SCV Randomization

John Rose, Stuart Swan - Cadence Design Systems, Inc.

28 October 2003

Table of contents:

Note: This version of this document is current with respect to the SCV 1.0 beta 4 kit which can be downloaded from www.systemc.org

1. Introduction to SCV Randomization

SCV is an open source C++ class library that layers on top of the open source SystemC class library to provide tightly integrated verification capabilities within SystemC. This white paper describes the capabilities and usage of the SCV randomization facilities.

Randomization is often used in logic verification to automatically generate design stimulus data that may be difficult to generate manually. This stimulus data can help uncover bugs that may otherwise go undetected.

Randomization can be used to choose types of tests to execute as well as to choose the data and control values for the design that will be generated during each specific test.

2. Related Documents and Materials

This paper provides detailed information about using SCV randomization. It is not intended as high-level introduction to either C++, SystemC, or SCV. Depending on your existing knowledge of SystemC and SCV, you may also wish to read some these other related documents and materials:

- A short, high-level slide introduction to SCV can be found at:

  www-ti.informatik.uni-tuebingen.de/~systemc/Documents/Presentation-7-UP1_swan.pdf

- A short whitepaper that introduces SCV can be found at:

  `www.testbuilder.net/whitepapers/sc_tut.pdf`

- A short whitepaper that introduces SystemC can be found at:

  `www.testbuilder.net/whitepapers/systemc_wp20.pdf`

- A broad introduction to SystemC can be found in the book “System Design with SystemC” available at www.systemc.org -> Products & Solutions -> Books.

- The SCV kit includes the full source code for SCV, documentation, and examples, and can be downloaded from www.systemc.org

- A wide range of fully working SCV examples can be found within the examples directory of the SCV kit.

- Very detailed reference information about SCV can be found in the “SystemC Verification Standard Specification” document within the `docs/scv/scvref` directory of the SCV kit.

- Discussion forums for SystemC related topics (including SCV, installation issues, etc) can be found at www.systemc.org -> Discussion Forum

3. Key Terms

Before discussing SCV randomization in detail, let us define the key terms related to randomization in SCV. These terms will be explained further later in this document, but here we provide a concise definition of all of the key terms so that they can be easily referred to as you read this document.

There are three types of randomization typically used in verification:

Unconstrained randomization: data variables have an equal probability of taking on any legal value allowed by the data type of the variable (e.g. an integer variable can take on values between MIN_INT and MAX_INT).

Weighted randomization: the set of legal values is weighted so that the probability of taking specific values is not uniform. For example, there may be a 90% probability that an integer variable will take a value between 0 and 10, and a 10% probability that it will take a value between 11 and 20. Weighted randomization is achieved in SCV by creating distributions using `scv_bag` and then using `set_mode` to assign the distribution to particular variables.

Constrained randomization: the set of legal values that a data variable can take is constrained according to some rules or by a set of constraint expressions. The constraint may be a simple limit of the legal values of the given data type, or it may be a complex expression involving multiple variables that are also being constrained.

Constrained randomization is a very powerful feature of SCV that allows complex stimulus to be generated from a concise set of constraint declarations. Here are the key terms related to constrained randomization. (These definitions may not be entirely clear to you until you read through this document, but they are presented here so that you can easily refer to them as you are reading the document.)

Constraint solver: The constraint solver is a software algorithm that is provided within SCV to generate legal values for variables whose values are constrained via complex constraint expressions.

Complex constraint expression: A complex constraint expression is any constraint expression that appears within the `SCV_CONSTRAINT/SCV_SOFT_CONSTRAINT` constructs. Complex constraints are specified using C++ expressions that evaluate to a simple Boolean value. Complex constraints can only be specified within constraint classes and appear within the constructors of constraint classes. An example of a complex constraint expression is the following:

```
SCV_CONSTRAINT(a() > b() && b() > c() && (a() – c() > 100) );
```

Simple constraint: A simple constraint in SCV is any construct that constrains the value of a single variable and which does not use the SCV_CONSTRAINT/ SCV_SOFT_CONSTRAINT constructs. Examples of simple constraints in SCV are the `keep_only/keep_out` constructs, and the `set_mode(scv_bag<T> / SCAN / RANDOM_AVOID_DUPLICATE)` constructs. These simple constraint constructs can be used both within constraint classes and outside of constraint classes.

Constraint class: A constraint class is any class that derives from class `scv_constraint_base`. Constraint classes in SCV are used to specify and group complex and simple constraints together so that they can be used to generate constrained random stimulus.

Constraint solver output variable: A constraint solver output variable is a variable for which a new value will be generated when the constraint solver is invoked. A variable is a constraint solver output variable if and only if the variable appears within an `SCV_CONSTRAINT/SCV_SOFT_CONSTRAINT` construct and it does not have `disable_randomization` or `keep_only/keep_out` or `set_mode(scv_bag<T> / SCAN / RANDOM_AVOID_DUPLICATE)` currently in effect for it. Note that any variable appearing within an `SCV_CONSTRAINT/SCV_SOFT_CONSTRAINT` construct must either be a constraint solver input variable or a constraint solver output variable, but it cannot be both at the same time. Note also that it is possible to dynamically change variables from being constraint solver output variables to constraint solver input variables and vice-versa as the simulation proceeds by dynamically using methods such as `disable_randomization`, `enable_randomization`, `keep_only/keep_out`, and `set_mode`.

Constraint solver input variable: A constraint solver input variable is a variable whose value will be not be generated by the constraint solver, but instead whose value will be used by the constraint solver to determine values of other variables appearing within `SCV_CONSTRAINT/SCV_SOFT_CONSTRAINT` constructs. A variable is a constraint solver input variable if and only if the variable appears within an `SCV_CONSTRAINT/SCV_SOFT_CONSTRAINT` construct and it has `disable_randomization` or `keep_only/keep_out` or `set_mode(scv_bag<T> / SCAN / RANDOM_AVOID_DUPLICATE)` currently in effect for it. (When the constraint solver is invoked using the "`.next()`" method on a subfield within a constraint class, this is equivalent to temporarily applying `disable_randomization` to the other higher level fields within the constraint class, and so those fields are also constraint solver input variables in this case.) Note that any variable appearing within an `SCV_CONSTRAINT/SCV_SOFT_CONSTRAINT` construct must either be a constraint solver input variable or a constraint solver output variable, but it cannot be both at the same time.

4. Key Requirements of a Constrained Randomization Solution

There are six keys to any good constrained randomization solution:

- Solutions must be uniformly distributed over the legal values. For example, if the legal values to satisfy a constraint are {0, 1, 4, 5} each of these numbers must have a 25% probability of being selected.

- Performance of constraint solver as the number of variables used within constraint expressions (i.e. variables within SCV_CONSTRAINT/SCV_SOFT_CONSTRAINT constructs) grows. For the constraint solver to be useful, it must be able to handle a large number of variables.

- Commutativity (order independence) of constraint equations. For example, the equations a == b must produce the same results as b == a.

- The ability of the user to create relevant constraints involving multiple variables. This implies that the API for creating constraint expressions is rich enough to express complex constraints of typical systems.

- Ability to debug over-constraints. When a constraint is unsolvable, it is critical that a user have a mechanism for figuring out why a constraint can't be met (e.g. is the constraint expression in error, or are constraint solver input variables taking on invalid values).

- The ability to control the value generation of constrained objects. This involves the ability to enable/disable randomization, and the ability to control pre-generation and post-generation work.

The SCV constrained randomization solution meets all of these requirements.

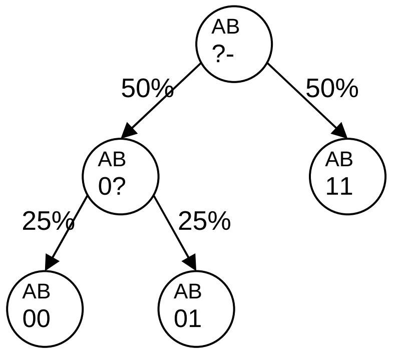
5. Uniform Distribution of Solutions Under Constraints

The uniform distribution of solutions with complex constraints is one of the most difficult aspects of developing a good constraint solver. The SCV constraint solver works on individual bits of variables when solving complex constraints. When a constraint expression is created, SCV generates a binary decision diagram (BDD) for the expression. The nodes of the BDDs are weighted based on the type of the expression in such a manner that the solutions picked are uniformly distributed across the set of all possible legal values.

For example, consider two one bit variables “a” and “b” having the constraint

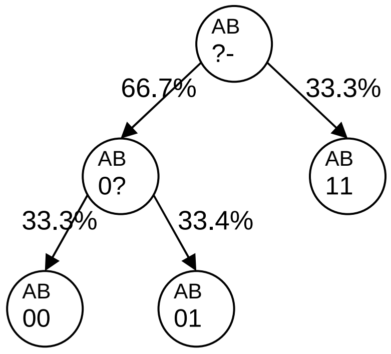
```
    SCV_CONSTRAINT(a() == 1 && b() == 1) || ( a() == 0));
```

In the above expression, a=1 and b=0 is not a legal value. In this case a simple binary graph where left arrows select “0” and right arrows select “1” might appear as:



Note that if each path from any node has a 50% chance of being selected, the leaf node with the value a=1 and b=1 has a 50% overall probability of being selected, and the leaf nodes with values (a=0,b=0) and (a=0,b=1) each have a 25% overall probability of being selected. This is a poor randomization solution because it incorrectly weights the probabilities towards one of the solutions.

The SCV constraint solver handles such issues by generating a probability for each path from a node such that traversal of the BDD will be uniform (e.g. each legal solution will have an equal probability of being selected). This is similar in concept to the binary graph shown below.



The SCV solution to this problem produces a solution that achieves probabilities of very close to 33.3% for each of the three solutions. Running the above equation 10000 times in SCV produces the results

Solution	**Number of Hits**	**Probability**
{0,0}	3403	34.03%
{0,1}	3267	32.67%
{1,0}	0	0%
{1,1}	3330	33.3%

6. Performance of the Constraint Solver

Solving simple constraints (e.g. ranges of legal values specified via `keep_only/keep_out, set_mode(scv_bag),` etc.) on a single variable is relatively fast, and performance does not change significantly as the constraints on the variable are changed. However, complex constraints on one or more variables specified via `SCV_CONSTRAINT/SCV_SOFT_CONSTRAINT` can have a significant impact on performance.

For SCV, the primary factor in the amount of time it takes to solve a complex constraint is the total number of variable bits and the total number and type of operations appearing within the `SCV_CONSTRAINT/SCV_SOFT_CONSTRAINT` constructs in a constraint class.

For example, a complex constraint involving four `int` variables requires 4x32=128 bits to be calculated. If the number of bits being evaluated is under 10,000, the performance of the constraint solver will be good. As the number of variable bits goes beyond this, the constraint solver may become a bottleneck, and it may be necessary to find ways to reduce the size or number of variables within the constraint class, possibly by partitioning it into several separate constraint classes. As a rule it is better to have a large number of small constraint classes, rather than a small number of large constraint classes. This helps improve both constraint solver performance and code clarity, and makes it easier to debug constraint solver failures when they occur. All of the limits mentioned in this section (e.g. number of bits within a complex constraint expression) relate to a single constraint class instance. By creating a large number of small constraint class instances you can circumvent many of the limits mentioned in this section.

It should also be noted that the number of bits in the constraint will affect the amount of memory that is required to produce the BDD. A BDD will grow non-linearly with respect to the number of bits. Thus, a constraint with a large number of variables may have an impact on the memory required for a simulation.

6.1 Simple Constraints vs. Complex Constraints

When a variable needs to be constrained but the variable does not have constraint relationships with any other variables, it is often possible to use the simple `keep_out/keep_only` constraint mechanism in SCV. Simple constraints do not use the BDD constraint solver, and can thus be solved much faster than equivalent constraint expressions. In general, it is a good practice to use the simple constraint mechanism when this is all that is required.

```
Example: Performance trade-off of simple vs. complex constraints

//fast simple constraint. Valid values for a are 10-20, 50-60, 90-100
struct faster : public scv_constraint_base {
  scv_smart_ptr <int> a;
  SCV_CONSTRAINT_CTOR(faster) {
    a->keep_only(10, 100);
    a->keep_out(21, 49);
    a->keep_out(61, 89);
  }
};

//slower complex constraint. values for a are 10-20, 50-60, 90-100
struct slower : public scv_constraint_base {
  scv_smart_ptr <int> a;
  SCV_CONSTRAINT_CTOR(slower) {
    SCV_CONSTRAINT( (a() >= 10 && a() <= 100) &&
                   !(a() >= 21 && a() <= 49) &&
                   !(a() >= 61 && a() <= 89) );

  }
};
```

In the above example, using the simple constraints is approximately 2x faster. In other situations the performance difference could be even greater. Note that in this simple example it is not a requirement to use a constraint class for the simple constraint, since a simple standalone `scv_smart_ptr<int>` would be sufficient in this case.

6.2 Variable Size

When generating complex constraints in SCV, it is a good practice to be cognizant of the size of variables inside of equations. For example, if an integer type is being used, then it requires 32 bits. However, if that type is being constrained to 4 bit values (values between 0 and 15), then it will be more efficient to use a `sc_uint<4>` type.

```
Example: Performance trade-off sized variable vs. complex constraint

//less efficient case using “int”
struct less_efficient : public scv_constraint_base {
  scv_smart_ptr <int> a, b;
  SCV_CONSTRAINT_CTOR(less_efficient) {
    SCV_CONSTRAINT( 0 <= a() && a() <= 15 );
    SCV_CONSTRAINT( 0 <= b() && b() <= 31 );
    SCV_CONSTRAINT( (a() + b()) < 24 );
    SCV_CONSTRAINT( (b() – a()) >= 10 );
  }
};

//more efficient case using sc_uint<4> and sc_uint<5>
```

```
struct efficient : public scv_constraint_base {
  scv_smart_ptr <sc_uint<4> > a;
  scv_smart_ptr <sc_uint<5> > b;
  SCV_CONSTRAINT_CTOR(efficient) {
    SCV_CONSTRAINT( (a() + b()) < 24 );
    SCV_CONSTRAINT( (b() – a()) >= 10 );
  }
};
```

In the above example, the efficient version of the constraint will operate approximately 30% faster than the less efficient version. In a testbench that uses a large number of complex constraints, this can have a even more significant impact on overall simulation performance.

6.3 Operators

Another consideration when creating complex constraints is the use of operators. For SCV (because it uses a BDD approach), Boolean operations are extremely efficient, however, arithmetic operations can slow down the constraint solver. This is due to the fact that arithmetic operations can require a large number of nodes for each solution. Multiplication in particular is the most inefficient operation; the current implementation can become excessively slow when solving complex constraints which have multiplication operations on two variables where the result of the multiplication is more than 20 bits.

Operator	**Maximum variable bits***
&&, ||, !, !=, ==,&,|,~,>>,<<	2048
>, >=, <, <=	2048
+, -	1024
*	20**

*It is possible to use larger variables. However, as the number of bits goes up, the size of the BDD grows and the generation of the BDD takes much longer. Constraint solving remains relatively fast, but the memory required to hold the BDD goes up significantly.

**Multiply operations can have a significant hit on performance. You can successfully multiply a larger sized bit vector by a constant (for instance a 32 bit integer by a 10 bit constant). When multiplying two variables within a complex constraint, try to avoid having the result of the multiplication be greater than about 20 bits.

7. Commutativity of Constraint Expressions

A key aspect of a good constraint solver is that the commutativity of an expression is maintained. That is, the order of variables in the expression can be changed without affecting the result of a randomization. The reason this is important is because you don't want the value of the variables to be dependent on which variable is solved first. It is important that all variables be solved simultaneously in order to get valid results.

Examples of commutativity of the SCV constraint solver:

Example: `Constraint commutativity`

```
struct abconstraint : public scv_constraint_base {
  scv_smart_ptr <sc_uint<4> > a, b;
  SCV_CONSTRAINT_CTOR(abconstraint) {
    SCV_CONSTRAINT( a() != 10 );
    SCV_CONSTRAINT( b() != 7 );
```

```
      SCV_CONSTRAINT( b() == a() + 2 );
    }
  };
```

In the above example, if the SCV constraint solver does not solve for the variables together, it is possible to get incorrect results. For instance, if “a” is solved first, it is possible that “a” may take the value 5, which would cause “b” to take the value 7. Since “b” can’t take the value 7, this solution would not be valid. Likewise, if “b” is solved first it may take a value of 12 which would cause “a” to take the value 10 which is not valid.

A constraint solver that solves one variable followed by the other in this way will potentially create non-uniform distributions over legal values. And, in the worst case, such a constraint solver will generate invalid values, or no values at all even when legal solutions to the constraint expressions exist.

8. Creating Simple and Complex Constraints in SCV

This section discusses the classes involved in SCV randomization and concentrates on the syntax for creating constrained objects in SCV. The way to create a constrained object depends on the type of constraint you plan to use (simple or complex). The different types of constraints will be discussed in detail in this section.

8.1 Classes involved in SCV randomization

The classes involved in SCV randomization are:

C++ built-in and user-defined types: These are the core data types used to model the variables that the user wishes to randomize. Examples include `int, char, sc_uint<>` and user-defined data types such as a `packet` structure. Any normal C++ object, SystemC object, or user defined class, struct or enumerated type can be used as a core data type. C++ template parameters can also be used to parameterize these types.

scv_extensions<T>: This class enables SCV type extensions to be specified for the core data types mentioned above. Most of the features within SCV are implemented via this class. The SCV type extensions for C++ built-in types and SystemC types are already included within SCV, so the user doesn’t need to do anything to use these type extensions. For user defined types and enumerations, the user generally needs to create a SCV extensions class before such types can be used within SCV. (There are tools such as cve_wizard_ext available at www.testbuilder.net that can automate this task). If you are not familiar with how to create SCV type extensions, you may want to read the section “Defining Extensions for Data Types” within the “SystemC Verification Standard Specification” within the `docs/scv/scvref` directory in the SCV kit. Also study the `examples/scv/extensions/{ex_01_userdata, ex_02_enumdata, ex_03_nestdata}` examples within the SCV kit.

scv_smart_ptr<T>: The user instantiates this class and uses it as if it were a pointer to the underlying core type “T”. The SCV randomization capabilities are automatically available via the use of this class. The `scv_smart_ptr<T>` object maintains a pointer to the core data object (of type T) as well as a pointer to the `scv_extensions<T>` object that provides the features specific to SCV.

scv_constraint_base: A user derives from this base class for creating constraint classes. Constraint classes contain one or more data objects (i.e. core data types that are referenced via `scv_smart_ptr<>` objects) as well as constraint specifications (simple or complex) on those objects.

With these types, C/C++ built-in and user defined data types are the core data object that a user creates and uses. When a variable will be randomized, a `scv_smart_ptr<T>` object is used instead (where T is a built-in or user defined data type). The `scv_smart_ptr<T>` object has access to the built-in or user defined object but also provides the ability to create simple constraints, set distribution modes, and randomize the variable.

8.1.1 Smart Pointer Concepts

The `scv_smart_ptr<T>` and `scv_shared_ptr<T>` classes are C++ "smart pointer" types. Smart pointers are objects in C++ that mimic the capabilities of normal "dumb" C/C++ pointers, but which also have intelligent capabilities added such as automatic memory management and garbage collection. Both `scv_smart_ptr<T>` and `scv_shared_ptr<T>` provide automatic memory management and garbage collection. If the concept of smart pointers is new to you, you may want to read the section on smart pointers and reference counting within the book "More Effective C++" by Scott Meyers. You may also want to read the link www.boost.org/libs/smart_ptr/shared_ptr.htm since this latter class is quite similar to `scv_shared_ptr<T>`.

The automatic memory management and garbage collection features of `scv_smart_ptr<T>` and `scv_shared_ptr<T>` are particularly useful when using SystemC and SCV because of the multi-threaded nature of SystemC and SCV designs and testbenches. In normal single-threaded C/C++ programming, ensuring that objects are properly deleted when they are no longer needed is a typical source of bugs. In SystemC the problem becomes even greater because more than one process may be accessing an object, so it becomes difficult to coordinate when the object should be deleted. Fortunately `scv_smart_ptr<T>` and `scv_shared_ptr<T>` automate the tasks associated with memory management and deletion and thus remove a large source of potential bugs.

Both `scv_smart_ptr<T>` and `scv_shared_ptr<T>` rely on reference counting techniques to provide automatic garbage collection. As discussed in the "More Effective C++" sections on smart pointers and reference counting, there are two caveats when using such smart pointers:

- Sometimes you may wish to obtain a "dumb" pointer to the underlying data object that the smart pointer points to. This should be done only with caution since the smart pointer doesn't know if you will retain the dumb pointer, and the smart pointer may subsequently delete the underlying object. When this happens the dumb pointer, if retained, will no longer point to valid data.

- Simple reference-counted smart pointers such as `scv_shared_ptr<T>` can be used to create links within acyclic dynamic data structures such as lists and trees. When the root node of such data structures is deleted, the reference counting scheme will ensure that the entire data structure is deleted at the appropriate time. However, reference-counted smart pointers should not be used to create cyclic data structures such as a ring data structure, since the reference-counting scheme will then never see the reference counts for such cyclic structures go to zero when there are no longer any external links to the data structure.

The `scv_smart_ptr<T>` class provides the `get_instance()` method to obtain a "dumb" pointer to the underlying data object. While this method is useful in a number of circumstances, it is important to realize that it should be used with caution. The pointer that is returned should not be retained such that it might be used after the underlying data object has been deleted by the smart pointer, since the smart pointer can not track the dumb pointer's lifetime. Rather than using the `get_instance()` method, it is almost always preferable to use the `read()` method of `scv_smart_ptr()`. The `read()` method returns the *value* of the underlying data object rather than a pointer to it.

As a rule, you should consider the type T used within a `scv_smart_ptr<T>` declaration to be a fixed-size data type that simply stores *values*. Furthermore you should consider an `scv_smart_ptr<T>` object to be a container for the underlying object of type T that allows *values* to flow in and out, but which does not allow *pointers* to flow in and out. So for example if you have a `scv_smart_ptr<my_packet>`

object, and the `my_packet` structure has `source, destination and payload` fields, it is safe to read and write the values of the `source`, `destination` and `payload` fields thru the smart pointer, but it is not advisable to obtain a pointer to the `source`, `destination, or payload` fields.

But what if the `payload` field needs to be a variable-sized array? One simple solution is to use a fixed size array for the `payload` field that is large enough to accommodate all possible payload sizes, along with a separate field that records the current (actual) payload size. For an example of this, see `examples/scv/randomization/ex_12_hier_constraint` in the SCV kit. An alternative way of handling variable-sized data structures that are to be randomized is presented in the "Performing Post-Generation Actions" section of this paper.

Why doesn't SCV randomization directly handle variable-sized data structures or dynamically changing data structures such as dynamic lists and trees? The reason is that it becomes quite problematic to apply the core SCV randomization capabilities directly to data structures that can dynamically change in size. As an example of such problems, if parts of a dynamic data structure that had complex constraints applied to it were deleted, what would or should SCV do with the complex constraints that now reference non-existent data? To avoid such issues, SCV at a fundamental level only supports randomization for fixed size data structures, but as mentioned above there are various ways to mimic randomization on dynamic data structures. The restriction to fixed size data structures also makes it easier to debug testbench problems and constraint solver failures when they occur, since the data structures that are being randomized are not changing during simulation.

Since SCV randomization works at a fundamental level on fixed-size data structures that simply store values, `scv_smart_ptrs` have some specific restrictions that you should be aware of. The underlying data type for an `scv_smart_ptr` (i.e. the T data type in `scv_smart_ptr<T>`) must adhere to the following rules.

- The type can be any C/C++ built-in type such as `int, unsigned, char, double,` etc.

- The type can be any SystemC abstract data type such as `sc_uint<>, sc_bv<>, sc_lv<>, sc_bigint<>` etc. (Such abstract data types simply store *values*.) It cannot be a SystemC channel, interface or port such as `sc_in<>` since such types do not simply store values.

- The type can be a user-defined `struct` or `class or enum.` In this specific case you will need to create an `scv_extensions<>` class for the user-defined `struct or class` or `enum` as described above.

- If the type is a user-defined `struct` or `class`, then the `struct` or `class` may contain data members that are any of the types listed above, and in addition it may contain members that are single or multidimensional arrays of any of the types mentioned above. For example, it is possible to have a member that is a multidimensional array of `structs`.

- The type must not contain any `scv_smart_ptrs` or `scv_shared_ptrs or scv_extensions` objects.

- The type must not contain STL collection objects such as `list<>` or `vector<>.`

- The type usually should not contain any C/C++ "dumb" pointers or C++ "references". If the type does contain pointers or references, SCV will ignore them for the purposes of randomization. See the "Performing Post-Generation Actions" section of this paper for an example of how pointers can be used to model variable-sized data structures that are randomized in SCV.

- You cannot declare a smart pointer of the form:

```
scv_smart_ptr<T [10]> a;
```

Instead you can write:

```
scv_smart_ptr<T> a[10];
```

Alternatively you can wrap the array in a `struct` or `class` as follows:

```
struct S {
  T a[10];
};

SCV_EXTENSIONS(S)
{
  public:
    scv_extensions<T [10]> a;

    SCV_EXTENSIONS_CTOR(S) {
      SCV_FIELD(a);
    }
};

scv_smart_ptr<S> s;
```

When an `scv_smart_ptr<T>` object is constructed, you can optionally supply a constructor argument that will initialize the object that the `scv_smart_ptr` points to. You have several choices:

- If you don't supply any object of type `T` to initialize the `scv_smart_ptr`, the `scv_smart_ptr` will automatically create an object of type `T` by calling `T's` default constructor. Later, when there are no longer any `scv_smart_ptrs` that point to the underlying object, it will be automatically deleted.

- If you supply an object of type `T` as a value or const reference, then the `scv_smart_ptr` will automatically create a new object of type `T` by calling `T's` copy constructor. Later, when there are no longer any `scv_smart_ptrs` that point to the underlying object, it will be automatically deleted.

- If you supply a pointer to an object of type `T` to the `scv_smart_ptr` constructor, then the `scv_smart_ptr` will not create a new underlying data object, but instead it will take over ownership of the object that you supplied. Later, when there are no longer any `scv_smart_ptrs` that point to the underlying object, it will be automatically deleted. Note that this means that you must not explicitly delete the object that you passed via pointer to `scv_smart_ptr`, and that the object must have been dynamically allocated on the heap via `new()`.

8.1.2 Instantiating a data object

A data object is instantiated directly or through a smart pointer. When an object is instantiated through a smart pointer, the smart pointer object points to the `scv_extensions<T>` object of the underlying object (not directly to the underlying object itself). The importance of this is that when a method of the underlying class needs to be called (for example), the user must access the underlying object first and then call the method. You can access the underlying object by using the `get_instance()` and `read()`methods as shown in the example below.

Example: Instantiation of randomizable user defined data objects

```
//definition of a user defined data object
struct mydata {
  int d1;
  sc_uint<8> d2;
  void print()  { cout << “d1: “ << d1 << “  d2: “ << d2 << endl; }
};

//definition of the scv extensions for the data type. This can be
//automatically created using a tool like cve_wizard_ext. Note: built-in
//types and systemc types already have scv extensions classes provided.

SCV_EXTENSIONS(mydata)
{
  public:
    scv_extensions<int> d1;
    scv_extensions<sc_uint<8> > d2;

    SCV_EXTENSIONS_CTOR(mydata) {
      SCV_FIELD(d1);
      SCV_FIELD(d2);
    }
};

//instantiating and using objects
void doit () {
  //instantiate a normal data object
  mydata d;

  //instantiate a randomizable data object
  scv_smart_ptr<mydata> d_p;

  //randomize the entire underlying object the smart pointer points to
  d_p->next();

  // call a method in the mydata structure. For the normal data object,
  // just call the method. For the smart pointer, must use get_instance()
  // to access the actual object.
  d.print();
  d_p->get_instance()->print();

  //randomize only field d2 while d1 retains its current value, then print
  d_p->d2.next();
  d_p->get_instance()->print();

  //assign to “d” the value pointed to by the smart pointer
  d = *d_p;
  d = d_p->read(); // same effect

  // write value of “d” to the data object within the smart pointer
  *d_p = d;
  d_p->write(d); // same effect

  // read one of the fields of the data object within the smart pointer
  sc_uint<8> d2;
  d2 = d_p->d2;
```

```
    d2 = d_p->d2.read(); // same effect

    // write one of the fields of the data object within the smart pointer
    d_p->d2 = d2;
    d_p->d2.write(d2); // same effect
  }
```

8.1.3 Copying data objects

The `scv_smart_ptr<>` data object is a pointer type object. Thus, copying a smart pointer does a shallow copy (in essence it just copies the pointer). There can be cases when you want to do a deep copy of a smart pointer object. The example below shows how the "`read()`" and "`write()`" methods provided with `scv_smart_ptrs` can be used to perform a deep copy of the value of underlying data type (the type `mydata` in this case).

```
Example: Deep copy of randomizable user defined data objects

class constraint : public scv_constraint_base {
  public:
    scv_smart_ptr<mydata> d;
    …
};

void dotest() {
  constraint c(“c”);  //constraint object used for data generator
  scv_smart_ptr<mydata> local;  //local data object

  for(int i=0; i<5; ++i) {
    c.next();  //generate constrained data
    local->write(c.d->read());  //do a deep copy of the data
  }
}
```

8.2 Keep_only / Keep_out modes

The `keep_only()` and `keep_out()` methods are made available via the use of `scv_smart_ptr<T>`. These simple constraint methods allow a user to specify specific sets of legal values for a variable. The values may be specified using ranges, or using lists.

`Keep_only` constraints use intersection semantics. Thus, a `keep_only(10,20)` and `keep_only(15,25)` will result in legal values of `{15,20}`. (In contrast, we will see in the next section that distributions using `scv_bag()` use union semantics when multiple objects or ranges are placed in a distribution. So if you desire union semantics you should consider using a distribution rather than `keep_only`.)

8.2.1 Simple Ranges

Specifying a range involves providing the min and max value of the range. In SCV the minimum and maximum values are inclusive (e.g. keep_only(0,10) includes the values 0 and 10 as legal values).

```
Example: Creating simple constraint ranges

void doit() {
  scv_smart_ptr<sc_uint<8> > data_p;

  //set the legal values to be between 30 and 100
  data_p->keep_only(30, 100);
```

```
      //keep out the value 50 and the value between 75 and 80 inclusive
      data_p->keep_out(50);
      data_p->keep_out(75,80);

      for(int i=0; i<10; ++i) {
        data_p->next();
        dosomething( *data_p );
    }
    void dosomething(int d) { cout << “d = “ << d << endl; }
```

The `keep_only/keep_out` methods are not limited to integer arguments. The underlying data type could be a C++ enumerated type (e.g. an enum representing instruction opcodes), in which case the enumeration literals can be directly supplied to `keep_only/keep_out`. It is also possible to supply `float` and `double` arguments to `keep_only/keep_out` for a variable that has a floating point type. In general, the data type supplied to `keep_only/keep_out` only needs to be the same as the underlying data type that is being constrained, and it could be any user-defined data type that supports the equality and comparison operators.

8.2.2 Lists of values

It is also possible to specify `keep_only` and `keep_out` of specific values by using lists (from the C++ STL library). This is convenient, for example, when you want to restrict read operations to addresses that have been previously written. Calls to `keep_only` are cumulative -- that is, if you call `obj->keep_only(mylist)` twice with different values, the set of legal values will then be the intersection of the first list with the second list. Calls to `keep_out` are also cumulative -- that is, if you call `obj->keep_out(mylist)` multiple times, each invocation removes the specified items from the values allowed for the variable in a cumulative manner.

Note, when a list is supplied to `keep_only`, the current values in the original list are copied to the `keep_only` constraint. Future additions to and removals from the original list are ignored unless you explicitly invoke `keep_only/keep_out` again.

```
    Example: Using lists of legal values

    SC_MODULE ( mod ) {

     toi);jT*(   lis g//dhe )]TttenejT*(   lis}jT*(   lise)soreto ao_d o)(uint<10> > l toi{
};
```

8.3 Distribution modes

In SCV distribution modes are a form of simple constraints which allow you to:

- create weighted distributions
- create weighted distribution ranges
- scan the set of legal values
- obtain random values without duplicating previously generated values

Using these special distribution modes allows a user to apply special control to generation of random values.

8.3.1 Weighted distribution

Weighted distribution is useful when an object can take a specific number of legal values, and when it is desirable for certain values to have a better chance of selection than others.

A typical case of this would be in the selection of transaction types, or opcodes for a specific interface. Generally, an interface has a discrete number of legal transaction types, and that number of types is usually small (less than 20). Likewise for instruction opcodes in a processor design, it is usual for the number of legal codes to be relatively small.

Creating a weighted distribution involves making a `scv_bag` object and populating it with values. The number of objects of a given value determine the weight of the value. The weight is relative to the total number of objects in the bag.

```
Example: Weighted distribution for an enumerated data object

// The enum must have an scv_extensions<T> class created for it.
// This can be done using a tool like
// cve_wizard_ext, or it can be done manually. The scv_extensions class
// is not shown here.

enum tx_types { READ, WRITE, BURST_READ, BURST_WRITE };

void doit () {
  scv_smart_ptr<tx_types> p; //randomization object
  scv_bag<tx_types> dist;  //a bag of transactions

  dist.add(READ, 40);
  dist.add(WRITE, 40);
  dist.add(BURST_READ, 10);
  dist.add(BURST_WRITE, 10);

  //there are 100 objects in the bag, 40 each of READ and WRITE, and 10
  //each of BURST_READ and BURST_WRITE. Using 100 total objects makes
  //the percentages more intuitive, e.g. a 40% probability of a READ
  //transaction.

  p->set_mode(dist); //use the bag as the distribution mode

  while (...) {
    p->next();
    switch(p->read()) {
      case READ: ... break;
      case BURST_READ: ... break;
      …
    }
  }
}
```

Note that the scv_bag data structure conceptually allows multiple copies of the same object values to be stored. However, the scv_bag implementation only keeps a single copy of each distinct object value that has been placed into the bag, along with a count of the number of copies of that object value that are currently in the bag. For this reason scv_bag is very efficient even if the total number of conceptual items in the bag is very large, as long as the number of different object values is not too large. (Hundreds or even thousands of different object values is still OK.)

8.3.2 Weighted range

A weighted range is similar to a weighted distribution shown above, except that *ranges* of values (rather than discrete values) are given probabilities. This is useful for creating non-uniform probabilities that specific sections of a device are used. For instance, in a memory mapped io device, this type of weighting may be used to set the probability that addresses associated with specific devices may be selected with higher probability than other devices.

Weighted ranges involve an scv_bag object that is populated with pairs of objects. Each pair represents a range of values that the object can take.

```
    Example: Distribution ranges on an integer data object

    void doit () {
      typedef pair<unsigned int, unsigned int> range_t;
      scv_smart_ptr<unsigned int> addr; //randomization object
      scv_bag<range_t> dist;  //a bag of ranges

      dist.add(range_t(0x0, 0x10), 25);
      dist.add(range_t(0x11, 0x100), 25);
      dist.add(range_t(0x1000, 0xffff), 25);
      dist.add(range_t(0x200000, 0xffffffff), 25);

      // In this example each range has an equal weighting (25) and thus an
      // equal chance of being selected. Within each range
      // the distribution will be uniform, so specific values within the
      // first range have a much higher probability of being selected
      // than specific values in the last range.

      addr->set_mode(dist); //use the bag as the distribution mode
      addr->next();
    }
```

You can use distribution ranges to approximate mathematical distributions such as Gaussian and exponential distributions. To do this, use an algorithm to divide the solution space into a set of discrete ranges (possibly having equal or varying lengths) and then assign an appropriate weight to each range. (An alternative way to generate mathematical distributions such as Gaussian distributions is to use an external randomization function that is invoked as a pre-generation action. To see this technique read the “Pre-Generation Actions” section of this paper.)

Consider the following example. Assume you wish to generate the range [0,1] 40% of the time and the range [2,10] 60% of the time. This can be achieved with the following code:

```
Example: Weighting of distribution ranges

scv_smart_ptr<int> data;
scv_bag<pair<int, int> > dist;
dist.add(pair<int, int>(0,1),  40);
dist.add(pair<int, int>(2,10), 60);
data->set_mode(dist);
```

In this example, when `data->next()` is called, a range will be selected according to the weights, and then a value will be picked from the selected 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 weights 40 and 60 correspond to the area under their respective ranges.

If the ranges within a distribution overlap a warning will be printed when `set_mode()` is called.

8.3.3 Scan mode

Scan mode allows the set of legal values to be traversed from the lowest to highest (instead of being accessed randomly). This mode is especially useful when the set of legal values is discontinuous. An object can use scan mode in conjunction with `keep_only/keep_out` constraints (ranges and lists).

```
Example: Scan mode with a keep constraint

void doit() {
  scv_smart_ptr<sc_uint<8> > data_p;

  //set the legal values to be between 30 and 100
  data_p->keep_only(30, 100);

  //keep out the value 50 and the value between 75 and 80 inclusive
  data_p->keep_out(50);
  data_p->keep_out(75,80);

  //set the mode to scan from the legal values
  data_p->set_mode(scv_extension_if::SCAN);
  for(int i=0; i<10; ++i) {
    data_p->next();
    dosomething( *data_p );
}
void dosomething(sc_uint<8> d) { cout << “d = “ << d << endl; }
```

8.3.4 Avoid duplicate

The last mode available for simple constraints is the avoid-duplicate mode. In this mode, a value is not reused until all possible legal values have been exhausted. This mode keeps a history of all of the values that the random object has taken and insures that the next selected value is not in the list of previously used values. This mode can be expensive if the legal value set is very large, and a large number of the legal values have been used.

```
Example: Avoid duplicate mode with a keep constraint

void doit() {
  scv_smart_ptr<sc_uint<8> > data_p;
```

```
      //set the legal values to be between 30 and 100
      data_p->keep_only(30, 100);

      //keep out the value 50 and the value between 75 and 80 inclusive
      data_p->keep_out(50);
      data_p->keep_out(75,80);

      //set the mode to scan from the legal values
      data_p->set_mode(scv_extension_if::RANDOM_AVOID_DUPLICATE);
      for(int i=0; i<10; ++i) {
        data_p->next();
        dosomething( *data_p );
    }
    void dosomething(sc_uint<8> d) { cout << "d = " << d << endl; }
```

8.4 Constraint Classes

The use model for SCV is to place objects that have complex constraint relationships inside constraint classes. Within constraint classes, simple constraints may be created for objects, and complex constraints may be set on multiple objects. The advantages of this use model are that it separates the declaration of the underlying core data type (e.g. a packet structure) from the declaration of constraints that are to be applied to the data type, and it allows you to use C++ inheritance (including multiple inheritance if desired) to efficiently reuse constraint declarations.

Constraint classes can be considered as containers for both the objects to be constrained and for the complex constraint expressions to be applied to those objects. In particular SCV enforces the discipline that complex constraint relationships can not cross outside of the containing constraint class. There are good reasons for this. Remember that complex constraints are declarative and bidirectional and rather than procedural. If complex constraints could be applied to arbitrary global objects, then due to the transitive nature of complex constraints it would be very easy to inadvertently create constraint relationships that affected many global objects that were apparently unrelated, causing lots of bugs or constraint solver failures. (For example, what would or should be done if the constraints crossed outside of the constraint class to a global object and that object was then deleted?) With the SCV use model, constraints are fully contained within their constraint classes, so it is easy to identify which objects can be affected by the constraints. Perhaps this constraint modeling discipline within SCV seems limiting, but in practice it leads to clean, well-structured code because it encourages the use of hierarchical constraints (using both C++ single and multiple inheritance) to cleanly compose constraints.

An important concept in the SCV constraint class use model is that complex constraint expressions are static. That is, each constraint class contains a set of complex constraint expressions which provide the constraints on the variables. These expressions are created in the constructor of the constraint class, and cannot be modified after the constraint object has been instantiated. However, the constraint expressions can be parameterized with `scv_smart_ptr` variables which will not be randomized, but which will affect the value generation of other variables in the expression. We will show examples of how to create parameterized complex constraints later in this paper.

8.4.1 Creating a base constraint class

A constraint class derives from `scv_constraint_base` and contains one or more smart pointer objects. The `scv_smart_ptrs` within a constraint class must be instantiated as simple objects (e.g. `scv_smart_ptr<int> i`;) – they must not be pointers or references to `scv_smart_ptrs` (e.g. `scv_smart_ptr<int>* i`). Generally at least one of the smart pointer objects within a constraint class will have either simple or complex constraints associated with it, but it is not a requirement that all smart pointer objects within the constraint class have simple or complex constraints associated with them. Below is an example of a constraint class definition.

```
Example: Creating a base constraint class

class my_constraint_class : public scv_constraint_base {
  public:
    //instantiate randomizable objects using scv_smart_ptr’s
    scv_smart_ptr<my_packet> p;
    scv_smart_ptr<int> d;

    SCV_CONSTRAINT_CTOR(my_constraint_class) {
      //add constraints to smart ptrs using operator()
      SCV_CONSTRAINT( p->addr() >= d() && p->addr() <= p->data() + 100 );
    }
};
```

Using a constraint object merely requires instantiating an object of the constraint class and calling the `next()` method. Unless `disable_randomization` has been set for certain variables, calling `next()` will randomize all `scv_smart_ptr<>` variables in the constraint class, as well as all objects in a composite `scv_smart_ptr<>` type (e.g. in the example above, `my_packet` is a composite type and all of its fields with be randomized).

```
Example: Using a constraint object

//using the constraint object
void doit() {
  my_constraint_class c (“c”); //requires a name
  for(int i=0; i<5; ++i) {
    c.next();  //generate values
    cout << “addr: ” << c.p->addr << “ data: ” << c.p->data << endl;
  }
}
```

Complex constraints can be specified using either the `SCV_CONSTRAINT` or `SCV_SOFT_CONSTRAINT` constructs. If soft constraints are specified, when the constraint solver is invoked it will find solutions that satisfy both the soft and the hard constraints if such solutions exist. If no such solution exists, it will ignore the soft constraints and then it will find solutions using the hard constraints if such solutions exist.

8.4.2 Hierarchical constraints

Constraint classes may be organized using standard C++ class hierarchies. This allows constraints for specific tests to be separated from base constraints that may apply to many tests. Often it is useful to organize a testbench such that the test procedure accepts a user defined constraint base class that is common to the hierarchy of constraints that have been created. Because the test procedure (`dotest()` below) accepts the base constraint class, any constraint class derived from the common constraint base class can be supplied.

```
Example: Creating and using a class hierarchy of constrained objects

//a hierarchical constraint based on the base constraint from above
class derived_constraint : public my_constraint_class {
  public:
    //create hierarchical constraints. The base constraints are inherited
    SCV_CONSTRAINT_CTOR(derived_constraint) {
      //must derive the base class constraints
      SCV_BASE_CONSTRAINT(my_constraint_class);

      //add a constraint
      SCV_CONSTRAINT( p->length() <= 52 && p->length() >= p->data() );
    }
};

// the test takes a reference to the common base class, but the
// reference can also refer to any class derived from the base class
void dotest(my_constraint_class& c) {
  for(int i=0; i<5; ++i) {
    c.next();  //generate values using derived class
    cout << "addr: " << c.p->addr << " data: " << c.p->data << endl;
  }
}
//the top level test
void maintest() {
  //use the derived class for the specific test to be applied
  derived_constraint c("c");
  dotest( c );
}
```

The hierarchical constraint class mechanism allows test writers to derive complex constraints but functions and tests can be written using the base constraint class for data generation (as in the `dotest()` function above). The data generation will utilize the conjunction of the derived constraints and the base constraints. A further example of hierarchical constraints is in the `examples/scv/randomization/ex_12_hier_constraint` example in the SCV kit.

In addition to the single inheritance for hierarchical constraint classes illustrated above, it is possible to use C++ multiple inheritance to compose more than one base constraint classes together. This is useful when you have two or more base constraint classes that should be declared independently of each other, but which need to be combined together in certain cases to model specific tests.

8.4.3 Using constraints from another scv_smart_ptr object

In certain situations it is desirable to dynamically bind constraints to a smart pointer that is not part of a constraint class. The `use_constraint()`method can copy constraints from a source smart pointer that is contained within a constraint class to a target smart pointer that is not within a constraint class.

When a `scv_smart_ptr<>` copies the constraints from another object, all of the constraints associated with that source object are copied (both simple and complex). An important note on this is that when the constraints are copied the values of all constraint solver input variables are copied from the source to the target. The target `scv_smart_ptr<>` will then always use the same values for the constraint solver input variables.

```
Example:  use_constraint() method

class default_packet : public scv_constraint_base {
  public:
    scv_smart_ptr<packet> sp;

    SCV_CONSTRAINT_CTOR(default_packet) {
      SCV_CONSTRAINT(sp->source() <= 0x3);
      SCV_CONSTRAINT(sp->destination() <= 0x3);
      SCV_CONSTRAINT(sp->source() != sp->destination());
    }
};
```

```
class nondefault_packet : public scv_constraint_base {
  public:
    scv_smart_ptr<packet> sp;
    SCV_CONSTRAINT_CTOR(nondefault_packet) {
      SCV_CONSTRAINT(sp->source() <= 0xf);
      SCV_CONSTRAINT(sp->destination() <= 0xf);
      SCV_CONSTRAINT(sp->source() >= sp->destination());
    }
}

enum packet_constraint_type { DEFAULT_PACKET, NONDEFAULT_PACKET };

void constrain_packet(scv_smart_ptr<packet> sp, packet_constraint_type t)
{
  switch(t) {
  case DEFAULT_PACKET: {
    default_packet dp(“dp”);
    sp->use_constraint(dp.sp); //use constraint from the constraint class
    break;
  }
  case NONDEFAULT_PACKET: {
    nondefault_packet np(“np”);
    sp->use_constraint(np.sp);
  }
  //NOTE: since the constraint is copied, it is okay that the constraint
  //object goes out of scope. The input object sp has its own copy of the
  //constraint.
}

void doit() {
  scv_smart_ptr<packet> pkt;
  constrain_packet(pkt, DEFAULT_PACKET);

  for (int i=0; i<5; ++i) {
    pkt->next();
    cout << “pkt is: “ << *pkt << endl;
  }

  constrain_packet(pkt, NONDEFAULT_PACKET);

  for (int i=0; i<5; ++i) {
    pkt->next();
    cout << “pkt is: “ << *pkt << endl;
  }
}
```

Further examples of `use_constraint` are in the `examples/scv/randomization/constraints` directory in the SCV kit.

8.5 Combining Simple Constraints with Complex Constraints

It is possible to mix usage of simple constraints and complex constraints within a constraint class. SCV uses a two step procedure for generating new values when the `next()` method is invoked: In the first step, all simple constraints (which by definition apply only to a single variable) are solved. In the second step, complex constraints (which may establish constraint relationships between multiple variables) are solved using the BDD constraint solver.

This mechanism allows some variables in a constraint expression to use distribution modes and/or simple `keep_only/keep_out` constraints while still affecting the value generation of other variables through complex constraints.

```
Example: Mixing complex constraints with other randomization modes

class my_constraint_class : public scv_constraint_base {
  public:
    scv_smart_ptr<my_packet> p;
    scv_smart_ptr<int> d;

    SCV_CONSTRAINT_CTOR(my_constraint_class) {
      SCV_CONSTRAINT( p->addr() >= d() && p->addr() <= p->data() + 100 );
    }
};

void doit() {
  my_constraint_class c(“c”);
  //use a keep_only/keep_out constraint on length
  c.p->length.keep_only(2, 56);
  c.p->length.keep_out(20, 40);

  //create a distribution for the variable d (that addr is constrained to)
  scv_bag<int> addr_d;
  addr_d.add(10, 80);
  addr_d.add(15, 10);
  addr_d.add(20, 20);

  c.d->set_mode(addr_d);

  // Note: all of the above code could also be placed
  // in the CTOR for my_constraint_class if desired

  for(int i=0; i<10; ++i) {
    // c.d and c.p->length are selected first, then the complex constraint
    // is solved for the other variables.
    c.next();
    do_transaction(*c.p);
  }
}
```

The two step constraint solving procedure within SCV is very powerful because it allows you to mix distribution constraints with complex Boolean constraints in a coherent manner, as the above example illustrates.

However, there may be cases where the two step solving procedure is disadvantageous. Consider the example below:

```
Example: Potentially undesirable two step constraint

class two_step_constraint : public scv_constraint_base {
  public:
    scv_smart_ptr<int> a;

    SCV_CONSTRAINT_CTOR(two_step_constraint) {
      a->keep_only(100, 200);
      SCV_CONSTRAINT( (a() & 0xf) == 0 );
    }
};
```

Ostensibly the above constraint class is constraining “a” to values between 100 and 200 that are multiples of 16. However, because of the two step constraint solving feature of SCV, the `keep_only` constraint will be solved in the first step and may potentially pick a value that is not a multiple of 16. In the second step when the complex constraint is evaluated a constraint solver failure may occur if an illegal value was selected in the first step.

If this behavior is undesirable, the way to avoid it is to express all of the constraints as complex constraints, as shown below. When this is done all of the constraints are evaluated simultaneously (rather than in two steps) so the problem never occurs.

```
  Example: Changing into a one step constraint

  class one_step_constraint : public scv_constraint_base {
    public:
      scv_smart_ptr<int> a;

      SCV_CONSTRAINT_CTOR(one_step_constraint) {
        SCV_CONSTRAINT( a() >= 100 && a() <= 200);
        SCV_CONSTRAINT( (a() & 0xf) == 0 );
      }
  };
```

9. Debugging Unsolvable Complex Constraints

With any constraint mechanism it is possible to over constrain variables such that it is not possible to obtain a solution to the constraint. When an over-constraint occurs, it is often necessary for a user to attempt to figure out what is causing the failure so that he can avoid the over constraint.

There are two typical types of over constraint:

- A constraint expression is created that has no valid solutions. This happens most often with hierarchical constraints where the effect of lower level constraints is not completely considered.

- A variable in a constraint expression is forced to a value at run time that causes the constraint to be unsolvable. This happens when randomization of a variable is disabled and it is set to an incorrect value, or when a distribution mode is used on a variable and the distribution contains some values that cause the complex constraint to be unsolvable. In this case the variable is a constraint solver input variable but the value of the variable allows for no legal solutions to the complex constraint expressions.

9.1.1 Printing the constraint expression

The easiest way to deal with either of these cases is to use the `scv_constraint_base::print()` method. This method prints out the full constraint for the object and also prints out the values of all of the solver input variables. From this information, it is possible for the user to determine where the problem is.

9.1.2 Example of over constrained expression

This example shows an invalid hierarchical constraint expression. The code below prints out the constraint object after it is instantiated so that the problem can be identified.

```
Example: Debugging an over-constrained expression

class base : public scv_constraint_base {
  public:
    scv_smart_ptr<sc_uint<8> > a, b;
    SCV_CONSTRAINT_CTOR(base) {
      //name the variables for debugging
      a->set_name(“a”); b->set_name(“b”);
      //set a constraint
      SCV_CONSTRAINT( a() > b() );
      SCV_CONSTRAINT( b() > 32 && b() < 48);
    }
};
class derived : public base {
  public:
    //create hierarchical constraints.
    SCV_CONSTRAINT_CTOR(derived) {
      SCV_BASE_CONSTRAINT(base);

      //add a constraint that make the base invalid
      SCV_CONSTRAINT( a() < 16 );
    }
};

//this constraint will have an error at instantiation time
void doit() {
  derived c(“c”);
  //print out the constraint expression
  c.print(cout, 1);
}

//Since detail level 1 was given to the print method, it will only print
//the constraint expression. This is all you need when the constraint
//expression is bad.
derived Name: c
  Hard constraints: ((1&&(a<16))&&(1&&(((1&&(a>b))&&
   ((b>32)&&(b<48)))&&1)))
  Soft constraints: (1&&(1&&(1&&1)))
```

When a constraint expression is over constrained, as in the case above, the important aspect of the print out is the expressions for the Soft and Hard constraints. In this case, there are no soft constraints, so the only expression to look at is the Hard constraint expression. The 1’s that are anded with parts of the expression are an internal artifact from the building of the expression in the constraint class. Taking the 1’s out of the Hard constraint yields the expression:

```
((a<16)&&((a>b)&&((b>32)&&(b<48))))
```

Seeing the full expression in this way makes it possible to notice the problem. The variable “b” is >32 and <48. The variable “a” is <16. But, the variable a is >b.

9.1.3 Example of run-time constraint violation

Debugging a run-time violation is similar. However, in this case, the current values of the constraint solver input variables play a key role. A run-time constraint failure can only occur in one of the following scenarios:

- A variable in the constraint expression is disabled via `disable_randomization`, and the current value of the variable causes the constraint to fail.

- A variable in the constraint expression is selected using a distribution mode (either `keep_only/keep_out`, or `set_mode(scv_bag<T> / SCAN / RANDOM_AVOID_DUPLICATE)`), and the value that is selected causes a constraint expression to fail.

```
Example: Debugging a run-time constraint violation

class base : public scv_constraint_base {
  public:
    scv_smart_ptr<sc_uint<8> > a, b;
    SCV_CONSTRAINT_CTOR(base) {
      //name the variables for debugging
      a->set_name("a"); b->set_name("b");
      //set a constraint
      SCV_CONSTRAINT( a() > b() );
      SCV_CONSTRAINT( b() > 32 && b() < 48);
    }
};
class derived : public base {
  public:
    scv_smart_ptr<sc_uint<8> > c; //a new variable

    //create hierarchical constraints.
    SCV_CONSTRAINT_CTOR(derived) {
      SCV_BASE_CONSTRAINT(base);
      c->set_name("c"); //for debugging
      c->disable_randomization(); //use like a constant
      *c = 64; //default value that is valid

      //add a constraint that potentially makes the base invalid
      SCV_CONSTRAINT( a() < c() );
    }
};

//this constraint will only have an error if c is <= 32.
void doit() {
  derived ctr("ctr");
  ctr.next();  //okay because everything is in range

  *ctr.c = 21; //an illegal value
  ctr.next(); //this will cause the runtime error since c is out of range
  //print out the constraint
  ctr.print();
}

//This time we are printing with the default detail level of 0 which
//prints both the expression and the values of all of the variables.
derived Name: ctr
  Hard constraints: ((1&&(a<c))&&(1&&(((1&&(a>b))&&
   ((b>32)&&(b<48)))&&1)))
  Soft constraints: (1&&(1&&(1&&1)))
  Number of elements: 2
  Current value of elements:
    a:  88
    b:  153
    c:  21
```

This is the same over-constraint as in the previous section. However, in this case, the error is due to the fact that the variable c is being explicitly disabled and set to an invalid value. In this case, to determine the cause of the problem you must look at both the expression and the current value. In this case, the current value of c=21 causes the constraint to be invalid.

As a rule it is better to have a large number of small constraint classes, rather than a small number of large constraint classes. This helps improve both constraint solver performance and code clarity, and makes it easier to debug constraint solver failures when they occur.

10. Controlling Random Value Generation

Throughout this paper we have alluded to the concept of controlling value generation. The ability to control the generation of values is critical in order to produce useful sets of random values. In SCV, there are four primary mechanisms for controlling value generation:

- enable/disable randomization for specific variables and/or fields of variables
- do pre-generation or post-generation actions when values are generated
- set specific generation modes
- set initial seed values for random variables

10.1 Enabling and disabling randomization

When a constraint class is used, all random variables (`scv_smart_ptr<>` types and any subfields of the underlying data types) are randomized by default. The user may change this default behavior by specifically disabling randomization using the `disable_randomization()` method. And, randomization can be re-enabled at any time using the `enable_randomization()` method. The common uses of the disable randomization method are:

- Provide runtime parameterization in a constraint expression to be modified by the user of the constraint object.
- Force a value on an object or object field, to direct some specific behavior.

Note that when you invoke the `next()` method on a subfield of a constraint class or a `scv_smart_ptr` rather than on the top level object, this is equivalent to temporarily setting `disable_randomization` on the non-included subfields and then invoking `next()` on the top level object.

10.1.1 Constraint parameterization

It is often desirable to have a constraint expression in which some variables are used as inputs to the constraint solver. For example, it may be necessary to allow the user to specify some address range to a constraint expression. The example below demonstrates how this can be done.

```
Example: Dynamic parameterization of constraints

class a_constraint : public scv_constraint_base {
  public:
    scv_smart_ptr<sc_uint<10> > dest_addr, offset_addr;
  protected: //allow access from derived classes, but not for users
    scv_smart_ptr<sc_uint<10> > min, max;

  public:
    SCV_CONSTRAINT_CTOR(a_constraint) {
      //name the variables for debugging
      dest_addr->set_name(“dest_addr”);
      offset_addr->set_name(“offset_addr”);

      min->set_name(“min”);
      max->set_name(“max”);

      //set the default min/max
      *min = 0;
      *max = 0x400;

      //disable randomization on min/max, they can be set by the
      //constraint user
      min->disable_randomization();
      max->disable_randomization();

      //set a constraint
      SCV_CONSTRAINT( dest_addr() >= min() && dest_addr() <= max() );
      SCV_CONSTRAINT( src_addr() == (dest_addr() + max()));
    }

  public:
    void set_range(unsigned int minimum, unsigned int maximum) {
       *min = minimum; *max = maximum;
    }
    unsigned min_addr() { return *min; }
    unsigned max_addr() { return *max; }
};

void doit() {
  //using the constraint
  a_constraint a(“a”);

  //set the range to use
  a.set_range(0x10, 0x100);

  for(int i=0; i<5; ++i) {
    a.next();
    a.print(cout, 2);  //just print the values
  }

  //set a new range to use
  a.set_range(0x110, 0x200);

   for(int i=0; i<5; ++i) {
    a.next();
    a.print(cout, 2);  //just print the values
  }
}
```

10.1.2 Enabling and disabling parts of complex constraints

Sometimes you may wish to enable or disable specific parts of a complex constraint expression. For example you may have a test that has specific modes, and in each of these modes different complex constraints should be used. Complex constraints cannot be dynamically added or deleted in SCV (since they are statically created when a constraint class is instantiated), but there is an easy way to dynamically disable and enable parts of constraint expressions as shown in the example below.

```
Example: Enable and disable parts of complex constraints

enum addr_modes {LOW, HI, ALL};

// scv_extensions<> for addr_modes not shown

#define if_then(a, b) (!(a) || (b))
#define if_then_else(a, b, c) ((!(a) || (b)) && ((a) || (c)))

class if_constraint : public scv_constraint_base {
  public:
    scv_smart_ptr<unsigned> addr, data;
    scv_smart_ptr<addr_modes> mode;

    SCV_CONSTRAINT_CTOR(if_constraint) {
      *mode = ALL;
      mode->disable_randomization();

      SCV_CONSTRAINT(if_then(mode()==LOW, addr() <= 0xffff && data()< 8));
      SCV_CONSTRAINT(if_then(mode()==HI , addr() >  0xffff && data()> 8));
      SCV_CONSTRAINT(if_then(mode()==ALL, addr() <= 0xffffffff) );
    }

    unsigned set_mode(addr_mode mode_) { mode = mode_; }
};
```

The `if_then()` and `if_then_else()` constructs illustrated above can be considered as a bidirectional implication constructs that can be used within complex constraints. Though this example is quite simple, this same approach can be extended for much larger and more elaborate constraints.

10.1.3 Forcing a value

In the `a_constraint` example above, the constraint class was responsible for setting the randomization of the `min/max` variables. The `min/max` variables cannot be accessed by the end user because they are declared as protected members. (Because they are protected members derived constraint classes can still access them).

The other use of `disable_randomization()` is by the test writer. In this case it is used when the user of the constraints wants to force specific variables (or fields).

In this case, randomization can be disabled when desired, and reenabled using `enable_randomization()` when a variable is no longer being forced.

```
Example: Forcing constraint variables to specific values

class constraint : public scv_constraint_base {
  public:
    scv_smart_ptr<sc_uint<8> > a, b, c;
```

```
      SCV_CONSTRAINT_CTOR(constraint) {
        //name the variables for debugging
        a->set_name(“a”);
        b->set_name(“b”);
        c->set_name(“c”);

        SCV_CONSTRAINT( a() > 1 && b() > 1 );
        SCV_CONSTRAINT( a() * b() == c() );
      }
  };

  void doit() {
    //using the constraint
    constraint cnst(“cnst”);

    // first use the constraint to find some
    // non-prime values for “c” that are less than 256

    cnst.next(); cnst.print(cout, 2);
    cnst.next(); cnst.print(cout, 2);
    cnst.next(); cnst.print(cout, 2);

    // now let’s disable randomization for “c” and set it to 180.
    // The solver will then find values for a and b that are factors of 180

    cnst.c->disable_randomization();
    *cnst.c = 180;

    cnst.next(); cnst.print(cout, 2);
    cnst.next(); cnst.print(cout, 2);
    cnst.next(); cnst.print(cout, 2);

    // now let’s re-enable randomization for “c” and let’s disable
    // randomization for “a”. Whatever value was last generated for “a”
    // will now “stick” and now values will be generated for “c” that
    // are multiples of the “stuck” value of “a”

    cnst.c->enable_randomization();
    cnst.a->disable_randomization();

    cnst.next(); cnst.print(cout, 2);
    cnst.next(); cnst.print(cout, 2);
    cnst.next(); cnst.print(cout, 2);
  }
```

The techniques illustrated above could be used in the following way: In a testbench that is stimulating a design, you might start out with certain variables (e.g. a memory address) in which randomization is enabled. Then you could monitor the design to see when it reaches some “corner-case” of interest (e.g. a FIFO is nearly full on some memory mapped peripheral). At that point you might disable randomization on the address variable in order to focus stimulus on that device so you can thoroughly test the “corner-case”.

10.2 Customized pre-generation and post-generation actions

In the SCV constraint base class `scv_constraint_base`, the value generation method `next()` is a virtual method. This means that the user is able to override the meaning of `next()` in his derived constraint class. Control of the `next()` method provides the user with a great deal of control over how variables are randomized.

There are several reasons that a user may wish to overload `next()`:

- To update a variable used within a constraint expression prior to the constraint expression being evaluated
- To store the current values of randomizable objects to other variables so that they can affect the generation of future values
- To perform post-generation calculations or actions
- To customize the value generation algorithm

As we will see in the following examples, when you define your own implementation of `next()` within your constraint class, you first perform any necessary pre-generation work, then you call the `next()` method of the base class for your constraint class, and then you perform any needed post-generation work. If your constraint class inherits directly from `scv_constraint_base` then you simply call `scv_constraint_base::next()` to invoke the next() method of your base class. However if your constraint class inherits from some other constraint class then you should call the `next()` method of the base class that you directly inherit from.

10.2.1 Updating the current value of a variable

Occasionally there are cases where it is necessary to constrain a data object based on the value of another data object that is not a `scv_smart_ptr`<> object within the same constraint class. In this case, it is necessary to first copy the value of the other object into a `scv_smart_ptr`<> object within the same constraint class, and then perform the constraint evaluation. Two typical examples of this would be: 1) the use of sc_signal<> objects in constraints, and 2) the use of pointer (or handle) objects in constraints.

```
Example: Overloaded next() and updating current state values

#define if_then(a, b) (!(a) || (b))
#define if_then_else(a, b, c) ((!(a) || (b)) && ((a) || (c)))

class constraint : public scv_constraint_base {
  private:
    //signal variable is attached by the test
    sc_in<sc_uint<8> >* d_in;
    sc_out<sc_uint<8> >* d_out;
    scv_smart_ptr<sc_uint<8> > d_in_obj;
    scv_smart_ptr<sc_uint<8> > d_out_obj;

  public:
    SCV_CONSTRAINT_CTOR(constraint) {
      //name the variables for debugging
      d_out_obj->set_name(“d_out”);
      d_in_obj->set_name(“d_in”);
      d_in = NULL;
      d_out = NULL;

      //d_in_obj will take the value of d_in prior to randomization

      d_in_obj->disable_randomization();
```

```
        // set constraint that if din < 100,
        // then dout < 100
        // else dout >= 100

        SCV_CONSTRAINT( if_then_else(din_obj() < 100,
                                     dout_obj()  < 100,
                                     dout_obj() >= 100 ) );

        void bind_din ( sc_in<sc_uint<8> >& port ) { d_in = &port; }
        void bind_dout ( sc_out<sc_uint<8> >& port ) { d_out = &port; }


        //do the randomization
        void next() {
            //if dout isn’t bound, then there is nothing to do here
            if(d_out == NULL) return;

            //if din is bound, then use it
            if(d_in != NULL) d_in_obj = d_in->read();

            //call the next() method of the base class
            scv_constraint_base::next();

            //write the dout signal with the newly generated value
            d_out->write(d_out_obj->read());
          }
  };

  //this constraint class could be used in an sc_module as shown
  SC_MODULE(mymod) {
    sc_in<sc_uint<8> > data_in;
    sc_out<sc_uint<8> > data_out;
    constraint c;
    SC_CTOR(mymod) : data_in(“data_in”), data_out(“data_out”), c(“c”) {
       c.bind_din(data_in[0]);
       c.bind_dout(data_out[0]);
       SC_METHOD(doit);
       sensitive << data_in;
    }
    void doit() {
      //everytime data_in changes value, a new value is calculated for
      //data_out
      c.next();
    }
  }
```

10.2.2 Generating random values that are dependent on previously generated values

Suppose that you want to generate randomized packets but that you want to make sure that the destination address of each packet is different than the destination address of the ten packets that were previously generated. This can be easily achieved in SCV by storing the previous destination addresses in a separate variable and then by establishing constraint relationships between those previous destination addresses and the current destination address that is to be generated. In the example below we use a pre-generation action to store the previous N destination addresses.

```
  Example: Overloaded next() to store previous generated values
```

```
 template <int N>
class constraint : public scv_constraint_base {
  public:
    scv_smart_ptr<my_packet> p;
    scv_smart_ptr<unsigned>  previous_dests[N];

  public:
    SCV_CONSTRAINT_CTOR(constraint) {
      for (i=0; i<N; i++) {
        previous_dests[i].disable_randomization();
        SCV_CONSTRAINT( p->dest() != previous_dests[i]() );
      }

      //do the randomization
      void next() {
          sc_assert(N > 0);
          for (i=N-2; i >= 0; i--)
            *previous_dests[i+1] = *previous_dests[i];

          *previous_dests[0] = p->dest.read();

          //call the next() method of the base class
          scv_constraint_base::next();
     }
};
```

10.2.3 Performing post-generation actions

Oftentimes the values of some variables are calculated based on the values of other variables. Examples of these types of calculations would include a crc value, or a dynamically sized buffer for a communication payload. The same general methodology as used for pre-generation work is done for this type of post-generation work. The only difference is that the call to `scv_constraint_base::next()` is done prior to the post calculation work.

In the example below, two post-generator activities are done, the first is to generate and randomize a dynamically-sized data buffer, and the second is to perform a crc calculation.

```
Example: Overloaded next for post-generation calculations

struct packet_t {
  sc_uint<32> src, dest;
  sc_uint<8>  length;
  sc_uint<32> crc;
  char *data_buffer;
  packet_t() : data_buffer(NULL) {}
  void create_buffer() {
    if(data_buffer) delete[] data_buffer;
    data_buffer = new char[length.to_uint()];
  }
  packet_t& operator=(const packet_t& rhs) {
    src = rhs.src; dest = rhs.dest; length = rhs.length; crc = rhs.crc;
    if(!rhs.data_buffer) data_buffer = NULL;
    else {
      create_buffer();
      memcpy(data_buffer, rhs.data_buffer, length.to_uint());
```

```
        }
        return *this;
      }
      packet_t(const packet_t& rhs) {
        *this = rhs;
      }
      void calculate_crc() {
        if(!data_buffer) { crc = 0; return; }
        //do the crc calculation based on the data_buffer
        …
      }
    };

    // scv_extensions<> for packet_t not shown here

    class constraint : public scv_constraint_base {

      public:
        scv_smart_ptr<packet_t> p;
        scv_smart_ptr<unsigned char> data_gen;

        SCV_CONSTRAINT_CTOR(constraint) {
          SCV_CONSTRAINT( p->src() != p->dest() );
          SCV_CONSTRAINT( p->src() < 0x10000  || p->dest() < 0x10000 ) ;
          p->length.keep_only(2, 128) ;
        }

        void next() {
              //call the next() method of the base class
              scv_constraint_base::next();

              // create and randomize the data buffer
              p->get_instance()->create_buffer();

              for(int i=0; i<p->length; ++i) {
                data_gen->next();
                p->get_instance()->data_buffer[i] = *data_gen;
              }

              p->get_instance()->calculate_crc();
            }
    };
```

The above example shows the concept of overloading `next()` in order to change the way value generation occurs. In this case, the `data_buffer` is generated separately after the rest of the packet has been created. Generation of the `data_buffer` is dependent on one of the fields in the packet, so the buffer can’t be created until the rest of the packet generation is done.

Note that in the above example `data_gen` is used to create values to store in the variable sized `data_buffer`. One nice aspect of this approach is that it is possible to apply complex constraints to `data_gen` and thus have the constrained values still be used within `data_buffer` even though `data_buffer` is a dynamic, variable sized data structure.

10.3 Random stability and seed management

The last important detail to consider for randomization is the control of the seed values used by the random objects. Each randomizable object instance has its own independent value generator (`scv_random` object). For constraint class instances, all of the randomizable objects (`scv_smart_ptr<>` objects) share a single value generator by default. All `scv_smart_ptr<>` objects which are not part of a constraint object have an independent value generator by default.

The algorithm for initializing `scv_random` value generator seeds is such that simulation runs will produce the same results by default. This is sometimes referred to as "random stability" and is an important requirement when using randomization in verification, because it is important that it is possible to reproduce stimulus from a given testbench across different simulation runs, different simulator versions which might utilize different thread orderings, and even if modifications are made to the testbench or design.

By default the seed value for a generator is based on the full hierarchical name of the SystemC process thread that the `scv_random` object is instantiated in, as well as the order in which the object is instantiated within that SystemC process, as well as the value of the global seed. Random number streams in SCV are thus stable by default because each of these separate seed components is stable across 1) different simulation runs, 2) different simulator versions which might utilize different thread orderings, and 3) even when modifications are made to the design or testbench.

The global seed can be set by using the `scv_random::set_global_seed()` static method. Changing this seed value allows multiple simulation runs of the same design and testbench to produce different results, since by default this single global seed will affect the seed values used for all random generators in SCV. The example below shows how the global seed generation algorithm can be used to create random sets, while associating some random generators with explicit seed values.

```
Example: Managing seed values to control random values

SC_MODULE(rand_data) {
  sc_out<sc_uint<8> > d1, d2, d3;
  scv_smart_ptr<sc_uint<8> > r1, r2, r3;
  SC_CTOR(rand_data) : d1("d1"), d2("d2"), d2("d3"),
    r1("r1"), r2("r2"), r3("r3"),
  {
    // set the global default seed based on the first
    // command line argument which is assumed to be an integer
    if (sc_argc() > 1)
      scv_random::set_global_seed(atoi(sc_argv()[1]));

    /*create a generator with an explicit seed of 101*/
    scv_shared_ptr<scv_random> rgen = new scv_random ("rgen", 101)

    /*set the r2 smart pointer to use the new generator*/
    r2->set_random(rgen);

    // random number streams for r1 and r3 will now be influenced
    // by the global seed specified via the seed argument, while
    // the random number stream for r2 will be completely stable and
    // not be influenced by changes to the global seed.

    SC_THREAD(doit);
  }
  void doit() {
    for(int i=0; i<20; ++i) {
      wait(20,SC_NS);
      r1->next();
      r2->next();
      r3->next();
      d1 = r1->read();
      d2 = r2->read();
      d3 = r3->read();
    }
  }
};
```

SCV also provides functions to easily write the `scv_random` object seeds that are used in a given simulation run out to a file, and functions to read seed values from the file for initialization of `scv_random` objects. See the “SystemC Verification Standard Specification” within the SCV `docs/scv/scvref` directory for further details.

By default SCV uses a 48 bit random number generator, which is the `jrand48` standard algorithm. This is a very high quality randomization algorithm that provides uniform results, while using a large seed value so that random number streams are not repeated for a very long time. You can select among different randomization algorithms or specify your own randomization algorithm to use. See the “SystemC Verification Standard Specification” for further details.

11. Other Considerations

This section describes some other things that a user should be aware of when using the randomization facilities of SCV.

11.1 Deep recursion (thread stack size)

The constraint solver can occasionally use many levels of recursion when solving a constraint. The depth of recursion depends on the number of variables in the constraint and the types of expressions in the constraint.

The reason that this is important for a user is because deep recursion can require a fairly significant stack in order to avoid a segmentation fault. In SystemC, the default stack size for a thread is small (it is operating system dependent, but generally around 64 Kbytes). Occasionally even a fairly simple constraint can have sufficient recursion to crash if used in a SystemC thread with the default stack size.

To work around this, a user should do one of two things:

- Only call `scv_constraint_base::next()` from an SC_METHOD.
- Set the stack size of an SC_THREAD to a large enough value to handle constraint solving.

11.1.1 Calling next from a method

SC_METHODS run on the main program stack, so the stack size is conceptually unlimited (it is limited only by the amount of virtual memory available to the process). So, calling next() from a method is always safe. A major drawback of this approach is that it may be necessary to set up some kind of special synchronization with the method in order to communicate with a thread that needs the new values. The example below shows how a method may be used to generate values for a thread.

```
Example: Calling next() from a method

SC_MODULE(mymod) {
  protected:
    //events for the method and thread to communicate
    sc_event do_next_e;
    sc_event next_done_e;

  public:
    //a constraint object
    my_constraint c;

  public:
    SC_CTOR(mymod) : c(“c”) {
      SC_THREAD(doit);    //thread process that needs values
      SC_METHOD(donext);  //method process to generate values
      sensitive << do_next_e;  //responds to the do_next_e event
    }

  protected:
    void doit() {
      while(1) {
        //request to generate a value
        do_next_e.notify();
        //wait for the next to finish
        wait(next_done_e);
        //use the randomized value
        …
      }
    }
    void donext() {
      c.next();  //this is a method that runs on the main stack
    }
};
```

11.1.2 Enlarging the stack size

Running in a method solves the stack size problem, but as the above example shows, this can be tedious. Since it is often the case that a thread process needs to call `next()` on a constraint, the better solution for this case is to increase the stack size. Below the example presented above is rewritten to use a larger thread stack instead of running on the main stack.

```
Example: Enlarging the stack size

SC_MODULE(mymod) {
  public:
    //a constraint object
    my_constraint c;

  public:
    SC_CTOR(mymod) : c(“c”) {
      SC_THREAD(doit);    //thread process that needs values
      set_stack_size(0x100000);  //set stack to 1MB
    }

  protected:
    void doit() {
      while(1) {
        //generate the next value, stack is 1MB, so it is big enough
        c.next();

        //use the randomized value
```

```
            …
          }
        }
    };
```

The approach presented above is usually the best strategy for ensuring that the constraint solver has adequate stack space. Usually even complex testbenches with a large number of distinct threads (e.g. fifty or even more) can use this approach. Most of the reserved stack space may not be used and will stay swapped out of physical memory. However you should make sure your machine has adequate swap space (usually greater than 200 Mbytes).

11.2 Compilation Speed

SCV makes heavy use of C++ templates, and as a result source files that include `scv.h` can sometimes take a long time to compile compared to typical C++ files. Here are several tips for enhancing compilation speed with SCV:

- Compile on a machine that has at least 256 Mbytes of physical memory and at least 512Mbytes of swap space.

- Make sure your compiler is up-to-date. C++ compilers have made significant improvements in template capabilities recently, and newer compiler versions tend to be significantly faster than older versions. If you are experiencing compilation speed problems, you may want to check whether patches or updates are available from your compiler vendor.

- Only include `scv.h` in source files that require it.

- It is usually better to have a larger number of relatively small files that include `scv.h` rather than a small number of big files that include `scv.h`. By following this approach only a small amount of code needs to be recompiled when you make changes.

- Turn off the compiler's optimizer and code inlining for files that include scv.h. For the GNU g++ compiler the flags to do this are `"-O0 –fno-default-inline –g"` . The "`-g`" option enables source code debugging, which you will often want enabled anyway, and which has the effect of disabling many optimizations, thus increasing compilation speed.

- If compilation speed is still a concern, consider trying a different compiler. There are large differences in compilation speed for SCV files for different C++ compilers.