SCV

SystemC Verification Library

15.1 Introduction

In the course of this book, we have covered the SystemC language and its many uses. We have explored the SystemC constructs that let us model hardware easily, including clocks, hardware data types, concurrency constructs, threads, etc. With this knowledge, you are equipped with the ability to construct a sophisticated system model and most of the features required to develop a robust testbench. There is an additional library, the SystemC Verification Library (SCV), which provides much of the features required to implement a robust reusable testbench without having to develop these on your own. This library is described in detail in the downloadable PDF document "SystemC Verification Standard Specification [Version 1.0b]" from <www.systemc.org>. Please note that the document references some aspects of SystemC that have changed from version 2.0.1 upon which it was originally based.

The SCV library includes many add-on features to SystemC including data introspection, extended data types, random data types, transaction monitoring, and transaction recording.

It is beyond the scope of this book to cover specific verification methodologies. We will, however, lightly touch on the topic of developing transaction-based verification and how this allows for higher levels of abstraction test cases, promotes reusable verification IP, and shortens the overall verification cycle.

15.2 Data Introspection

Data introspection is one of the key features of SCV. Introspection allows for variable manipulation without compile-time knowledge of the variable type. SCV implements this feature using partial template specialization. From the user point of view, SCV provides a standard abstract interface, `scv_extensions_if`, through which the user can access and manipulate the desired data.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_15, © Springer Science+Business Media, LLC 2010

The **scv_extensions_if** class is an abstract interface. This means that the class does not implement any methods directly. Instead, this interface contains five components, which in turn provide the methods to manipulate the data. The component interfaces are:

- **scv_extension_util_if**
- **scv_extension_type_if**
- **scv_extension_rw_if**
- **scv_extension_rand_if**
- **scv_extension_callbacks_if**

These components can be further categorized (Fig. 15.1) into two groups: static and dynamic extensions. Static extensions do not require additional data to be associated with the extended data type. The **type** and **rw** interfaces provide the methods for static extensions by letting the user extract data type information and to read and write to the data object. On the other hand, dynamic extensions require additional data to be associated with the extended data object for the purpose of storing constraints for randomization, and for storing callback function pointers.

We will present each of the **sc_extension** interface components, but will not

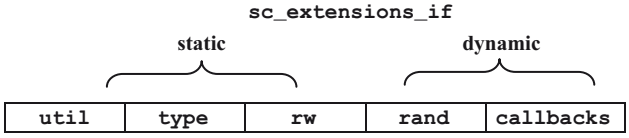

Fig. 15.1 **sc_extensions_if** components

give an exhaustive coverage of all the methods, which are covered in detail in the SCV specification.

15.2.1 Components for scv_extension Interface

As mentioned above, there are five components in the **scv_extension** interface. This section will give a quick overview of the five components, and then we will develop some of the details in later sections.

The **util** component provides the utility methods to obtain information about the data, including the name of the data object, whether dynamic extensions are supported, and whether the data object is a valid extension. SCV supports both C/C++ and SystemC built-in data types. In the following example (Figs. 15.2 & 15.3), the code tests whether the user-defined type has valid extensions.

In the example (Figs. 15.1 & 15.2), the test for valid extensions will fail unless the Packet **struct** is extended. We will cover user-defined data type extensions in the next section.

```
//File: Packet.h
struct Packet { // user-defined type
  enum type { SIMPLE, EXTENDED };
  type        mode;
sc_uint<16> address;
sc_uint<32> data;
};
ostream&operator<<(ostream& os, const Packet& p) {
  os<< "{"
    <<((p.mode==p.SIMPLE)?"SIMPLE":"EXTENDED")
    <<hex
    << ":addr ess=0x"
    <<p.address
    <<",data=0x" << p.data
    <<"}";
return os
}
```

Fig. 15.2 User-defined type

```
#include "Packet.h"
if (scv_get_extensions(Packet).has_valid_extensions())
{
  // Tests to see if
  // Packet has a valid
  // extension
   ….
}
```

Fig. 15.3 Test for extensions on user-defined type

Another **sc_extension** component, **type**, provides methods to extract data type information, including type name and bit width. SCV provides the data types shown in Fig. 15.4:

```
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
}
```

Fig. 15.4 Supported data types for **sc_extensions_if**

The next component of the **sc_extension** interface is **rw,** which provides methods to read and write to the data object. Using this interface might seem cumbersome, when all you want is to obtain or set the data, but it lets your code be written more generically and thus, it is more maintainable. The example in Fig. 15.5 shows how the **type** and **rw** components might be used.

```
void print_data(scv_extensions_if* data_ptr) {
  switch(data_ptr->get_type()) {
    case scv_extensions_if::BOOLEAN:
        cout<<data_ptr->get_type_name()
            <<"_value is: "
            <<data_ptr ->get_bool();
…
  }//end switch
}// end print_data
```

Fig. 15.5 Using **type** and **rw** components

The last two components are **rand** and **callback,** which provide the methods to perform random operations on the data and register for callbacks. Recall that **rand** and **callback** are dynamic components that require additional data associated with the object. These components will add some overhead to your code execution.

As the name implies, the **rand** component provides the interface to generate a random data stream for your extended object. Used with the SCV constraint class and weighted randomization techniques, this interface becomes a powerful tool for building a full-featured testbench. We will cover this interface in more detail later in the chapter.

As for the **callback** interface, the methods provided let the user register for callback if the value of the extended object has either changed or been deleted. You will need to write your own callback function and implement the actions required upon such a change. We will present more on callbacks later in this chapter.

15.2.2 Built-In scv_extensions

As we mentioned before, SCV provides extensions for all the built-in C/C++ and SystemC data types.In addition to the extension methods discussed so far, the template extensions also include operators that let you manipulate the extended objects as you would a built-in type. For example, you can use +=, <<=, *=, etc. on the extended data objects to perform simple operations. The template extensions also provide **read()** and **write()** functions to access the data.

15.2.3 User-Defined Extensions

In some cases, you may want data introspection on your user-defined type. To extend the user data object, you must implement the partial template specialization of the **scv_extensions**. Let us demonstrate in Fig. 15.6 by using the example shown earlier.

```
//File: Packet_ext.h
#include "Packet.h"

// Extend above user-defined type as follows:
SCV_EXTENSIONS(Packet) {
public:
  scv_extensions<sc_uint<16>> address;
  scv_extensions<sc_uint<32>> data;
  SCV_EXTENSIONS_CTOR(Packet) {
    SCV_FIELD(address);
    SCV_FIELD(data);
  }
bool has_valid_extensions() { return true; }
};
```

Fig. 15.6 Extending user-defined types

Fortunately, Cadence Design provided an open source Perl script, **tb_wizard_ext**, that takes your user-defined type as input and generates the appropriate partial template specialization. We have included the script in the examples that go with this book.

15.3 scv_smart_ptr Template

In most cases, it is easier to use a template provided by SCV to handle the **scv_extensions** pointer. The **scv_smart_ptr** class acts just like a C++ pointer to the **scv_extensions** object. Under the hood, the template incorporates both **scv_extensions** and **scv_shared_ptr** objects:

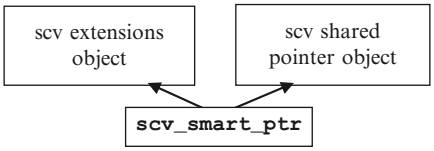

Fig. 15.7 **scv_smart_ptr** template

The **scv_shared_ptr** lets multiple threads share data objects by implementing the necessary memory management. To use this template, instantiate the object with the appropriate data type.

```
scv_smart_ptr<Packet> pPkt;

pPkt->address = 0;
pPkt->data = 0;
```

Fig. 15.8 Using **scv_smart_ptr** on user-defined type

An important feature of the smart pointer is that it implements both the static and dynamic components of the **scv_extensions_if**. This feature means that you can use the smart pointers to implement randomization and register callbacks. We will revisit the smart pointers later in the chapter.

15.4 Randomization

Traditionally, hardware designs have been verified using directed testing methodology. In recent years, there has been more focus on using random testing methodology to achieve wider test coverage. In many situations, verification engineers implement a combination of directed and random tests to achieve their testing goals.

In directed testing, one creates certain scenarios to test each feature of the design. If you want to vary sequences of reads/writes, packet size, address, block sizes, or data values, you must do so manually. When one creates tests using randomization, the stimulus is created using constrained randomization. The expected results can be calculated with a reference model, possibly your system model.

In a unit test environment, it may be desirable to use directed testing and make sure your design block is behaving correctly as you walk through all the scenarios. Some others advocate doing directed unit testing by way of a fully constrained (constrained to one value) random testbench.

In system-level verification, this approach becomes tedious and time consuming. It becomes difficult for the human to consider all possible combinations of test vectors that are required to test all features in normal and boundary conditions. Using random testing methodology, you can easily create many different scenarios for the test stimulus.

SCV provides the infrastructure for you to create basic randomization, constrained randomization, and weighted randomization tests. We will cover the basics of these random concepts and provide some demonstration of how to use the SCV library using the standard API. We will also cover two important **sc_interface_if** templates used for randomization: **scv_smart_ptr**, and **scv_bag**.

15.4.1 Global Configuration

SCV provides a random class, which provides the basis for the random stream generator. Before generating a random number for your data variable, you may

want to set some global configuration parameters to define the desired random distribution or seed value. By default, SCV uses the **jrand48()** algorithm from the standard C library, but you may choose a different algorithm or customize your own by using the **set_default_algorithm** method. The following is a list of possible algorithms to choose from:

```
enum  value_generation_algorithm {
   RAND,
   RAND32,
   RAND48,     // default
   CUSTOM      // requires further configuration setup
}
```

Fig. 15.9 List of randomization algorithms

In addition to the algorithm, you can also specify the method by which the random values are generated. By default, the mode is set to **RANDOM**, which enables the data to be generated across all possible values for that data. For example, randomizing an **sc_uint**<8> with this mode allows 2^8 possible values to be generated in the data stream. On the other hand, you can specify regions of values to keep in or out from the generated data by specifying the **DISTRIBUTION** mode. Specify the mode by calling the **set_mode** method and selecting one of the four modes described next:

```
enum mode_t {
      RANDOM,  // default
      SCAN,
      RANDOM_AVOID_DUPLICATE,
      DISTRIBUTION
}
```

Fig. 15.10 Supported modes using the **set_mode** method

- **RANDOM**—uniform distribution across all legal values
- **SCAN**—maintains the history; starts with the smallest legal value
- **RANDOM_AVOID_DUPLICATE**—maintains the history; avoids duplicated values until all legal values have been generated, at which point it resets the history
- **DISTRIBUTION**—uses specified constraints to affect generated values

Another important configuration is the seed value. Through the **scv_random::set_global_seed** method, the user can change the default seed at the start of simulation. If unspecified, SCV defaults the seed to 1. You may also reset the seed during the simulation by calling **scv_random::set_current_seed**, which will set the seed for the next generated random number. While most

users will set the global seed, you may want to consider generating a unique seed for each process thread. This implementation allows for better repeatability that is independent of your SystemC simulator, and therefore independent of any scheduler differences.

Lastly, you can further control your randomization by enabling or disabling the random feature on a per-data-object basis. If you have a composite type, you can also choose to turn off randomization for just one of the data members. The methods to do this are **enable_randomization()** and **disable_randomization()**.

15.4.2 Basic Randomization

When you have decided to use randomization in your testbench, the most basic feature you need is to generate random sequences for your data object. If you have extended your object using **scv_extensions**, you can generate a random sequence by calling the **next** method in the **scv_extensions_if** random component interface as shown in Fig. 15.11.

```
#include "Packet.h"
scv_smart_ptr<Packet> pPkt;
pPkt->next();//creates random values for address &
data
```

Fig. 15.11 Generating random stream on user-defined type

In the example, **next()** returns one of 2^{32} possible integer values for address and data. In a composite type such as Packet above, you can opt to disable selective data members for randomization. For instance, the code in Fig. 15.12 allows only the data to be randomized:

```
scv_smart_ptr<Packet> pPkt;
pPkt->address.disable_randomization();
pPkt->next();
```

Fig. 15.12 Disable randomization

Alternatively, you can call **next()** on just the data member you want to randomize as shown in Fig. 15.13.

```
scv_smart_ptr<Packet> pPkt;
pPkt->address.next();
```

Fig. 15.13 Selective randomization

If you need to constrain the randomization to a range of values, or weigh small packets to occur more often, you will need to use constrained random or weighted random methods as described in the following sections.

15.4.3 Constrained Randomization

Constrained randomization restricts the random generated data stream by letting you specify allowable regions or values. SCV provides a constraint class, **scv_constraint_base**, with convenient macros to specify each constraint rule. The following is a list of the macros:

- **SCV_CONSTRAINT_CTOR**—constructor for the constraint class
- **SCV_CONSTRAINT**—defines a hard constraint
- **SCV_SOFT_CONSTRAINT**—defines a soft constraint
- **SCV_BASE_CONSTRAINT**—defines a base constraint

The constraint class uses **scv_smart_ptr** to implement the randomized data. To define a constraint, first create a constraint class to define rule(s) for constraining your data type. This class must inherit from the base constraint class, **scv_constraint_base**, and implement the constructor.

```
class Pkt_constraint
      :virtual public scv_constraint_base {
public:
scv_smart_ptr<Packet> pPkt;
SCV_CONSTRAINT_CTOR(Pkt_constraint) {
    // define constraints
SCV_CONSTRAINT(
      (pPkt->address() != 0x00000000) &&
      (pPkt->address()  < 0x00000800)
    );
SCV_CONSTRAINT(pPkt->data() >= 0x00001000);
}
};
```

Fig. 15.14 Constraint class

An important point to note in the constraint macros concerns the use of the parenthesis operator to construct Lamda[1] expressions. Notice that every reference to a data member is followed by an empty parenthesis pair (i.e., "()"). This syntax is needed because the class creates an internal representation of the constraint equations, which are used with a solver. Detailed explanation of how this is accomplished goes beyond the scope of this book, but suffice it to say that these parenthesis pairs are required. It should also be noted this it is a common error to forget to supply these pairs when coding.

[1] The inquisitive reader may choose to research Lamda calculus to understand the reasons for this. Basically, the SCV needs to store equations to allow it to do constraint solving.

To use the constraint, instantiate as you would any class as illustrated in Fig. 15.15:

```
Pkt_constraint cPkt;

cPkt.next();
cout<< "data = " << cPkt.pPkt->data <<endl;
```

Fig. 15.15 Using constraint class

By default, **SCV_CONSTRAINT** specifies hard constraints. Using hard constraints means that if SCV cannot find a legal value for this constraint, it will generate an error and ignore the constraint. You can specify a soft constraint, **SCV_SOFT_CONSTRAINT**, and SCV will generate only a warning if it cannot find a legal value.

Since constraints are captured in a constraint class, you can build hierarchical constraints using inheritance. Start with a basic constraint class defining fundamental constraints and add in more complicated constraints using hierarchy to define more specific or complicated constraints.

15.4.4 Weighted Randomization

While constraints are used to define legal values of the random data stream, weighted randomizations are used to define frequency of certain generated values. SCV provides some methods to define simple distribution through **scv_extensions**:

- **keep_only**—define a value or range of values to include in distribution
- **keep_out**—define a value or range of values to exclude in distribution

When you call these methods, SCV automatically sets the randomization mode to **DISTRIBUTION** and disregards any previous **set_mode** distributions for that particular data object. Likewise, any previous constraints defined for the data object are also disregarded. Note that you can define multiple **keep_only** and **keep_out** ranges for the data object, and the result is a cumulative effect of the defined ranges.

```
scv_smart_ptr<Packet> pPkt;
pPkt->address.keep_only(1,9999);
pPkt->data.keep_out(0);
pPkt->data.keep_out(10000U,(1U<<30));
```

Fig. 15.16 Using **keep_only** and **keep_out** to define random distribution

In some cases, you may want to define more complicated distribution rules for your data. In this case, SCV provides a templated class, **scv_bag**, to define relative weight of particular values that is illustrated in Fig. 15.17.

```
// define a bag
scv_bag<int> intBag;

intBag.add(0, 25); //add 25 objects of value 0 to bag

intBag.add(1, 25); //add 25 objects of value 1 to bag

intBag.add(2, 50); //add 50 objects of value 2 to bag


scv_smart_ptr<int> smart_int;
smart_int->set_mode(intBag);   //set smart_int
                               //distribution
```

Fig. 15.17 Using **scv_bag** to define random distribution

In dealing with multiple smart pointer variables within a constraint, you can specify distributions for each data member by using the **set_mode** method as shown in Fig. 15.18.

```
class Pkt_constraint
: virtual public scv_constraint_base
{
public:
 scv_smart_ptr<sc_uint<16>> address;
 scv_smart_ptr<sc_uint<32>> data;
 SCV_CONSTRAINT_CTOR(Pkt_constraint) {
    // define constraints
SCV_CONSTRAINT(
      (address() != 0x00000000) &&
      (address() < 0x00001000));
SCV_CONSTRAINT(data() >= 0x1000);
  }
};

void test() {

  typedef pair <sc_uint<32>, sc_uint<32>> data_range;
  scv_bag<data_range> data_dist;
  //set range distribution for data
  //data range (0x1000, 0xffff) occurs 30%
  //data range (0x10000, 0x20000) occurs 70%)
  data_dist.add(data_range(0x1000, 0xffff),30);
  data_dist.add(data_range(0x10000, 0x20000), 70);

  Pkt_constraint cPkt;
  cPkt.next();          //generate addr and data using
                        //constraints
  cPkt.data->set_mode(data_dist);
  cPkt.next();          //generate addr using
                        //constraints and generate
                        //data using 'data_dist'
                        //distribution
}
```

Fig. 15.18 Changing randomization modes

15.5 Callbacks

Callbacks present a powerful mechanism for monitoring variables (Fig 5.19). For that purpose, the SCV provides two main methods, **register_cb** and **remove_cb**. Once a function has been registered as a callback, it will be called anytime the referenced object changes. It should be obvious that if abused, this usage can result in a lot of overhead for the simulation. Therefore, you should use caution when selecting which variables to use callbacks on.

Let's look at a simple example in Fig. 15.20 of how to use a callback. For this example, we will use the previously used Packet type.

This example illustrates the concept that more than one callback may be registered on an object.

```
enum callback_reason {
  VALUE_CHANGE,
  DELETE
}
// Register a simple callback function
callback_h register_cb (
void (*f)(
scv_extensions_if& OBJ,
callback_reason    REASON
)
);
// Template method registers a callback function
// with an extra argument in a type-safe manner
template<typename T>
callback_h register_cb(
  void (*f)(
    scv_extensions_if& OBJECT,
    callback_reason REASON,
    T ARG)
  , T arg);

// Remove existing callback
virtual void remove_cb(callback_h HANDLE);
```

Fig. 15.19 Interface for callbacks

```
#include "Packet.h"

static unsigned changes;
// A function to monitor changes on a Packet
void Packet_cbA(
scv_extensions_if& obj,
scv_extensions_if::callback_reason reason
) {
if (reason == scv_extensions_if::VALUE_CHANGE)  {
cout<< "Packet " << obj.get_name()
<< " value change to " << obj.get_unsigned()
<<endl;
  } else {
cout<< "Packet " << obj.get_name()
<< " deleted." <<endl;
  }
}
void Packet_cbB(
scv_extensions_if& obj,
scv_extensions_if::callback_reason reason
) {
  if (reason == scv_extensions_if::VALUE_CHANGE)  {
   changes++;
  } else {
cout<< changes << " distinct values"
<<endl;
  }
}
scv_smart_ptr<Packet> pPkt1("pPkt1"),pPkt2("pPkt2");
scv_extensions_if::callback_h
  h1A(pPkt1->register_cb(Packet_cbA)),
h1B(pPkt1->register_cb(Packet_cbB)),
  h2A(pPkt2->register_cb(Packet_cbA));

for (int i=0; i!=10; ++i) {
  pPkt1->next(); pPkt2->next();
}
pPkt1->remove_cb(h1A);
```

Fig. 15.20 Example callback

15.6 Sparse Arrays

Almost a seemingly unrelated topic, a model for a sparse array is included in the SCV, but it is not completely unrelated. Memories and large memories are a part of almost every electronic system today. When simulating memories in a system, it is not possible to simulate a 4 GB memory while running on a 2 GB simulation computer without some compromises.

Fortunately, most of the time, simulations only use a tiny fraction of a large memory. For that reason, it makes sense to model memories as sparsely populated. Although, one could use a standard STL **map<>** container for this purpose; there are several useful extensions that make the **scv_sparse_array** a better choice. For one, reading an

unwritten location returns a default value. Another aid is the definition of memory bounds (i.e., upper and lower limits for the address). Fig. 15.21 is the constructor syntax.

```
scv_sparse_array<T1,T2> NAME(
const char * name,
const T2& default_value,
const T1& indexLB = 0,
const T1& indexUB = INT_MAX
);
```

Fig. 15.21 Creating a sparse array

The first typename, T1, designates the type of the index for the sparse array. The second typename, T2, designates the data value types. Accessing the memory is straightforward. Here is an example:

```
scv_sparse_array<unsigned,short> mem("mem",0,0,1e6);
scv_smart_ptr<unsigned> a_ptr;
scv_smart_ptr<short> d_ptr;
a_ptr->keep_only(0,1e6);
d_ptr->keep_out(0);
for (unsigned count=0; count!=30; ++count) {
  a_ptr->next(); d_ptr->next();
mem[*a_ptr] = *d_ptr;
}
for (unsigned count=0; count!=30; ++count) {
  a_ptr->next();
  *d_ptr = mem[*a_ptr];
cout<< *d_ptr <<endl;
}
```

Fig. 15.22 Sparse array example

One thought that comes to mind when modeling a system that may use consistently sized chunks of memory (e.g., a 256 or 1024 block of data), suggests deriving a custom sparse array that contains blocks of data (e.g., a **vector**<*T*>) sized to contain the data. Using a custom sparse array may prove more efficient when treating small groups of locations repeatedly.

15.7 Transaction Sequences

In many systems today, test teams are faced with the obstacle of verifying complex designs. In some cases, the testbench itself requires a great deal of work, including code partitioning, design for reusability, and flexibility. In many cases, you can use the concept of transaction-based testing to help achieve these goals. Throughout the book, you have been exposed to various SystemC constructs that let you design a transaction-based testbench.

A transaction is a set of activities defined by a start and finish and lasting a certain duration. For example, a read or write to memory is considered a transaction, including arbitration and transfer of data.

15.8 Transaction Recording

The SCV 1.0 library provides a set of APIs that lets the user record transactions. According to the OSCI documentation, this set of APIs is not an official part of the standard yet; however, many companies are using the interface. So, it is unlikely to go away. Furthermore, several commercial EDA tools are available to help visualize the results.

The APIs are categorized into three classes:

- **scv_tr_db**—transaction database containing a collection of transaction streams
- **scv_tr_stream**—transaction stream containing a collection of transactions
- **scv_tr_generator**—transaction generator for a specific transaction type

In a given simulation, you can instantiate one or more collections of transaction streams. This instantiation is usually done in **sc_main**. Then, within your modules, instantiate the **scv_tr_stream** and **scv_tr_generator** objects. A simple data recording example follows is shown in Fig. 15.23.

```
// note, scv_tr_db instantiated in sc_main.

class simple_transactor : public simple_ports {
scv_tr_stream read_stream;
scv_tr_generator<sc_uint<8>, sc_uint<8>> read_tr;
SC_CTOR(simple_transactor)
    // assign a name and type to your read_stream
    :read_stream("read_stream", "transactor")
    // assign name to this transaction generator
    // associated with the stream, and assign names
    // to the associated attributes.
    ,read_tr("read", read_stream, "addr", "data")
    {…}

  sc_uint<8> read(sc_uint<8>&* addr) {
    // signals the start of a transaction
    scv_tr_handle xactionHandle =
           read_tr.begin_transaction(addr);
    sc_uint<32> data;
    //process read
    …
    // signals the end of the transaction
    read_tr.end_transaction(xactionHandle
                                ,data);
    return data;
  }
};
```

Fig. 15.23 Transaction recording

Every time the read function is called, it records the transaction to the stream with a unique handle, and the transaction is terminated when the read transaction is done.

We encourage the reader to explore more advanced features of transaction recording in the SCV documentation. Be sure to see the exercises at the end of this chapter.

15.9 SCV Tips

Some simple observations are in order:

1. SCV extensions carry overhead both in execution speed and size. Be judicious in their use.
2. When designing constraints, remember to use the **`operator()`** to reference values.
3. Consider the option of overriding the **`next()`** method as a means to controlling the randomization.
4. Understand the distribution of your data when using a sparse array. For small arrays, a normal vector may be sufficient.
5. Don't abuse callbacks. Be strategic.
6. There is no substitute for a well thought out design.

15.10 Exercises

For the following exercises, use the samples provided at www.scftgu.com

Exercise 15.1: Randomize 1000 **`struct`**s containing two **`int`**s and display a histogram of their distribution. Divide the space into approximately 100 buckets.

Exercise 15.2: Using the same base code as in exercise 15.1, add restrictions to keep the random numbers in the ranges of 0–10 and 16–100.

Exercise 15.3: Create a distribution of $-\pi$ to $+\pi$ using **`float`**s. Using **`scv_bag`**, create a sinusoidal distribution.

Exercise 15.4: Create a custom structure to represent a configuration register space for a cell phone. Include fields for transmit/receive frequencies, display types (LCD, plasma, none), supported email and the phone number. Be sure the phone numbers are the correct number digits for your region. Restrict randomization of phone numbers to legal values for your region (e.g., USA uses athree-digit area code and a seven-digit number and neither group may begin with 0 or 1). Randomize 16 times and display. Use constraints to ensure numbers are not duplicates.

Exercise 15.5: Use callbacks to implement functional coverage with an STL map.

Exercise 15.6: Try using a sparse memory to model automobiles on a grid. Try using an STL `pair` as the index to model the x,y coordinate system.

Exercise 15.7: Explore transaction recording by reviewing, compiling and running the recording_ex code in the downloads.