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

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

A covergroup construct encapsulates a set of coverpoints and crosses.

To create a covergroup, you have to define a class which extends fc4sc::covergroup

Example:

```
// In a header
#include "fc4sc.hpp"
...
class my_first_cvg : public covergroup {
// coverpoints and crosses
};
```

After creating the covergroup, you will have to declare the coverpoints and/or crosses as members:

Example:

```
// In a header
#include "fc4sc.hpp"
...
class my_first_cvg : public covergroup {

};
```

One thing that needs to be done to get results is to register your covergroup in the library (so the information gets in the report). To do this, an internal function will be called when an instance gets created. The *CG_CONS(custom_type)* macro is supplied:

```
CG_CONS(my_first_cvg) {
      // set options or type options
}
```

This gets expanded to:

```
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 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
}
```

Coverpoints

A coverpoint is a set of bins(i.e. set of values) and an expression which is to be evaluated.

```
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 - 1)),
    bin<int>("negative", interval(-1,INT_MIN + 1))
);
```

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

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 the same behavior as coverpoints, just that you will define them using coverpoint instances rather than bins.

To create a cross you need to specify:

- A pointer to the covergroup it belongs to
- A name (optional)
- 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 associates a set of values with a counter and a name. Bins hold values or intervals of interest. Each time a value contained in the bin is sampled, its associated counter increments.

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

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

To create a bin you need to specify:

- A name for the bin (optional)
- A list of values or intervals of types (for a *bin<T>* instance):
    - *T* for a single value
    - *interval<T>(T a, T b)* for an interval of values

Example:

```
// Create a bin named "power_of_2" with value 1 V [2,3] V [4,7] V [8,15]
// (where V is reunion)
auto my_bin =  bin<int>(
    "power_of_2",
    1,
    interval(2,3),
    interval(7,4),       // order doesn't matter
    interval(15,8)
);
```

You can also create *illegal_bins*. 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);
```

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 "normal" category.

Tying everything together

All that's left now is to put everything together.

```
class my_first_cvg : public covergroup {

    coverpoint values_cvp = coverpoint (this ,
      bin("low1", interval(1,6), 7), // intervals are inclusive
      bin("med1", interval(10,16), 17),
      bin("med2", interval(20,26), 27),
      bin("high", interval(30,36), 37)
   );
```

```
    coverpoint flags_cvp = coverpoint (this ,
      bin("zero", 1),
      bin("one", 2),
      bin("ten", 1024),
      illegal_bin("some_illegal_config", 3),
      ignore_bin("uninteresting", 8)
   );

    // Cross (cartesian product) the two coverpoints
    cross valid_data_cross = cross (this,
        &flags_cvp,     // the coverpoints crossed
        &values_cvp
    );

};
```

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();
}
```

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()	For covergroups: always defined by user
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:

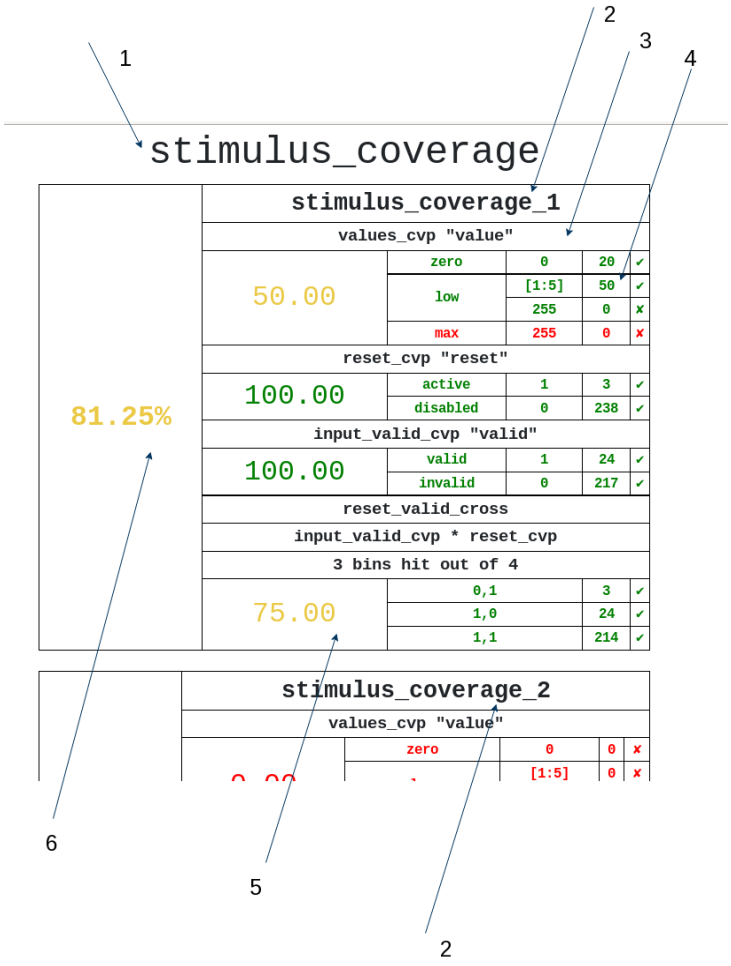
```
fc4sc::global::coverage_save (std::ofstream& stream)
```

```
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.

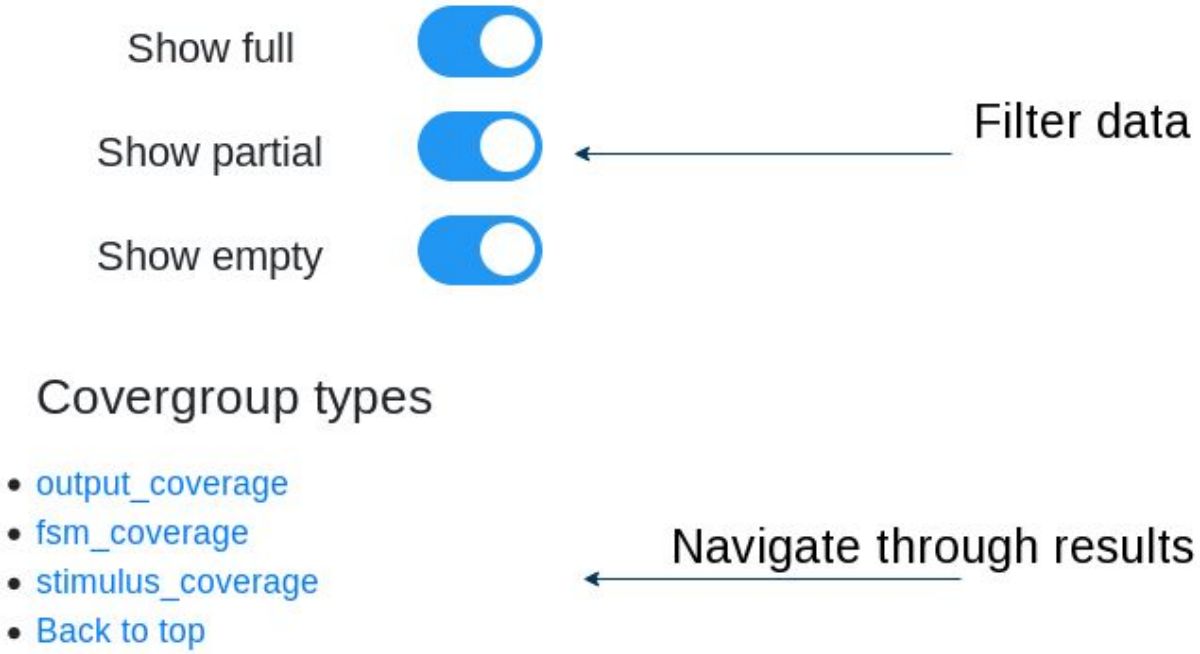
Open fc4sc/report/index.html and load the file:



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.

- Better filtering in crosses (binsof , intersect)
- Finish bin array implementation (needs more testing)
- 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.
- Possibility to use and customise default bins
- 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.
- Merge of different coverage databases

References

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