FC4SC User's Guide

Revision History

Version	Date	Comments
1.0	20.02.2018	First release.
1.1	04.09.2018	Minor text/code corrections and typo fixes.
1.2	03.10.2018	Updates to the covergroup/coverpoint section related to the new syntax for creating coverpoints.
1.3	02.11.2018	Added description for the bin arrays under the *Coverage Definition -> Bins* section. Also updated the Roadmap.

About FC4SC

FC4SC stands for Functional Coverage for SystemC. This library provides a functional coverage collection mechanism similar to the one in SystemVerilog (see this table for similarities and differences).

If you want to know more about the functional coverage you can read Section 19, of the IEEE-1800 SystemVerilog standard[1]. The next excerpt should be revealing what functional coverage is:

Functional coverage is a user-defined metric that measures how much of the design specification has been exercised. It can be used to measure whether interesting scenarios, corner cases, specification invariants, or other applicable design conditions have been observed, validated, and tested.

Installation and Integration

First step is to download the source files from the dedicated GitHub repo.

FC4SC is a C++11, header only, library. The self checking tests are using googletest.

The API documentation can be generated using doxygen:

```
$> cd /path/to/fc4sc/doc
$> doxygen
```

Include the main header (i.e. *fc.hpp*) in your sources to use FC4SC.

For example in my_file.cpp you should add next line:

```
// other headers...
#include "fc4sc.hpp"

// coverage definitions
```

and the following arguments when compiling (requires C++-11):

```
${CXX} my_file.cpp ${FLAGS} -I path_to_fc4sc/includes -std=c++11
```

Since the library relies heavily on templates you might see a small increase in compilation times of your application.

Coverage Definitions

Covergroups

The covergroup class is the core element of a functional coverage model and represents a construct that encapsulates a set of coverpoints and crosses.

To create a covergroup, you have to:

1) Define a class which extends fc4sc::covergroup.
2) Register your covergroup in the library in order for the collected coverage information to appear in the coverage database. (*)
3) Declare the coverpoints and/or crosses as members of the covergroup.

Example:

```
#include "fc4sc.hpp"

class my_first_cvg : public covergroup {
```

```
    CG_CONS(my_first_cvg) {
      // set options or type options for covergroup and nested coverpoints
    }
    // coverpoints and crosses instantiation
};
```

The "*CG_CONS*" macro declares a constructor which delegates the *covergroup* constructor with relevant contextual information in order to register this instance in the FC4SC library. Thus, the following

```
CG_CONS(my_first_cvg)
```

gets expanded to:

```
using covergroup::sample(); // required in order to use "cvg_inst.sample()"
my_first_cvg(string inst_name="") : fc4sc::covergroup("my_first_cvg",
__FILE__, __LINE__, inst_name)
```

All this information is useful when generating the coverage report. You can skip the macro and just call the parent constructor with the type given as argument and that would register the instance :

```
my_first_cvg(string inst_name="") : fc4sc::covergroup("my_first_cvg")
```

Notice that the instance name is optional. That means that if you want to pass additional parameters, you will need to either provide default values or declare another constructor that calls this above one.

Example: default values:

```
CG_CONS(my_first_cvg, int weight=1, int some_param=42) {
    // handle arguments
}
```

Or just another constructor:

```
CG_CONS(my_first_cvg) {};

my_first_cvg(string inst_name, int param) : my_first_cvg(inst_name) {
    // handle param
}
```

Lastly, the covergroup also provides a default *sample()* function which automatically samples all the coverpoint elements nested inside it.

Coverpoints

A coverpoint defines the following:

1. an expression to be sampled
2. a collection of bins containing values to be sampled
3. optionally, a boolean expression which conditions sampling

NOTE: The Old coverpoint syntax is now deprecated and will be removed in the future because of the following reasons:

1) It forced the user to manually bind a sample variable to the coverpoint (using *SAMPLE_POINT(...)*), which is error prone
2) A segmentation fault would be generated if *SAMPLE_POINT(...);* was declared after its associated coverpoint is instantiated
3) It forced sampling of the coverpoint to be made on a variable, not supporting complex/compound expressions

The new way to declare a coverpoint makes use of the *COVERPOINT* macro and is cleaner and similar to the coverpoint definition in the SystemVerilog standard. To create a coverpoint you need to specify:

1. The data type of the expression to be sampled
2. The name of the coverpoint
3. The expression to be sampled
4. A list (can be empty) of bins templated by the same type as the type of the sample expression.

The first 3 requirements are passed as arguments to the *COVERPOINT* macro and the list of bins is contained in curly brackets, directly following the *COVERPOINT* macro.

Example:

```
COVERPOINT(int, data_ready_cvp, sample_expression) {
    illegal_bin<int>("illegal_zero", 0),
    bin<int>("positive", interval(1,INT_MAX)),
    bin<int>("negative", interval(-1,INT_MIN))
};
```

This will create a coverpoint of type *int* with the name "data_ready_cvp" which will collect data returned by evaluating the *sample_expression* each time the parent covergroup is sampled.

Additionally, a sample condition can be also provided. When a sample condition exists, the coverpoint will only be sampled if the condition evaluates to *true*.

Example:

```
COVERPOINT(int, data_ready_cvp, sample_expression, sample_expression != 0) {
    bin<int>("positive", interval(1,INT_MAX)),
    bin<int>("negative", interval(-1,INT_MIN))
};
```

This coverpoint definition is the same as the previous one, without the illegal bin and with the additional constraint that sampling (for this coverpoint) will only take place when *sample_expression* evaluates to non zero.

Crosses

A cross is the cartesian product of its member coverpoints' bins. This means that sampling is done on multiple values (1 for each coverpoint), and that a hit happens when each sampled value is present in its coverpoint (i.e. for the first value, the first given coverpoint has a bin containing that value etc.).

```
template <typename ...T>
class cross: public cvp_base
```

Crosses have similar behavior to coverpoints, main difference being that that they are defined based on coverpoint instances rather than bins.

To create a cross you need to specify:

1. A pointer to the covergroup it belongs to
2. A name (optional)
3. 0 or more coverpoints, templated by the same types (see example)

Example:

```
// Inside a covergroup declaration
auto some_cross = cross<int, double> (
    this,
    &my_int_cvp,        // decltype(my_int_cvp) == coverpoint<int>
    &my_double_cvp,     // decltype(my_double_cvp) == coverpoint<double>
);
```

Bins

A bin is a named collection of interesting values and intervals defined by the user. Each bin has a "hit counter" that keeps track of how many times the bin was hit. A bin is considered to be "hit"

when it is sampled with a value that is either directly contained inside it, or inside any interval contained by the bin.
Multiple types of bins exist, each with a different behavior (explained in the following subsections), but all are defined in the same way and obey the same definition rules as the regular bin type.

Regular Bin

A regular bin is templated by the type of values/intervals it holds:

```
template <typename T>
class bin : public bin_base
```

To create a regular bin you need to specify:

- A name
- A list of values or intervals of the same type used in instantiating the bin:
  - *T* for a single value
  - *interval(T a, T b)* for an interval of values

Example:

```
// Create a bin named "power_of_2" with value {1} U [2:3] U [4:7] U [8:15]
// Note: interval arguments order doesn't matter
auto my_bin = bin<int>(
    "power_of_2",
    1,              // {1}
    interval(2,3),  // [2:3]
    interval(7,4),  // [4:7]
    interval(15,8)  // [8:15]
);
```

Restrictions:

- The bin name ***must*** be specified exactly once for all bin instances and must be the first argument to the constructor
- The bin ***must*** contain at least one value/interval

The code will not compile unless the above restrictions are met!

Bin Arrays

Bin arrays can be used to define coverage in a flexible manner, without having the user explicitly state all the bins. There are 2 typical cases for which bin arrays can be used:

1) Splitting an interval into multiple equal pieces

Arguments that you need to specify:

- A base bin name
- The number of subintervals that the interval should be split into
- The interval to be split

Example:

```
auto bin_ar = bin_array<int>("split", 3, interval(0,9));
// expands to 3 bins (when used in a coverpoint):
//   split[0] -> [0:2]
//   split[1] -> [3:5]
//   split[2] -> [6:9]
```

Note1: if the interval cannot be split into equal size subintervals, the last bin will contain the extra elements!
Note2: If the specified number of subintervals is bigger than the number of elements in the interval to be split, the *bin_array* will be expanded to only 1 bin containing the whole interval.

2) Creating bins in a dynamic way

Bin arrays can also construct bins from a *std::vector* parametrized by either regular values, or intervals of any primitive type. This can be used together with a function which returns a *std::vector* containing the coverage data that you want to create the bins from.

Example:

```
// returns a vector of N fibonacci numbers generated by starting with 1 and
2
auto fibonacci = [](size_t N) -> std::vector<int> {
  int f0 = 1, f1 = 2;
  std::vector<int> result(N, f0);
  for (size_t i = 1; i < N; i++) {
    std::swap(f0, f1);
    result[i] = f0;
    f1 += f0;
  }
  return result;
};
// this bin_array expands to 5 different bins for values: 1, 2, 3, 5, 8
auto fibonacci = bin_array<int>("fibonacci", fibonacci(5));
```

Similarly, a *std::vector<fc4sc::interval_t<T>>* can be passed to a *bin_array<T>* in order to create bins containing intervals.

Illegal Bin

You can also create *illegal_bin*. When these get sampled, an exception is thrown and the simulation will end. You can pass a flag (i.e. do a define) to the compiler to disable this behaviour (the exception will be caught and an error message will be printed). Other than that, illegal bins are used in the same way as regular bins, and they don't contribute to coverage.

Example:

```
auto my_illegal_bin = illegal_bin<int>("illegal_zero", 0);
```

Ignore Bin

Another type of bin is *ignore_bin*. This doesn't change overall coverage (i.e. hits on ignored bins don't increase coverage percentage) and it doesn't show up in the report. It can be used with crosses to ignore certain configurations.

```
auto bin = ignore_bin<int>("uninteresting", 0);
```

At sample time FC4SC tries to identify a value as belonging to ignore_bins, then to illegal_bins and last to "regular" bins category.

Tying everything together

All that's left now is to put everything together. Below is a complete example of a covergroup containing 2 coverpoints and a cross between them.

```
class my_first_cvg : public covergroup {
public:
    int value = 0;
    int flags = 0;
    CG_CONS(my_first_cvg) {}

    COVERPOINT(int, values_cvp, value) {
        bin<int>("low1", interval(1,6), 7), // intervals are inclusive
        bin<int>("med1", interval(10,16), 17),
        bin<int>("med2", interval(20,26), 27),
        bin<int>("high", interval(30,36), 37)
    };

    COVERPOINT(int, flags_cvp, flags) {
        bin<int>("zero", 0),
        bin<int>("one", 1),
        bin<int>("ten", 10),
```

```
        illegal_bin<int>("some_illegal_config", 3),
        ignore_bin<int>("uninteresting", 8)
    };


    // Cross (cartesian product) of the two coverpoints
    auto valid_data_cross = cross<int, int> (this,
        &flags_cvp, &values_cvp
    );
};
```

In the above example, *values_cvp* will sample the value of the *value* variable and *flags_cvp*, the *flags* variable. All that's left to do is to trigger the sampling by calling *sample()* whenever desired. Note that the sample expressions are always evaluated at the call to the *covergroup::sample()*, which means that you have to make sure the variables are properly set before that.

Example:

```
my_first_cvg cvg_inst;   // instantiate the covergroup
cvg_inst.value = 15;     // assign values to the variables
cvg_inst.flags = 10;
cvg_inst.sample();       // sample the values
```

Compilation flags

Disabling coverage

Compile with *-DFC4SC_DISABLE_SAMPLING* to disable all sampling.

Continue on illegal hit

As mentioned above, by default, a simulation will stop on an illegal bin hit. To keep the simulation running, compile with *-DFC4SC_NO_THROW* . You will still get an error message on each hit.

Note: the report is not generated if the simulation stops early (i.e. on a hit of an illegal sample)

FC4SC API

Each coverage item offers the same functions that SystemVerilog provides with few exceptions (see also this table for similarities).

Function	Exception Description
void sample()	-
double get_inst_coverage()	-
double get_inst_coverage(int&, int&)	-
void set_inst_name(const string&)	-
void start()	-
void stop()	-
static double get_coverage(const string& type)	Since you can't inherit static methods, custom covergroups won't have the get_coverage(...) methods specific to their types (i.e. all covergroups will have the same method). To bypass this, you can call the functions from a global object fc4sc::global_access::get_coverage(const string& type)
static double get_coverage(const string& type, int&, int&)	fc4sc::global_access::get_coverage(const string& type, int&, int&)

UCISDB/XML Support

The generated report is written in XML format, respecting the UCIS Schema[2] (see UCIS Standard, Chapter 9.8).

To generate such a file, call:

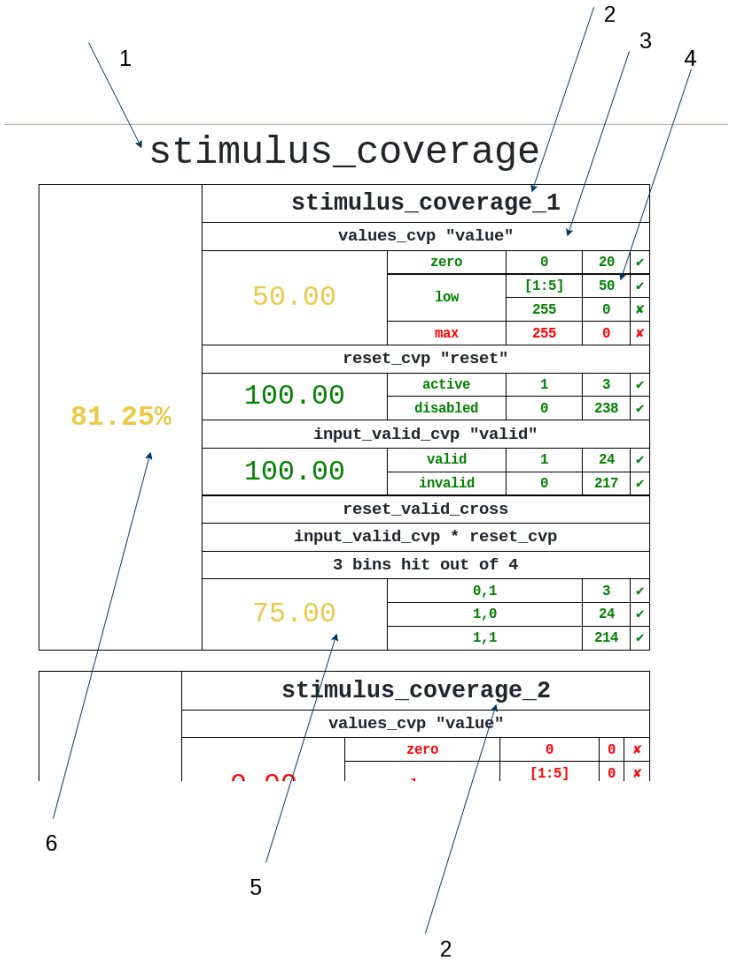
```
fc4sc::global::coverage_save (std::ofstream& stream)
```

```
fc4sc::global::coverage_save (const std::string& file_name)
```

Reporting and Visualizing Coverage

The collected functional coverage is saved to an UCISDB/XML file which can be loaded by an HTML/JavaScript application to present the collected data into a human-, analysis-friendly way.

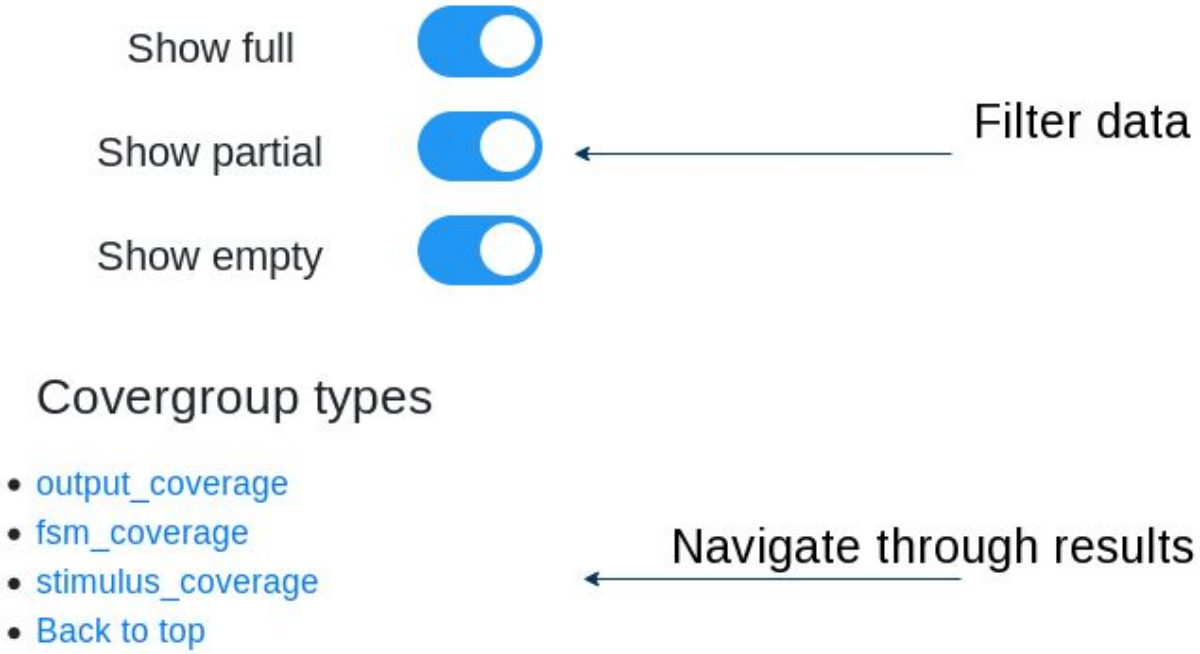
Open fc4sc/report/index.html and load the file:



Explanations:

1. Covergroup type (i.e. name of your class)
2. Instances of that type
3. Coverpoints of that covergroup
4. Bins of that coverpoint and their hitcount
5. Coverage percent of the coverpoint
6. Coverage percent of the covergroup instance

You can navigate or filter the coverage results using the menus:



Running unit tests

The library is tested using the googletest distribution. In order to be able to run the unit tests, the following steps are required (we recommend version 1.8.0, as it was confirmed to work with FC4SC):

1. Download & extract the googletest framework. The FC4SC provides a script for automating this process

```
cd fc4sc/test
./fetch-googletest.sh
```

2. Build the googletest library. After downloading, from the `fc4sc/test` directory, run:

```
cd googletest
mkdir build && cd build
cmake -DBUILD_GTEST=ON BUILD_GTEST -DBUILD_SHARED_LIBS=ON ..
make
```

3. Now that the googletest framework has been built, FC4SC unit tests can be compiled and run:

```
cd test/fc4sc # cd to the FC4SC unit test directory
make # compiles the unit tests
make run # runs the unit tests
```

Running examples

FC4SC also comes with a SystemC example design that was modified to collect functional coverage.

To run:

1. Go to examples dir:

```
$> cd examples/fir
```

2. Build and run :

```
$> make
$> make run
```

Note: You must have a *SYSTEMC_HOME* variable set to where you installed SystemC
Note: Results will be written in *coverage_results.xml*

Roadmap

Here is a list of enhancements that are planned for future releases.

- Merge of different coverage databases
- Better filtering in crosses (binsof , intersect)
- Move to other output format. The current implementation uses the UCIS format in order to be compatible with 3rd party vendors of functional coverage tools. In case you don't need this format other data formats (e.g. json) are more desirable.
- Automated translation of SystemVerilog coverage definitions. This is a nice to have for SystemC models that are used for verification purposes and which can follow the same functional coverage model.

References

[1] IEEE 1800 - 2012 SystemVerilog Standard
[2] Chapter 9.6, UCIS Standard
[3] How to Export Functional Coverage from SystemC to SystemVerilog

Appendix

1. Old coverpoint syntax

```
template <typename T>
class coverpoint: public cvp_base
```

As shown above, coverpoints are templated with the type of the data to be sampled.

To create a coverpoint you need to specify:
A pointer to the covergroup it belongs to
A name (optional)
0 or more bins, templated by the same type (i.e. coverpoint<int> will only accept bin<int> instances

Example:

```
// Inside a covergroup declaration
coverpoint<int> data_ready_cvp = coverpoint<int> (
    this,
    illegal_bin<int>("illegal_zero", 0),
    bin<int>("positive", interval(1,INT_MAX)),
    bin<int>("negative", interval(-1,INT_MIN))
);
```

If you don't specify a name, the coverpoint will take the instance name (e.g. above, it will be named "data_ready_cvp").

Using coverpoint in a covergroup

The covergroup is responsible with dispatching sampling values to the coverpoints/crosses. Two things need to be done for this to happen:

1. Get data inside the covergroup
2. Dispatch it to coverpoints/crosses

For the first task, we declare fields inside the covergroup to hold those values:

```
int SAMPLE_POINT(value,values_cvp);
int SAMPLE_POINT(flags, flags_cvp);
```

The *SAMPLE_POINT* macro does two things:

1. Declares a member of given type (first arg) and tells the coverpoint to look there when sampled (second arg)
2. Tokenizes arguments such that the coverpoints name and sample variable show up in the report

Now we just assign them when sampling:

```
void sample(int data, int flags) {
    this->value = data;
    this->flags = flags;
}
```

All that's left to do is to trigger the sampling by calling *sample()* at the end of our function and the library will take care of calling sample for each item:

```
covergroup::sample()
```

Complete function:

```
void sample(int data, int flags) {
    this->value = data;
    this->flags = flags;

    covergroup::sample();
}
```