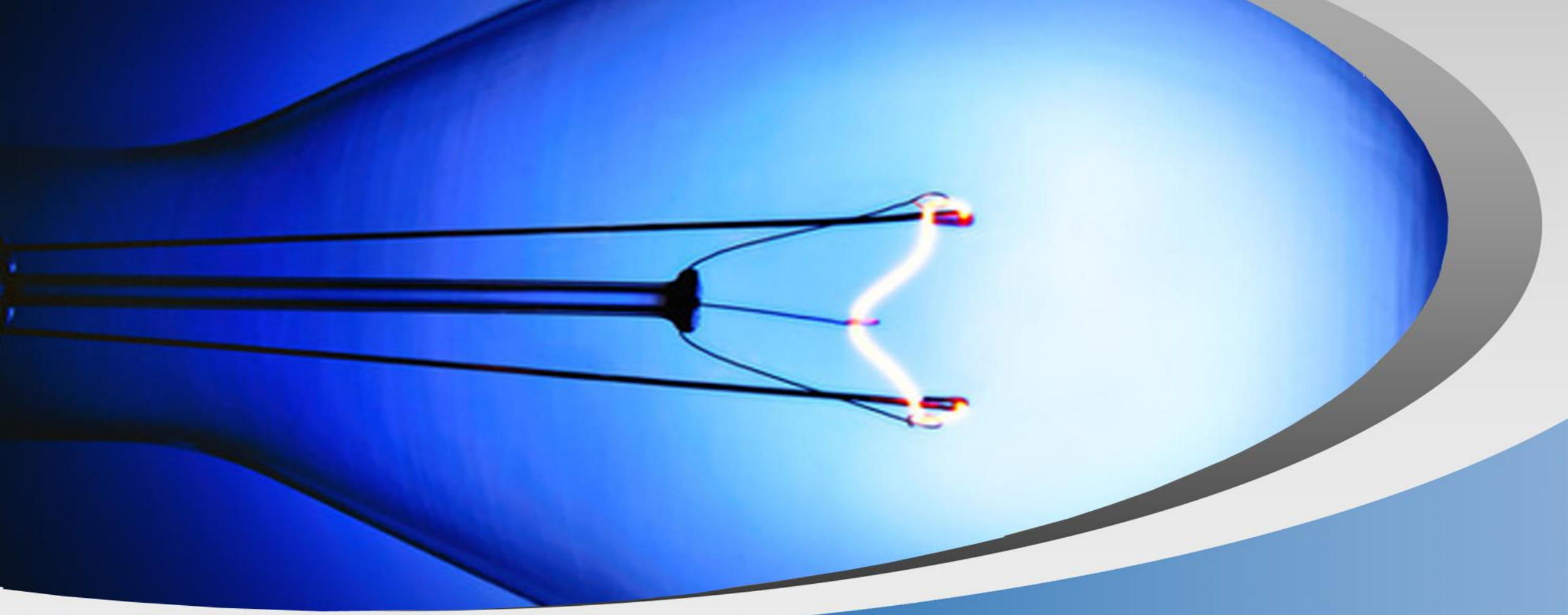

SCP : Reporting library

Mark Burton

accellera
SYSTEMS INITIATIVE

Keep It Generic

Goal of the SystemC Common Practices is that you should be able to pick up a ‘component’ on it’s own and use it – without needing to take the whole library.

SCP : reporting

- **What did we want:**
  - Generic and independent of other library components (of course)
    - Wanted this, but ... Our current implementation has some dependencies.
    - At least try to ensure that there is no dependencies on other SCP components.
  - Work with SystemC (using sc_report_ under the hood)
  - A convenient user API (FATAL, INFO….)
  - Nice syntax like FMT (“the answer is {}.”, 42)
  - VERY LOW simulation cost (a single ‘if’ to determin whether to report or not)
  - Driven by ‘CCI’ configuration so its easy to switch on/off

- **Existing implementation here:**
  - https://github.com/accellera-official/systemc-common-practices/tree/main/report

Some examples….

- SCP_TRACE() << "My trace message”
- SCP_INFO(”thing”)(“A is {} b is {} together they are {}”,1,1,42);
- SCP_DEBUG(())(”Devils inside”)
- SCP_WARN(())(“George, don’t do that”);
- SCP_FATAL(SCMOD)<<“No Going Back”;
  - . . .

Architecture stack

Existing features : Basic logging

- **SCP_TRACE() << "My trace message”**
  - Print a trace out.
- **SCP_TRACE()(“The answer is {}”, 42);**
  - Print using {FMT}
- **SCP_TRACE(“my feature”) << “Another trace”**
  - Print a trace on a feature. (The feature can then be switched off/on)
- **SCP_TRACE(SCMOD) << “More trace”**
  - Convenience to use the current module name as the feature name

Existing features : Init/Config

Uses spdlog ‘under the hood’ (an external dependencies ☹)

MANY options to configure the output

(uses sc_report as a ‘backend’)

If this isn’t called, a ‘default’ setup is provided.

```
scp::init_logging(
 scp::LogConfig()
  .logLevel(scp::log::DEBUG)
  .msgTypeFieldWidth(20)
  .fileInfoFrom(5)
  .logAsync(false)
  .printSimTime(false)
  .logFileName(logfile));
```

```
SystemC 2.3.4-Accellera --- Feb 26 2024 10:17:20
Copyright (c) 1996-2022 by all Contributors,
ALL RIGHTS RESERVED
[I] [            0 s ]CLArgumentParser          : Parse command line for --gs_luafile option (11 arguments)
[D] [            0 s ](I1) : LuaFile_Tool Constructor
[W] [            0 s ]CLArgumentParser          : --images-dir is an internal option used for testing. Do not make any assumptions on its behavior as it may change or even disappear in the future.
[I] [            0 s ]CLArgumentParser          : Option --gs_luafile with value /Users/mburton/work/tmp/qqvp/configs/fw/8540/bsp/qnx/conf.lua
[I] [            0 s ]CLArgumentParser          : Lua file command line parser: parse option --gs_luafile /Users/mburton/work/tmp/qqvp/configs/fw/8540/bsp/qnx/conf.lua
[I] [            0 s ](I1) : Read lua file '/Users/mburton/work/tmp/qqvp/configs/fw/8540/bsp/qnx/conf.lua'
[I] [            0 s ]CLArgumentParser          : Setting param platform.with_gpu to value false
[I] [            0 s ]CLArgumentParser          : Setting param platform.timeprinter.log_level1 to value 4
[I] [            0 s ]CLArgumentParser          : Setting param platform.qemu_inst.sync_policy to value "multithread-quantum"
[W] [            0 s ]pla....hexagon_cluster_0.l2vic: QOM Device creation l2vic
[W] [            0 s ]pla...hexagon_cluster_0.qtimer: QOM Device creation qct-qtimer
[W] [            0 s ]pla...in.hexagon_cluster_0.csr: Reset
[W] [            0 s ]pla...uster_0.hexagon_thread_0: QOM Device creation v67-hexagon-cpu
```

Existing features : Feature loggers

- **Avoid “lookup hash”**
- **SCP_LOGGER((my_logger));**
  - Define the variable my_logger as a logger that can be used in an SCP_TRACE.
  - The default logger is (), and it’s name will be the current module name (AND the module class name!)
  - Loggers can be named to other strings – e.g. SCP_LOGGER((),”my_feature”)
- **SCP_TRACE((my_logger)) << “More trace”;**
  - my_logger is a variable in the current context (it is an integer which carries the level of logging above which the logger will output).
  - There is also an ‘array’ mechanism to build an array of loggers.
  - This whole mechanism is somewhat ‘awkward’ but we have it because we can’t associate loggers with modules within ‘standard’ SystemC.

Pic of a module with a logger in it

Without the ‘logger’, the macro expands and calls a std::hash/map to find if we’re logging. (quite expensive)

The logger is just an int…

Initialized on first use (e.g. with CCI log level)

Once initialized, all SCP_ calls will use the int.

```
#include <systemc>
#include <scp/report.h>


class myMod : public sc_core::sc_module
{
private:
SCP_LOGGER();
myMod(const sc_core::sc_module_name& name)
 SCP_TRACE(())(”constructor");
}
```

(“Macro magic all collapses to ”if (this->logger) sc_report…”)

Existing features : CCI configuration

- **Each feature (from a logger or not) can be enabled/disabled using CCI**

  My.module.feature.log_level=5

- **The value sets the level above which logging will be enabled.**

- **A bunch of ‘matching’ rules makes enabling/disabling easier:**
  - E.g. top.log_level sets the log level for everything below top.
  - *.b.log_level sets the log level for anything with ‘b’ under it.

Problem : everything else wants to use it!

- Every single other components needs to use some sort of reporting
- It’s horrible to have to go back to sc_report_...

- So – the reporting library needs to go upstream !!!!

accellera SYSTEMS INITIATIVE

So – what do we need (revisited)

- Generic and independent of other library components (of course)
- ~~Work with SystemC (using sc_report_ under the hood)~~
  - Maybe better to build sc_report over the top of a better interface?
- A convenient user API (FATAL, INFO….)
- Nice ‘FMT’ syntax (“the answer is {}.”, 42)
- VERY LOW simulation cost (a single ‘if’ to determin whether to report or not)
- ~~Driven by ‘CCI’ configuration so its easy to switch on/off~~
  - ‘CCI’ isn’t in the kernel or an IEEE standard (yet) – it can be one way to set what is enabled/disabled, but we could have a ‘clean’ interface.

- One (non) discussion, by the time this goes into the SystemC standard, we would probably be moving to C++20, which already has FMT ’built in’ (so no issues about external libraries).
- Spdlog is currently a ‘back end’ implementation, which does not need to be part of the standard.

Architecture stack

This….

Or maybe . . .

To Be Descided . . . .

What else?

- PLEASE JOIN IN!

- ‘loggers’ that send to multiple feature logs?
- Can we remove some complexity?
- Is spdlog the right ‘back end’ – should we build sc_report ontop, or should we use sc_report as a back end?

- ”step one” – separate the CCI mechanism from the loggers, so the loggers can be added to e.g. ‘sc_module’ (anywhere else? sc_object?)