C++ Testbench using UVM phase and agent concepts

JinHaeng Cho
NVIDIA

September 24, 2015
SNUG Boston

Agenda

Testbench architecture

Implementation details

Conclusions

Future work

Testbench Architecture

Considerations for choosing testbench approach

- Industry standard and trend
  - Most common frameworks, methodology and testbench languages
  - Availability of tools and libraries
  - Easy to access expertise and resources such as documents, online examples
  - Long term maintenance and upgrade cost
- Integrating and exporting IP
  - Interoperability with existing environment
  - Import external VIP
  - Export model to another environment

Testbench Architecture

Considerations for choosing testbench approach

- Project Schedule
  - Familiarity of language for quick ramp-up and implementation
  - Model and testbench readiness
- Reusability and future expansion
  - Reuse prototyping model in verification phase
  - Reuse in the future revision or project
  - Easy to update for new feature additions

Testbench Architecture

Why C++?

- UVM is becoming the de-facto testbench standard but …
  - Desire to use testbench components with existing C++ environment
  - Modeling work started even before specification was settled: quick prototyping was required
- SystemC could have been an alternative choice
  - Provides many useful libraries
  - But requires non-trivial ramp-up time
  - Need some infra-structure related work
- External VIP
  - There was external VIP to integrate which utilized cycle based, 2 state, C++ modeling

Testbench Architecture

Benefits of using C++?

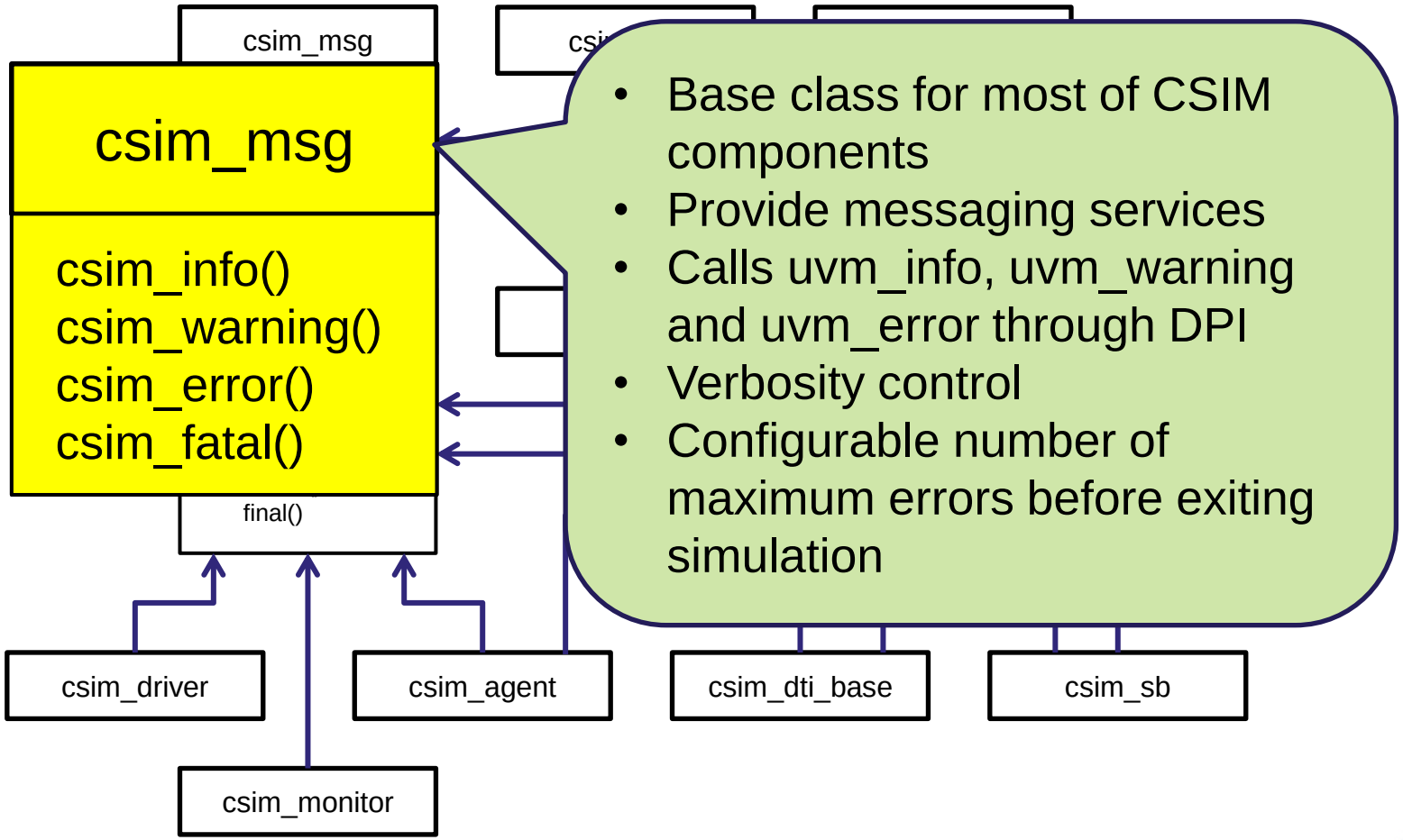
- Build time reduction
  - C++ model is built into shared-object file
  - No need to rebuild whole VCS executable
- Light simulation overhead
  - SystemVerilog and UVM provides many convenient features but they decrease simulation performance
- Early readiness for DUT verification
  - Prototyping models were reused in the design verification phase
  - C++ model was instantiated as the DUT so that once RTL was ready, simple DUT replacement created a full design verification environment

Implementation Details

CSIM classes

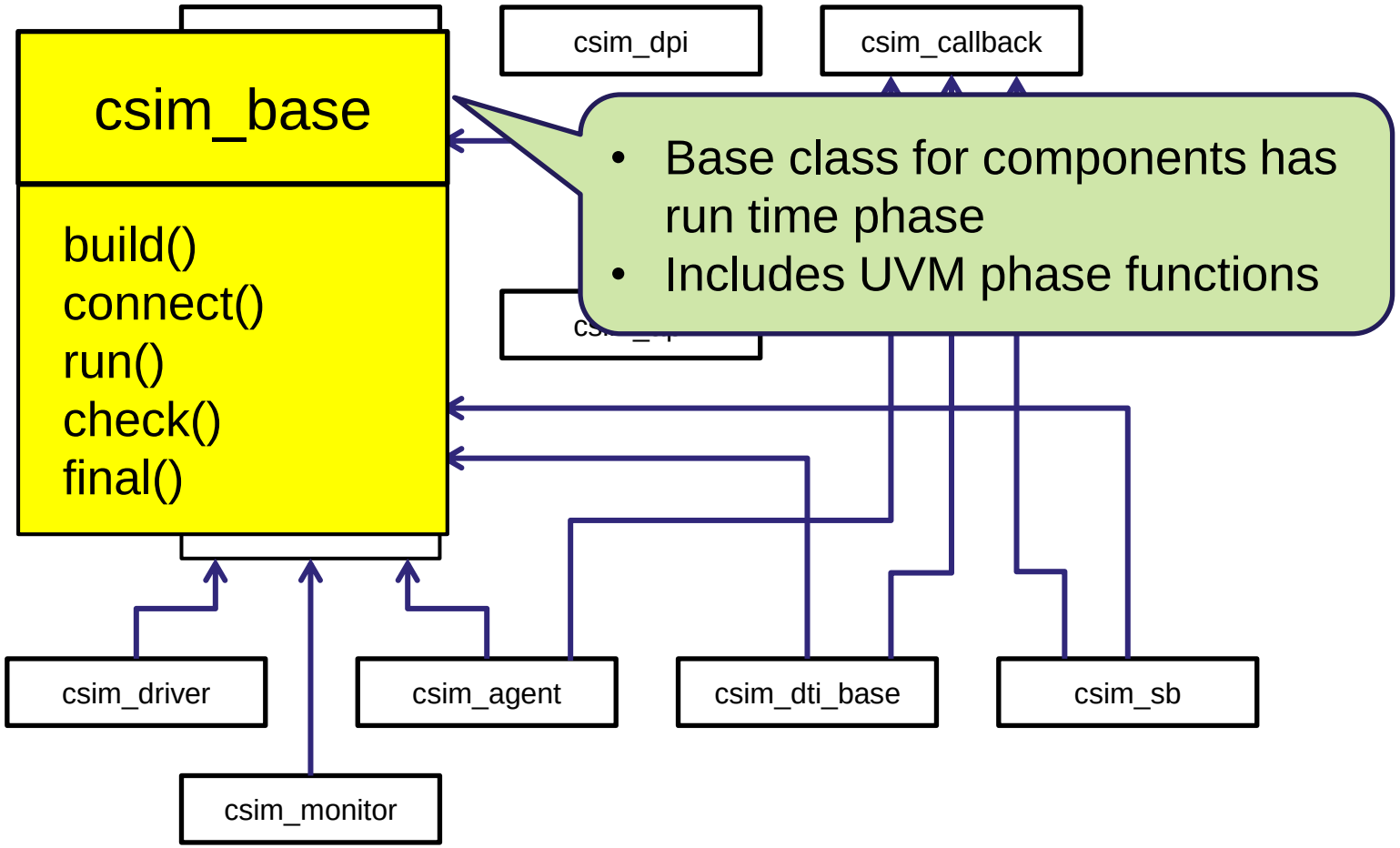
Implementation

CSIM classes – csim_msg



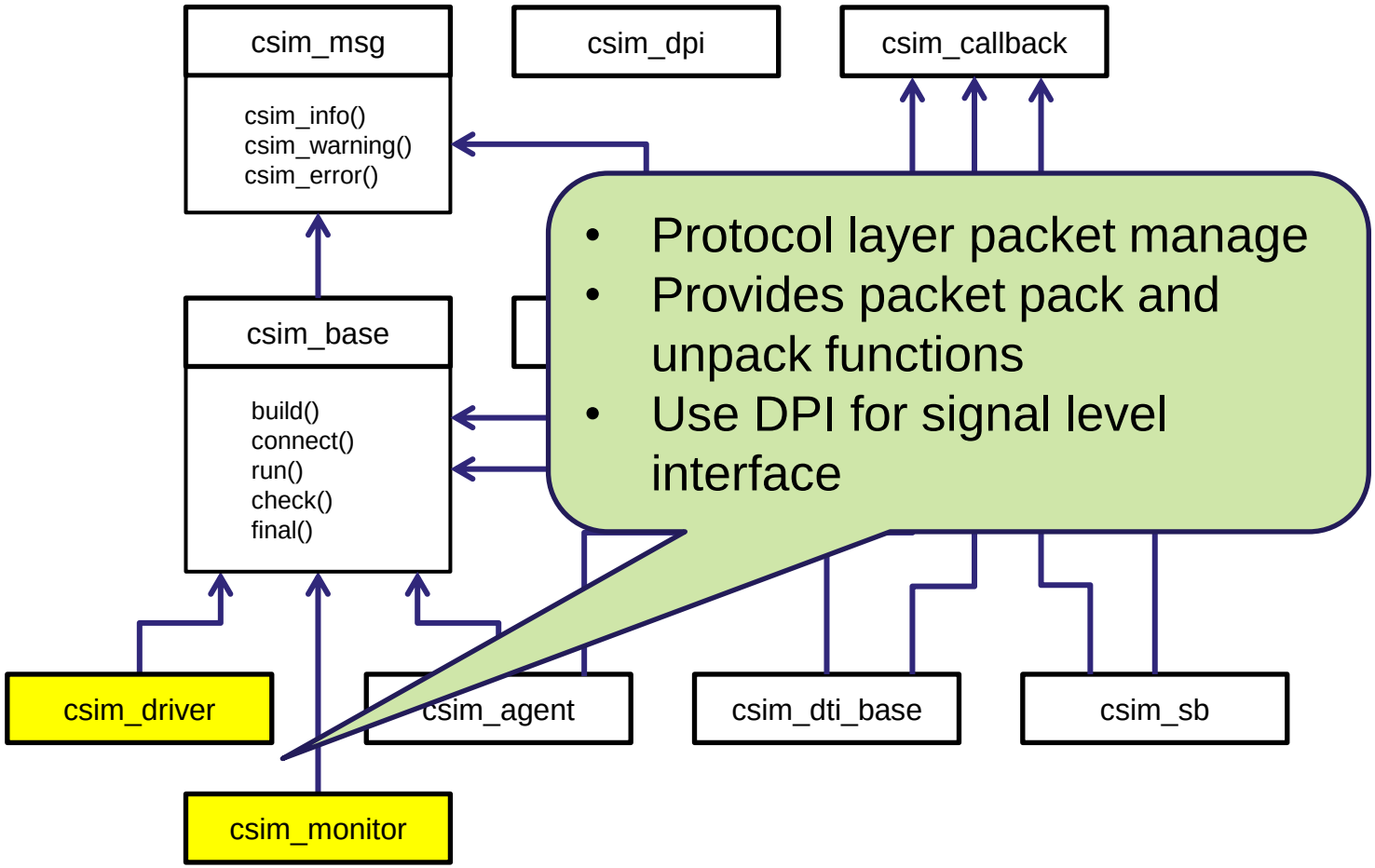
Implementation

CSIM classes – csim_base



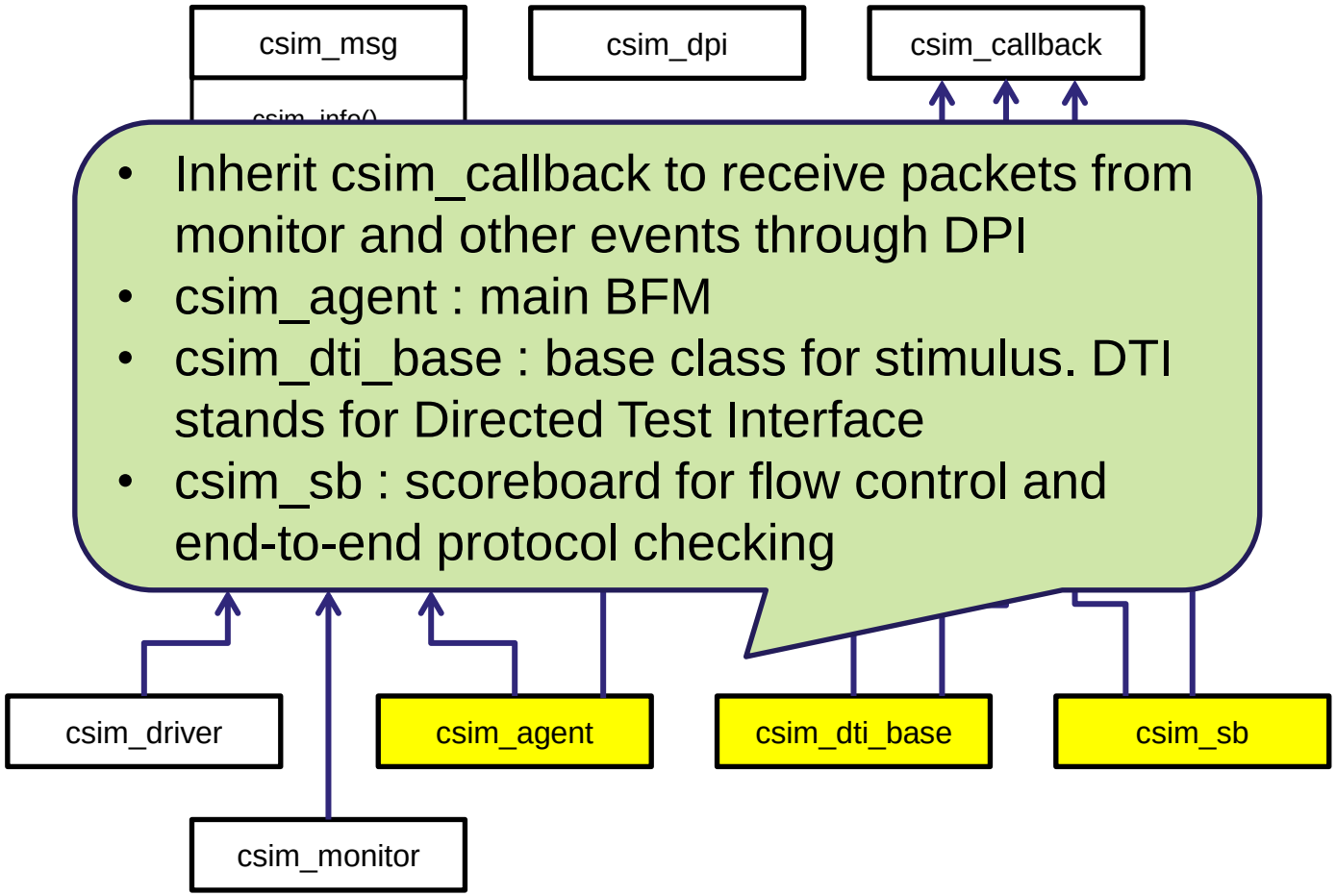
Implementation

CSIM classes – csim_driver, csim_monitor



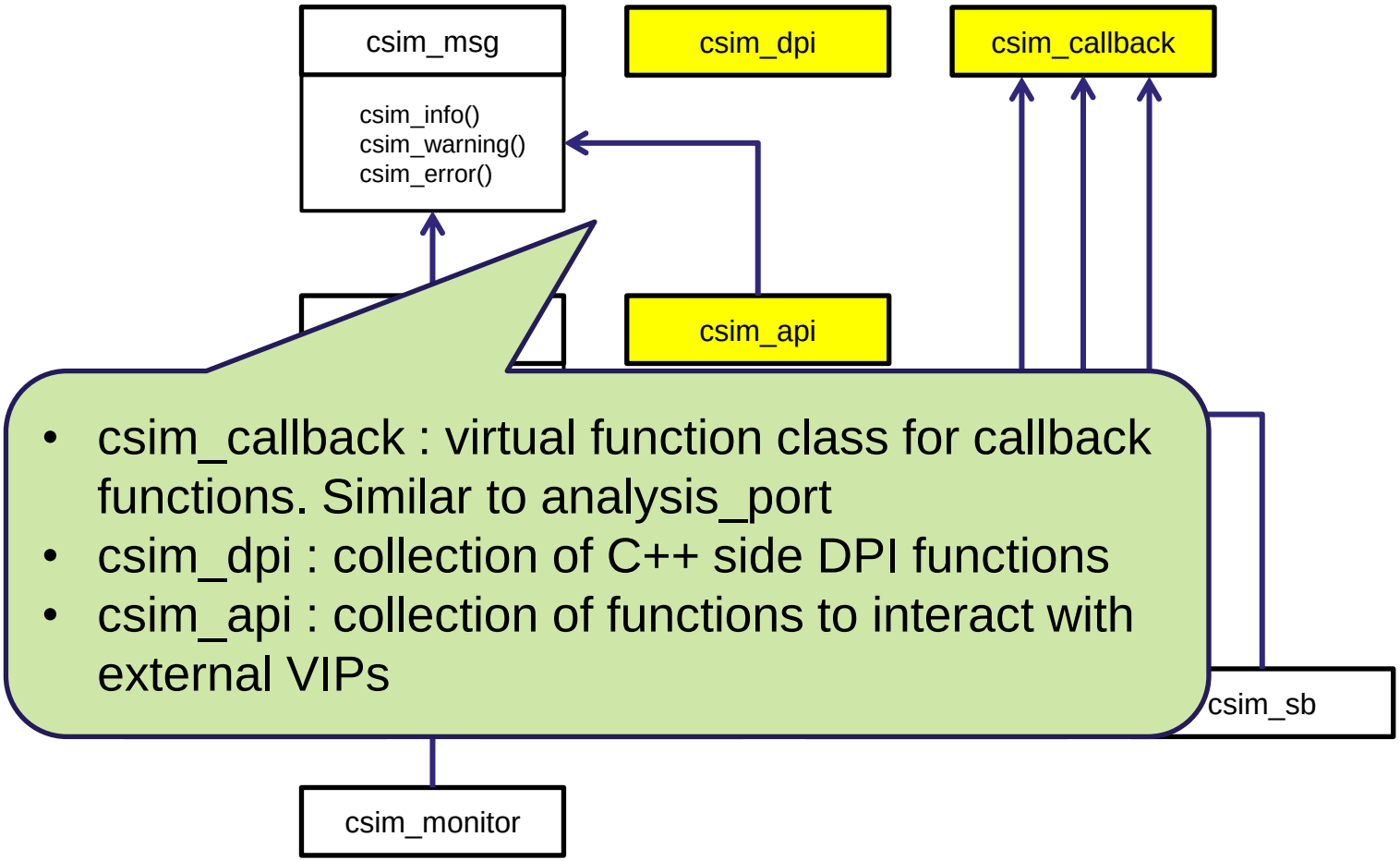
Implementation

CSIM classes – csim_agent, csim_dti_base, csim_sb



Implementation

CSIM classes – csim_callback, csim_dpi, csim_api



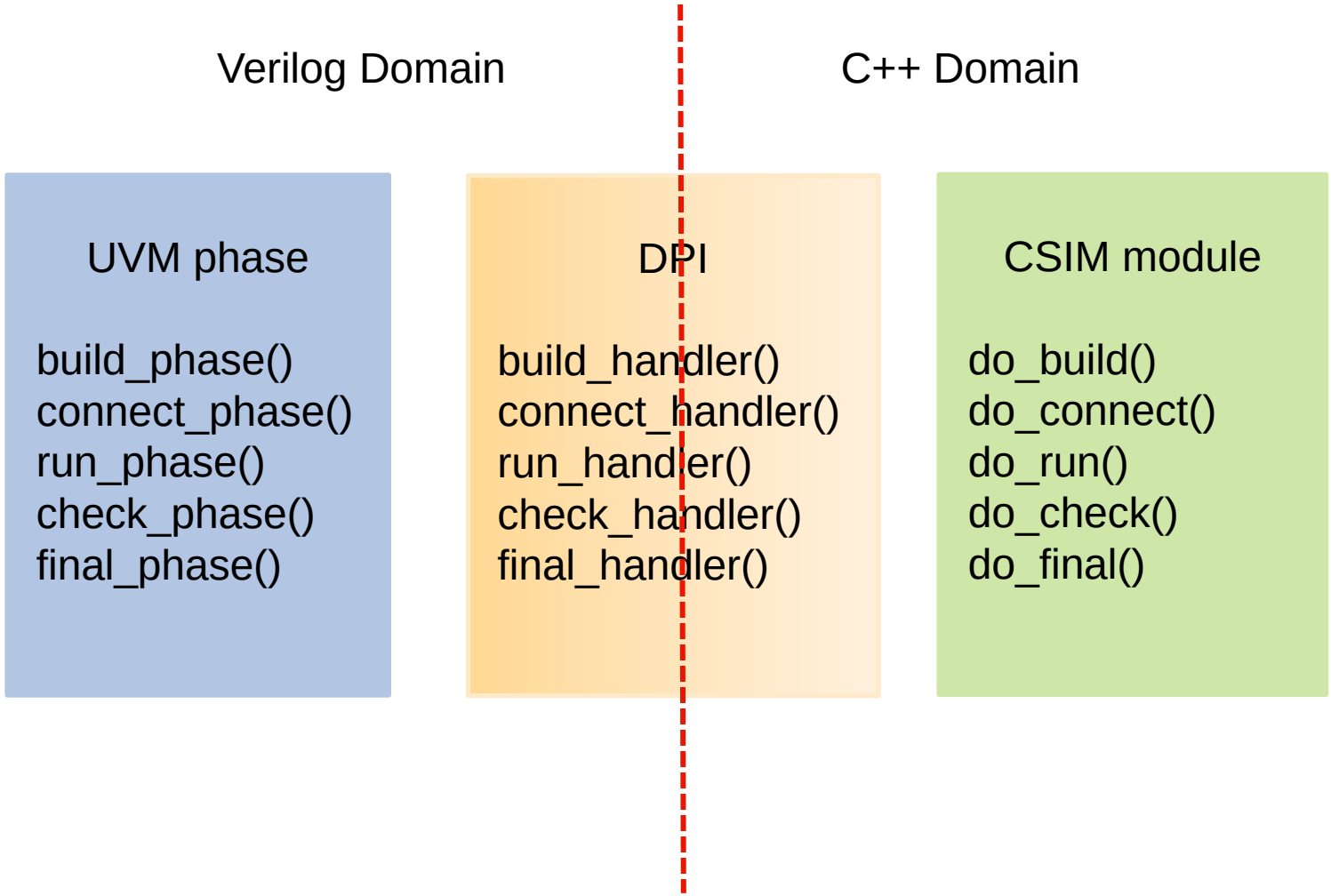
- csim_callback : virtual function class for callback functions. Similar to analysis_port
- csim_dpi : collection of C++ side DPI functions
- csim_api : collection of functions to interact with external VIPs

Implementation Details

CSIM phases

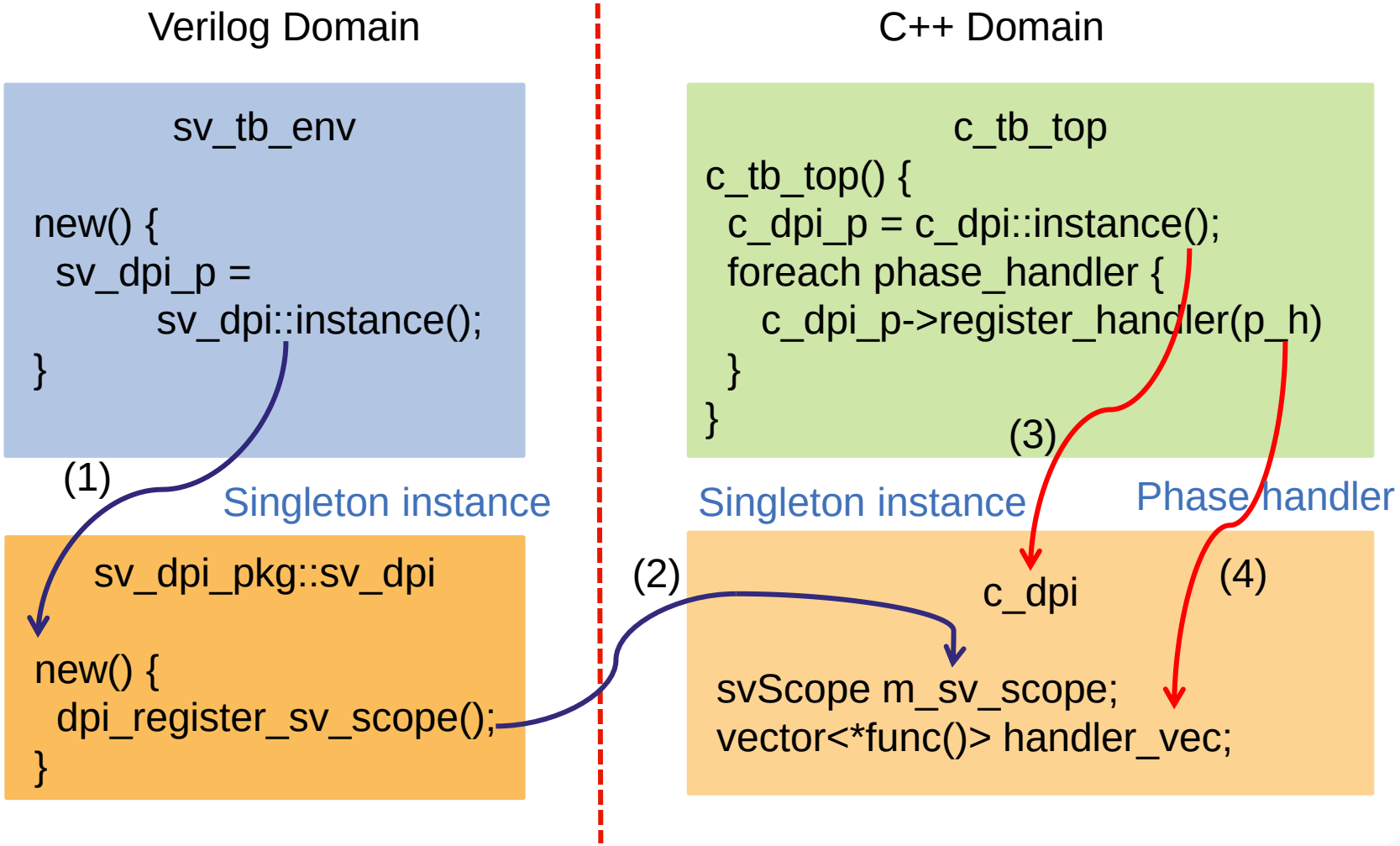
Implementation

CSIM phases



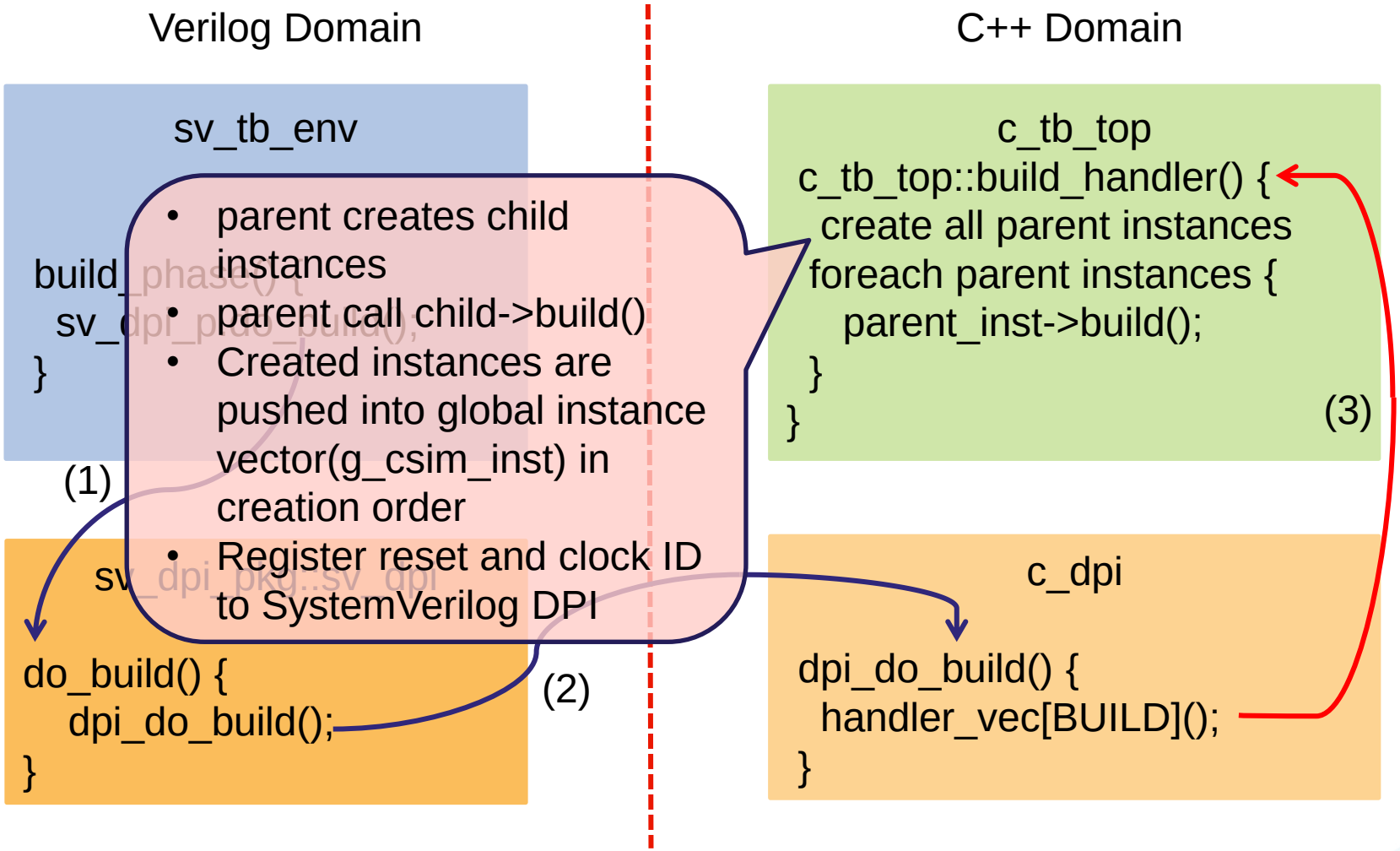
Implementation

CSIM testbench – new()



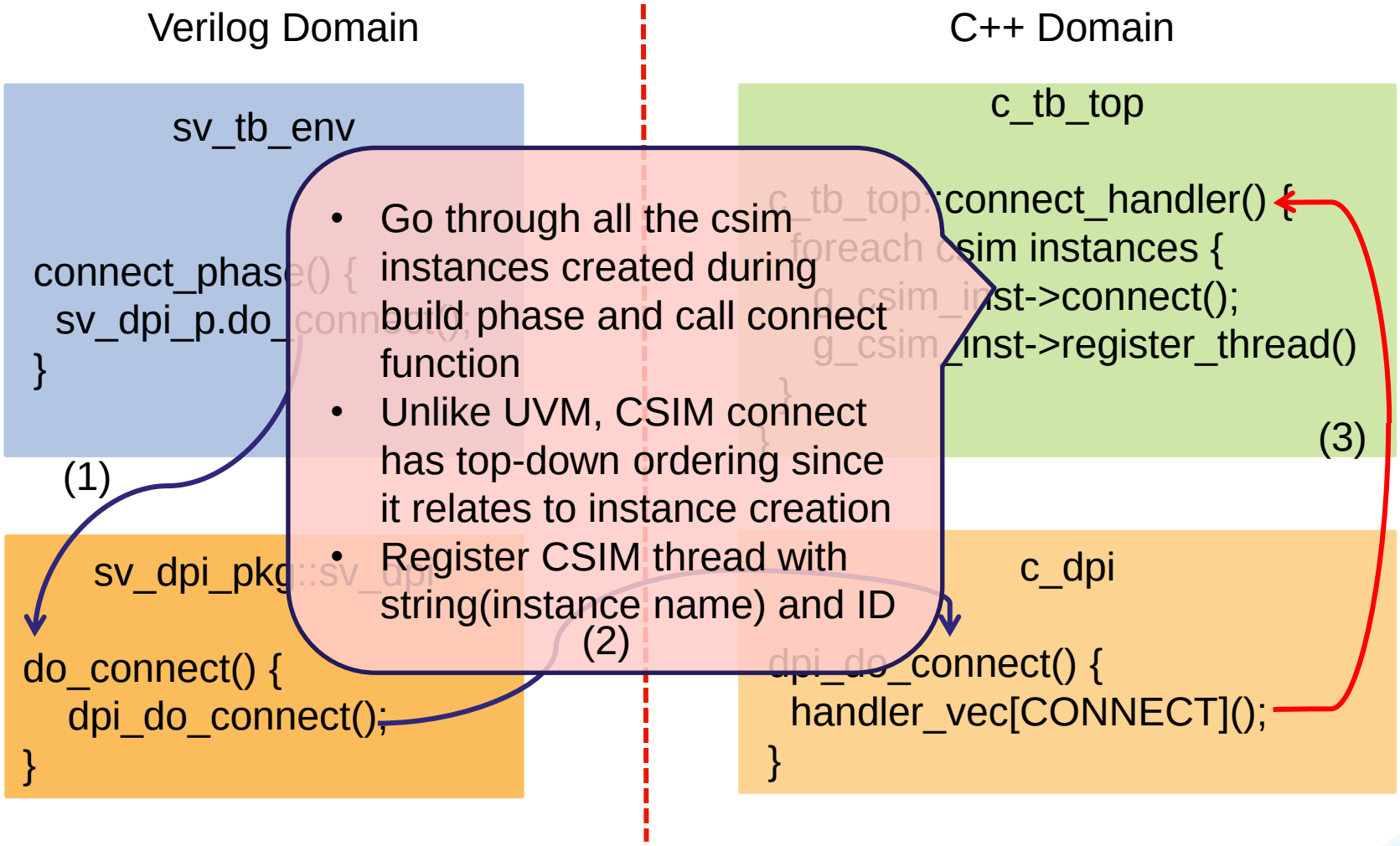
Implementation

CSIM testbench – build_phase()



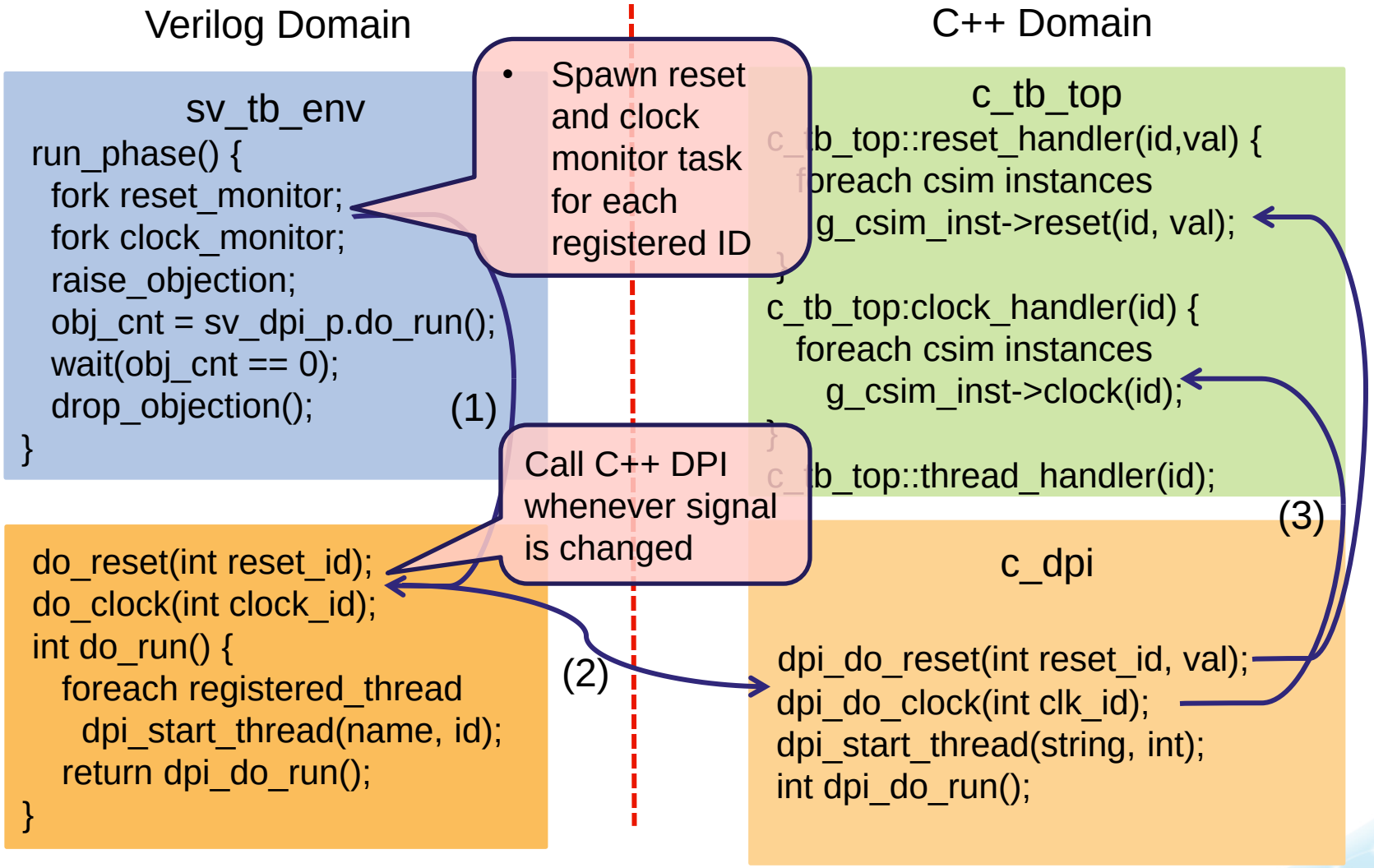
Implementation

CSIM testbench – connect_phase()



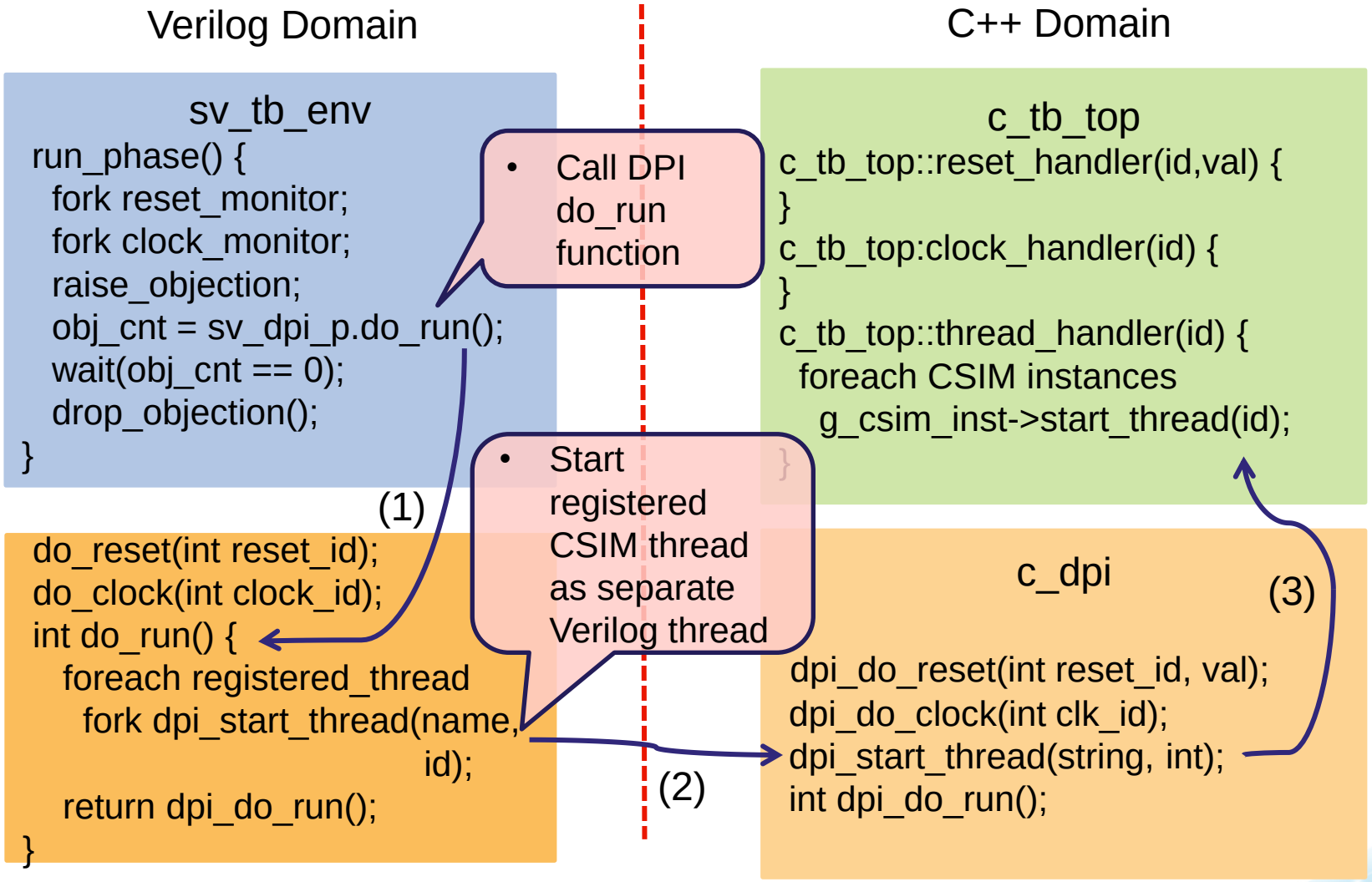
Implementation

CSIM testbench – run_phase()



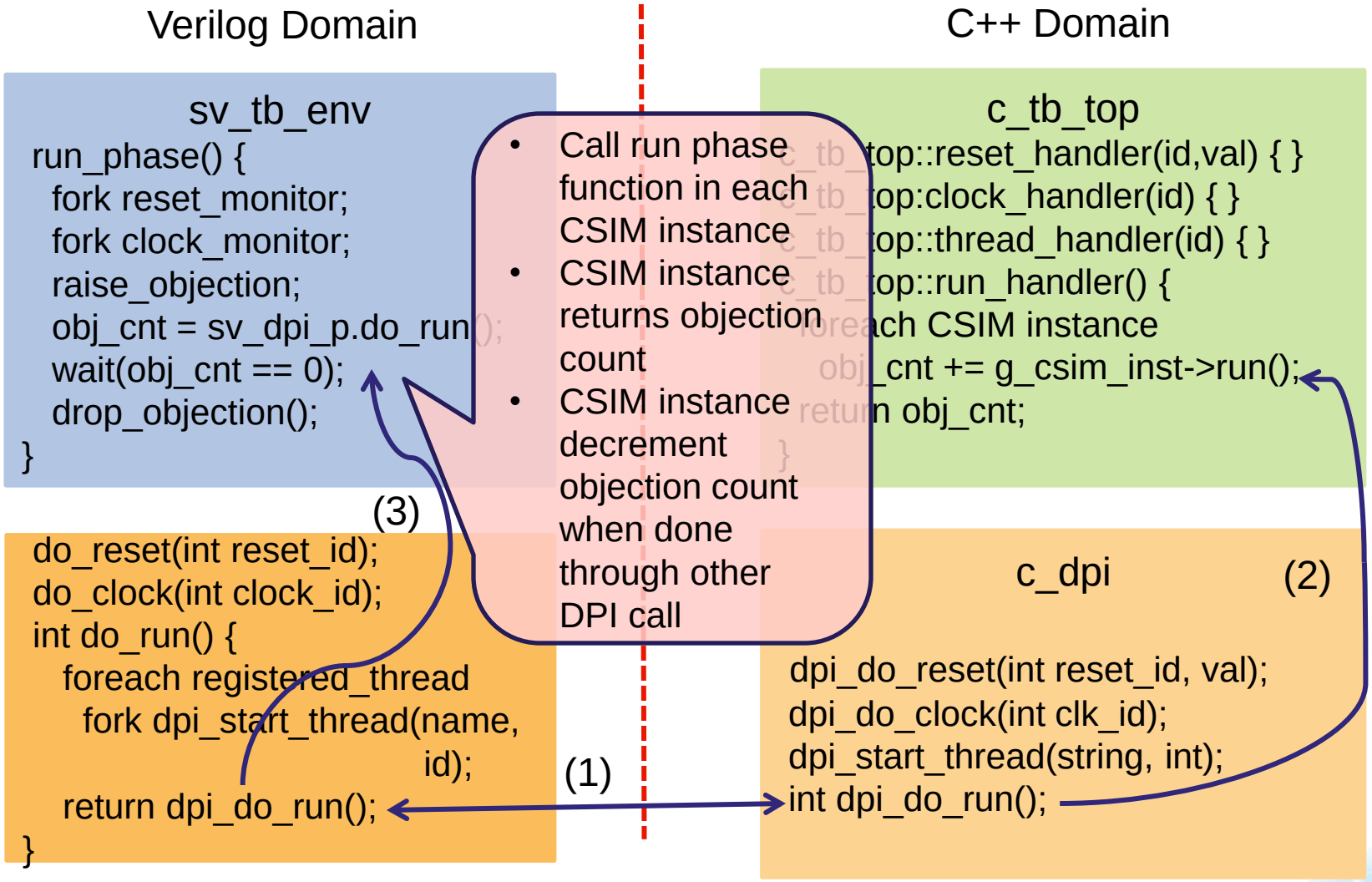
Implementation

CSIM testbench – run_phase()



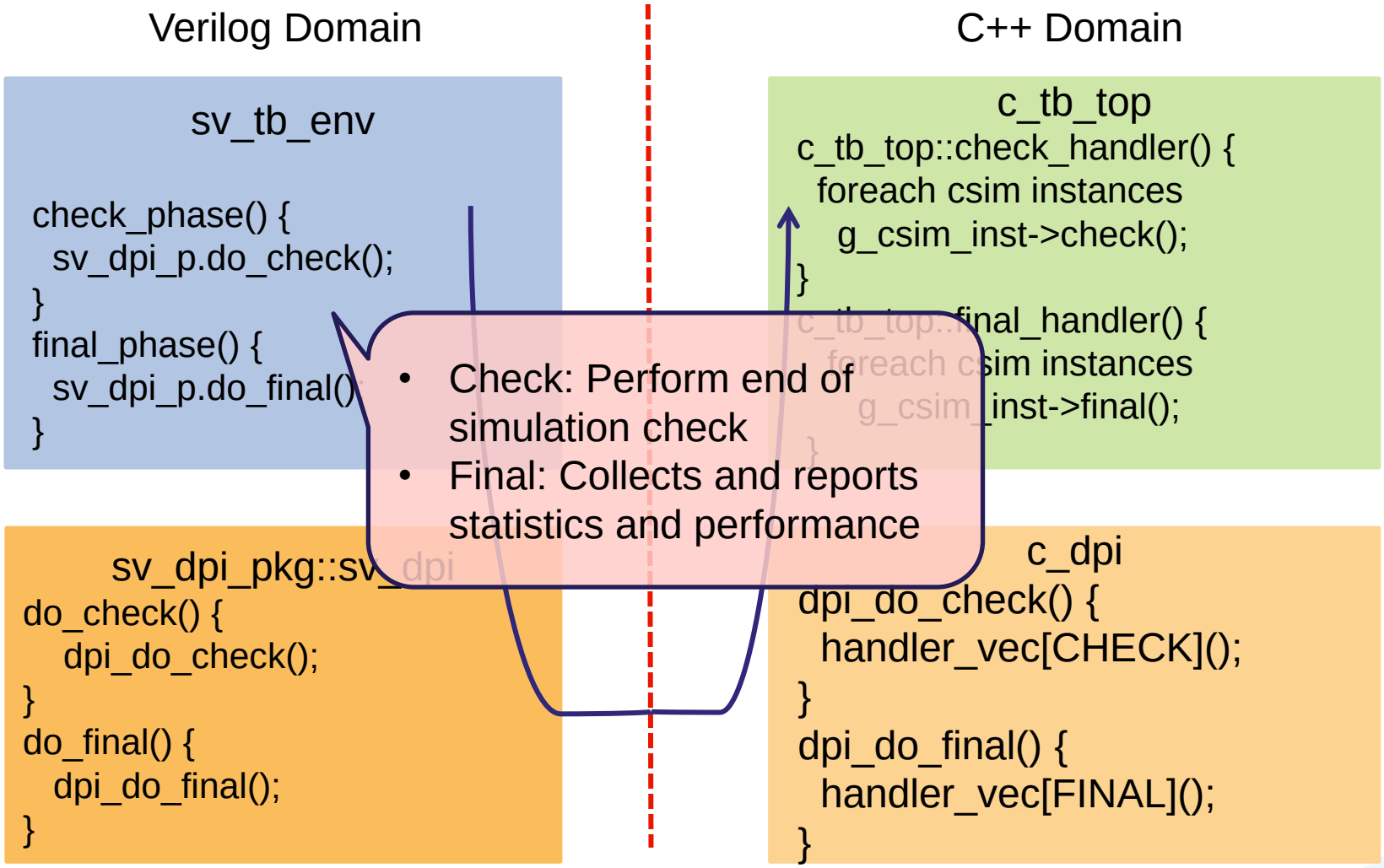
Implementation

CSIM testbench – run_phase()



Implementation

CSIM testbench – check_phase() & final_phase()



Implementation

Driving and Collecting packets

- Split into two parts
  - C++ : protocol engine that implements pack and unpack functions
  - SystemVerilog : signal level interface
- Driving packets
  - C++ driver unpacks packets and calls DPI to send bit array to SystemVerilog side
  - SystemVerilog driver drives signals through virtual interface
- Collecting packets
  - SystemVerilog monitor sends bit array to C++ monitor through DPI
  - C++ monitor packs it to construct a packet
  - Once whole packet is constructed, sends it to the consumer through CSIM callback class registered to the monitor

Implementation
Multi-Threading

- Default thread, clock(int) is similar to SystemC SC_METHOD
- SystemC SC_THREAD like operation can be added
- CSIM module registers thread function by name and ID string during connect phase
- Each CSIM thread started as separate SystemVerilog thread during run phase
- Each CSIM thread may raise objection then drop objection when it is done
- CSIM thread is generally used by stimulus

Implementation

Multi-Threading - registration

```
class c_dti_test::csim_dti_base {
  string m_inst_name, m_thread_name;
  int m_obj_cnt = 0;
  void connect() {
    register_thread(m_inst_name,
                    m_thread_name);
  }
  void start_thread(string t_name) {
    if(!t_name.compare(m_thread_name.str())) {
      m_obj_cnt++
      my_thread_func();
    }
  }
  void run() {
    return m_obj_cnt;
  }
  void my_thread_func() {
    …
    drop_objection(m_inst_name, 1);
  }
}
```

- Thread information is stored in sv_dpi's queue
- Called during run phase and called before run() call
- Returns number of local objections
- When done decrement objection count

Implementation

Stimulus generation

- Packet CSIM class, similar to uvm_sequence_item
- All test cases are basically directed scenarios
- Packet class has fields that can be randomized
- Packet field randomization constraints are controlled through command line options
- Common code among test cases are implemented in the base test class: csim_dti_base
- The base test class provides
  - Testbench initialization
  - Idle condition check for end of simulation detect

Implementation
Testbench configuration

- Since CSIM components are implemented in C++, uvm_config_db or uvm_resource_db are not used
- Publicly available sknob library is utilized
- Configuration information are passed either as VCS command options or a separate file which is processed by the sknob tool
- The sknob library supports regular expression formation and some randomization in the option values
- Testbench configuration includes
  - Topology of component interconnect
  - Flow control and other DUT and testbench initialization parameters

Implementation

Invoking test cases

- Each test case is implemented in its own class and file
- Each test case compiled as individual object file and collected into a single shared-object file
- Individual test case can be selected by test class name string passed as simulation option
- The test launch class object gets the test case string through sknob and creates the test case instance by dynamic loading mechanisms, i.e. dlopen() and dlsym()
- Multiple test cases can be run in one simulation

Implementation

API for external VIP

- Bridge external cycle based, 2 state, C++ Verification IP
- Provides
  - clock event to advance the simulation cycle in the VIP
  - standardized C++ side 2-state drive and monitor interface
  - signal ports in C++ to SystemVerilog interfaces to drive and sample Verilog signals
  - simulation phase and status information handshake
  - unified messaging functions

Conclusion and Future Work

Conclusions

- Reuse proof of concept model in verification phase
- Accelerated design verification testbench development
- Successfully integrated C++ based external VIP
- Quick bug turn around time in CSIM components due to fast model rebuild time and light simulation overhead

Future Work

- Transition to UVM
  - Utilize SystemVerilog randomization and constraint solver for packet class
  - Better debugging through DVE or Verdi
- Use public libraries
  - TLM based interface between C++ and SystemVerilog
  - Planning to use SystemC-based architectural model so UVM Connect library could be a good fit

Thank You