Functional Coverage in SystemC

Mark Glasser
NVIDIA Corporation

March 21-22, 2018
Silicon Valley

Agenda

Functional Coverage

SystemVerilog Coverage Model

SystemC Coverage Flow

SystemC Coverage API

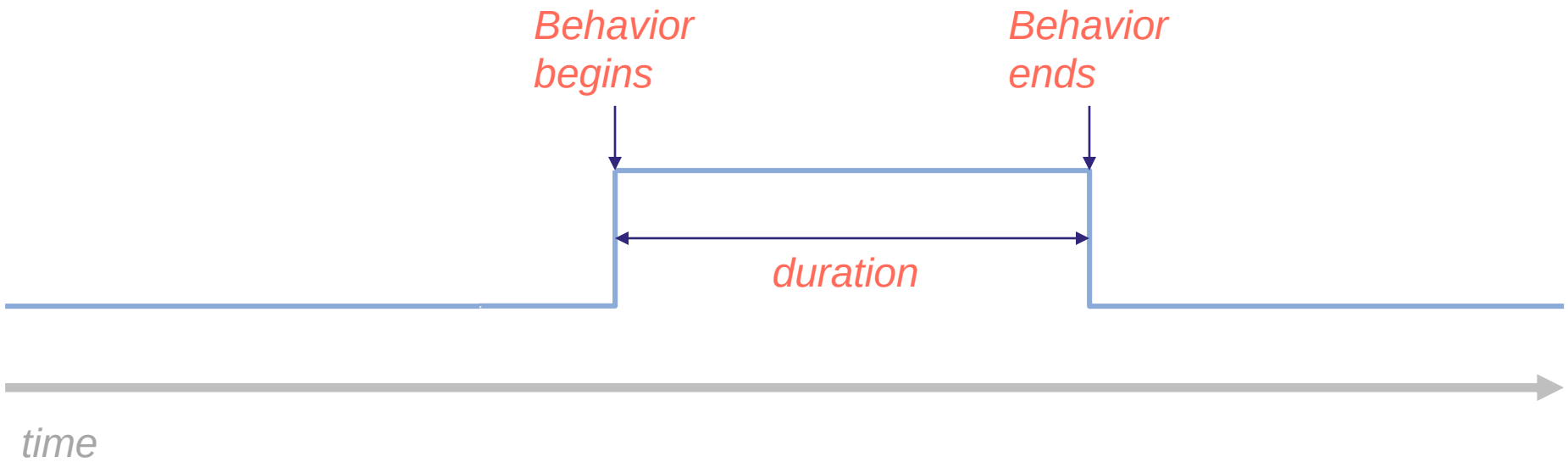
Use Models

Functional Coverage

- If a behavior occurs in a test it has been covered
- Counting occurrences of specific behaviors
- Defining a coverage model
- Covering the model via simulation
- Coverage closure is achieved when all (interesting) behaviors in coverage model have been covered.

Behaviors

- Any change or changes in state
- Bounded in time
- Examples
  - occurrence of an interrupt
  - transfer of a video frame



Coverage Model

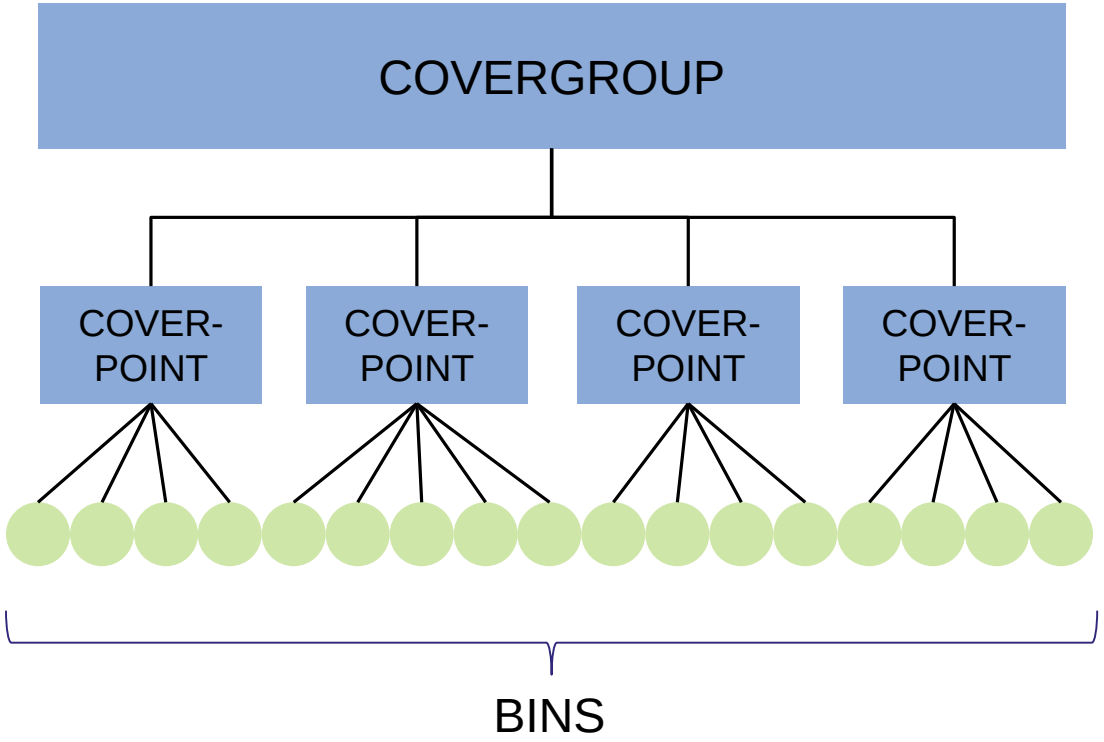
- List of behaviors to be covered
- Represented abstractly in English
- Represented concretely via modeling language constructs
- Forms a contract between Design/Architecture teams and Verification team

Why Coverage in SystemC?

- SystemC does not natively have any way of collecting functional coverage data
- There are reasons to adds this capability
  - SystemC models can be validated using randomized stimuli
  - Tests can be vetted early in the design cycle
  - Coverage claim about RTL can be made using SystemC reference models (more on this later)

SystemVerilog Coverage

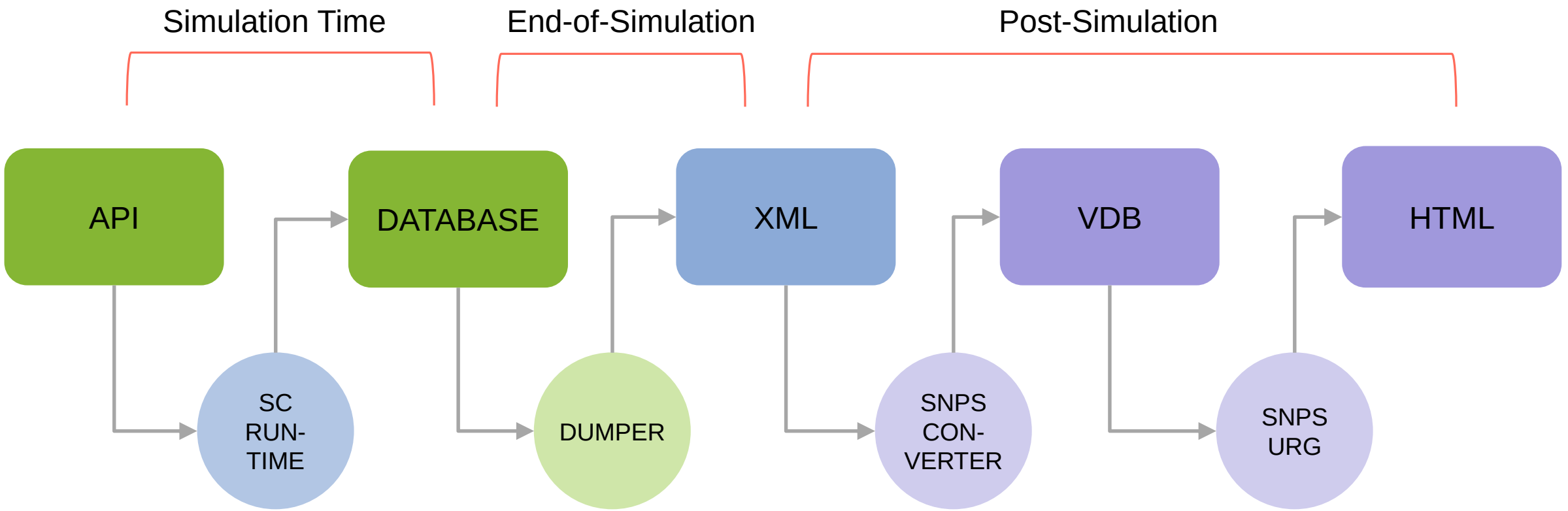
- Covergroups
  - Contain collections of coverpoints
  - Supply sampling event
- Coverpoints
  - Contain collections of bins
  - Have elaborate syntax for defining bins
- Bin
  - Counter of specific behaviors
- SV has elaborate syntax for defining covergroups, coverpoints, and bins



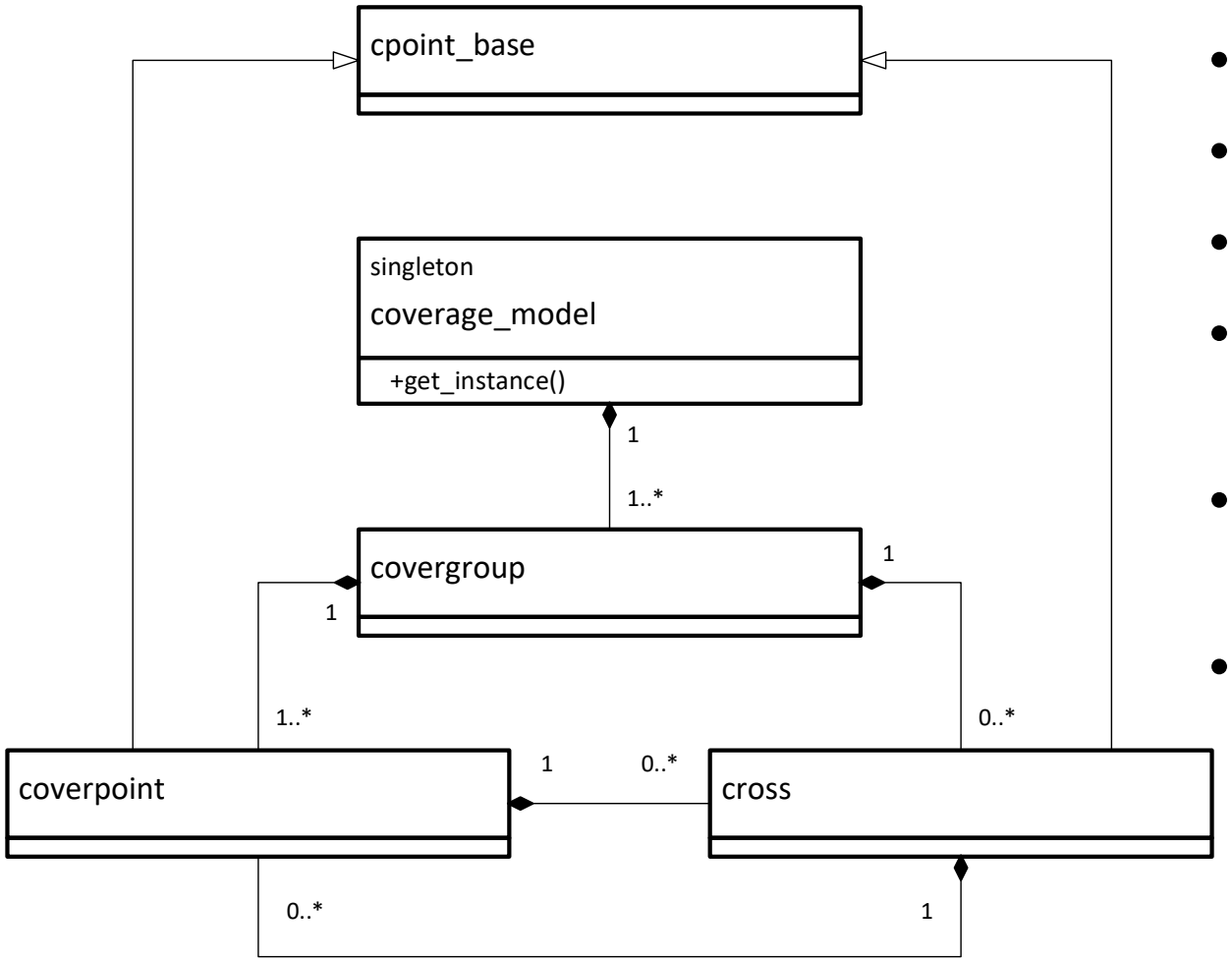
SystemC Coverage Challenge

- How to automate the recognition and counting of the occurrences of behaviors described in the coverage model?
- Requires three elements
  - database for storing covering information
  - API for instrumenting models
  - Reporting mechanism
- Emulate SystemVerilog Coverage Model to the extent possible
  - Enable integration with SV coverage flow
  - Make sense of SV and SC coverage

SC Coverage Flow



Database



- Database resides in memory
- Updated by API
- Coverage model is a singleton
- Covergroup contains coverpoints and crosses
- Coverpoints and crosses are derived from the same base class
- Crosses and coverpoints are crossreferenced

Coverage API

```
#define SC_COVERGROUP(cg)
#define SC_ADD_COVERGROUP(cg, inst_name)
#define SC_ADD_COVERPOINT(cg, name, bins, th)
#define SC_ADD_CROSS(cg, name, cpname)
#define SC_COVERPOINT_BIN_NAMES(cg, name, names, count)
#define SC_COVERPOINT_IGNORE_BIN(cg, name, bin)
#define SC_COVERPOINT_ERROR_BIN(cg, name, bin)
#define SC_COVERPOINT_DEFAULT_BIN(cg, name, bin)
```

Go in SC_MODULE constructor

```
#define SC_EXPR_COVERPOINT(cg, name, expr, bin)
#define SC_RANGE_COVERPOINT(cg, name, expr, lo, hi, bin)
#define SC_VAR_COVERPOINT(cg, name, var)
#define SC_SAMPLE_COVERPOINT(name, sample_event)
```

Go in SC_MODULE class definition

```
#define SC_INLINE_EXPR_COVERPOINT(cg, name, expr, bin)
#define SC_INLINE_RANGE_COVERPOINT(cg, name, expr, lo, hi, bin)
#define SC_INLINE_VAR_COVERPOINT(cg, name, var)
```

Go inline in procedural code SC_METHOD or SC_THREAD

Coverage Model Creation

- Create covergroups in SC_MODULE
- Add covergroups to coverage model
- Add coverpoints and crosses to covergroups
- Optionally, supply names for bins
- Optionally, designate bins as ERROR, IGNORE, or DEFAULT

Sampled Coverpoints

- Coverage is sampled when a specified event occurs
- Expression Coverpoint
  - specified bin in incremented when expression evaluates to true (non-zero)
- Variable Coverpoint
  - specified bin is incremented unconditionally
- Range Coverpoint
  - specified bin is incremented when expression evaluates to a value in a range

Sampled Coverpoint Implementation

```
#define SC_EXPR_COVERPOINT(cg, name, expr, bin)      \
  void coverpoint_##name()                           \
  {                                                  \
    bool cov = (expr);                               \
    if(cov)                                          \
      cg.incr_bin(sc_str(name), bin);                \
  }
```

- Macro defines a small function
- Bin is incremented when condition is met

More Coverpoint Implementation

```
#define SC_SAMPLE_COVERPOINT(name, sample_event)    \
   SC_METHOD(coverpoint_##name);                     \
   sensitive << sample_event;
```

- Indentified coverpoint function as an SC_METHOD
- Establishes static sensitivity to specified event
- Each sampled coverpoint must have a sampling event specified via SC_SAMPLE_COVERPOINT

Inline Coverpoints

- Increments bin when locus of control passes through coverpoint
- The locus of control serves as the sampling ”event”
- Expression, Variable, and Range coverpoints
  - Same semantics as for sample coverpoints

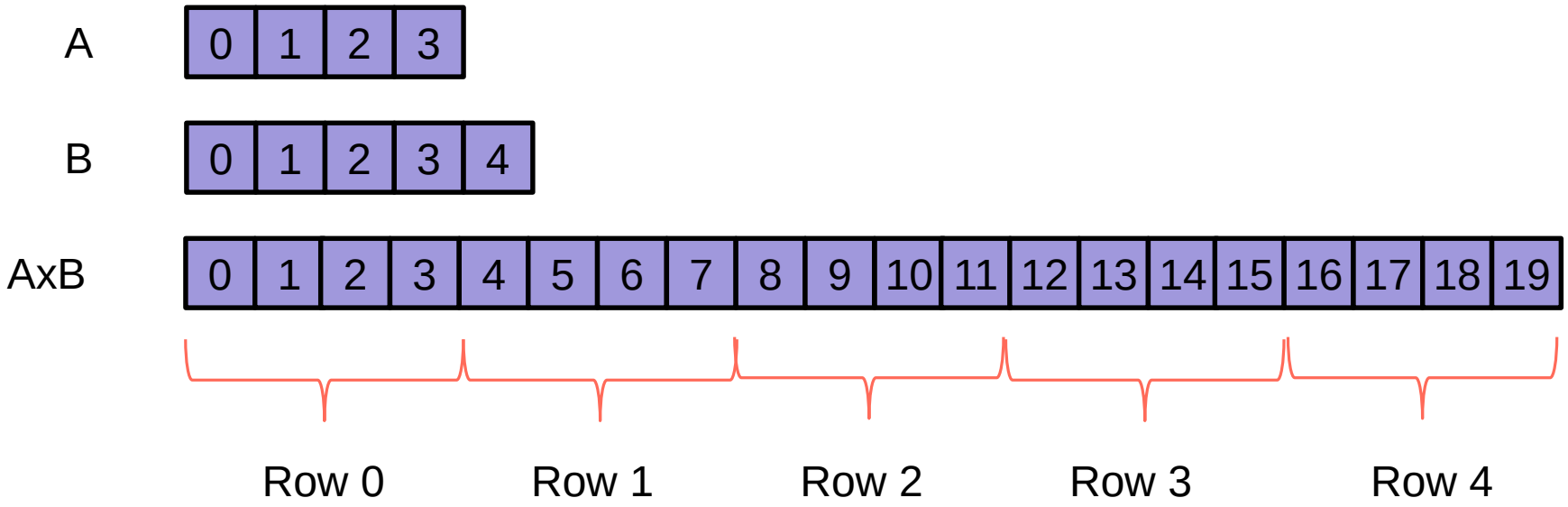
Inline Coverpoint Implementation

```
#define SC_INLINE_EXPR_COVERPOINT(cg, name, expr, bin)     \
  {                                                         \
    bool cov = (expr);                                      \
    if(cov)                                                 \
    {                                                       \
      cg.incr_bin(sc_str(name), bin);                       \
    }                                                       \
  }
```

- Expression is evaluated when locus of control passes through the coverpoint
- Bin is incremented if condition is true
- Similar semantics to sampled coverpoints

Crosses

- A cross is a list of coverpoints
- Number of bins in a cross is the Cartesian product of member coverpoints
  - N-dimensional matrix of bins
- After each coverpoint update, associated crosses are updated
  - Database tracks last bin updated for each coverpoint



Turn Coverage On/Off

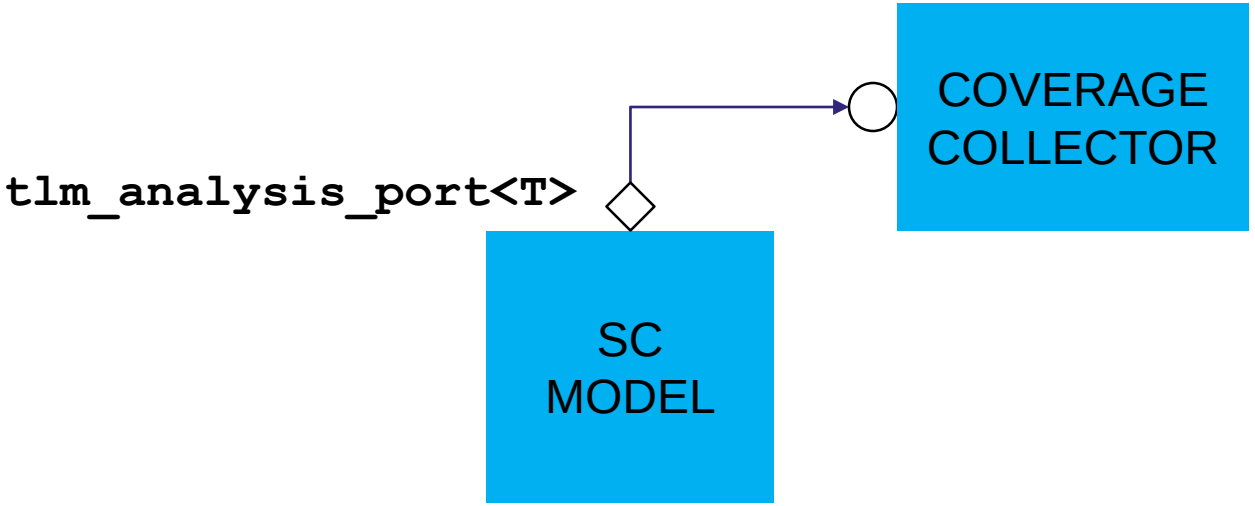
- Collecting coverage consumes overhead
- Not always desirable (or feasible) to consume excess overhead
- Functional coverage can be turned off
- Compile models with `-DNO_SC_COV` to turn coverage off
- Defines macros with vacuous definitions
- Run-time switch under development

Use Models

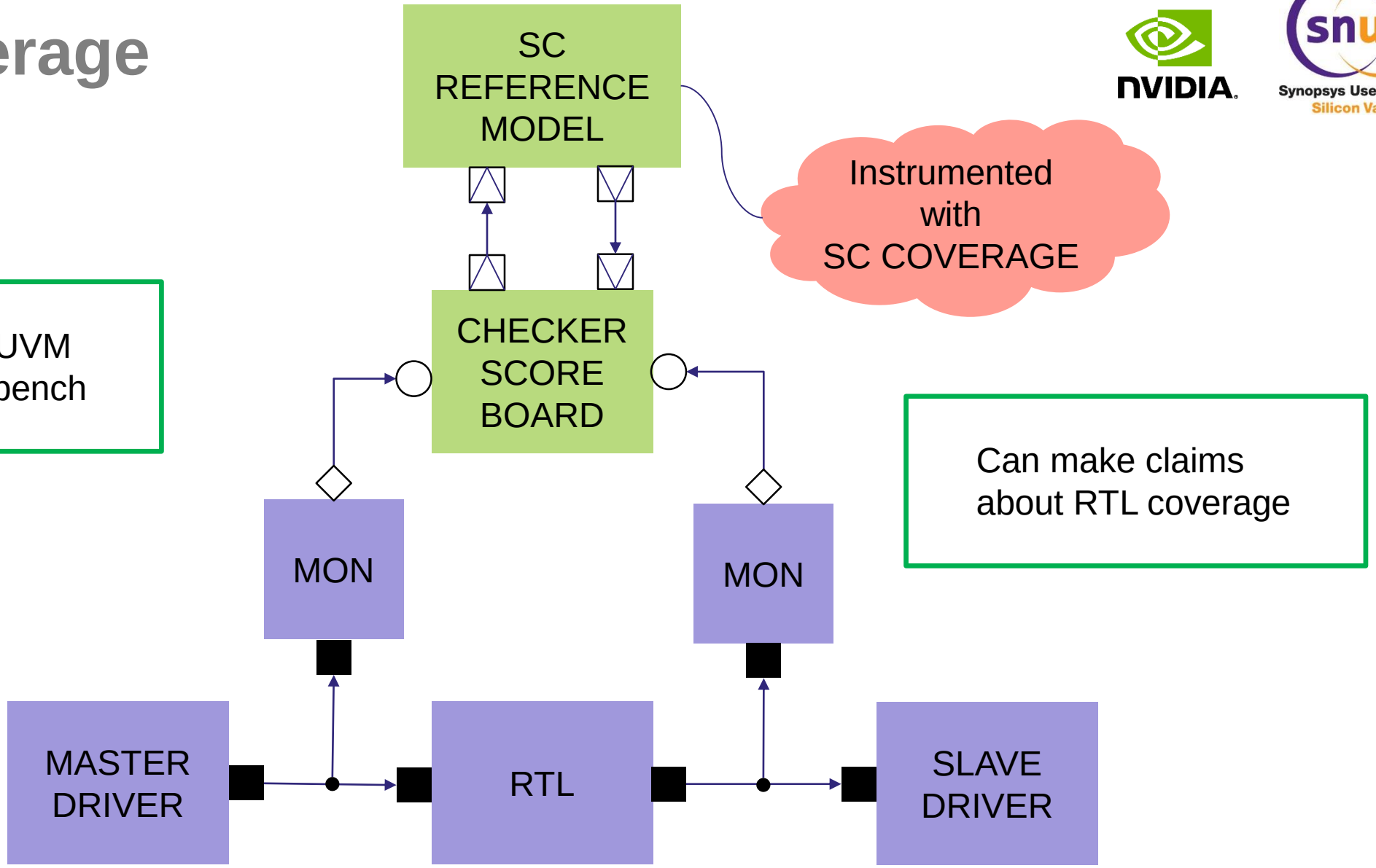
- Validation of SystemC architectural models
- Gaining confidence in tests early in the design cycles
- Functional coverage in RTL verification

Keeping Models Clean

- Don’t always want to modify model code to add functional coverage
- Use analysis port and subscriber to keep functional coverage separate
- Coverage collector is an `SC_MODULE` with an `tlm_analysis_if<T>` and an implementation of `write()`
- Use inline coverpoints
- Call to `write()` is the sampling event



RTL Coverage



Conventional UVM
Unit-level testbench

Can make claims
about RTL coverage

Conclusion

- SystemC is widely used for architectural and RTL reference models
- SystemC has no native functional coverage facility
- The facility we have developed provides functional coverage semantics in a manner parallel to SystemVerilog functional coverage
- The flow enables us to gain confidence in our SystemC models

Thank You