2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Bridge the Portable Test and Stimulus to UVM Simulation Environment

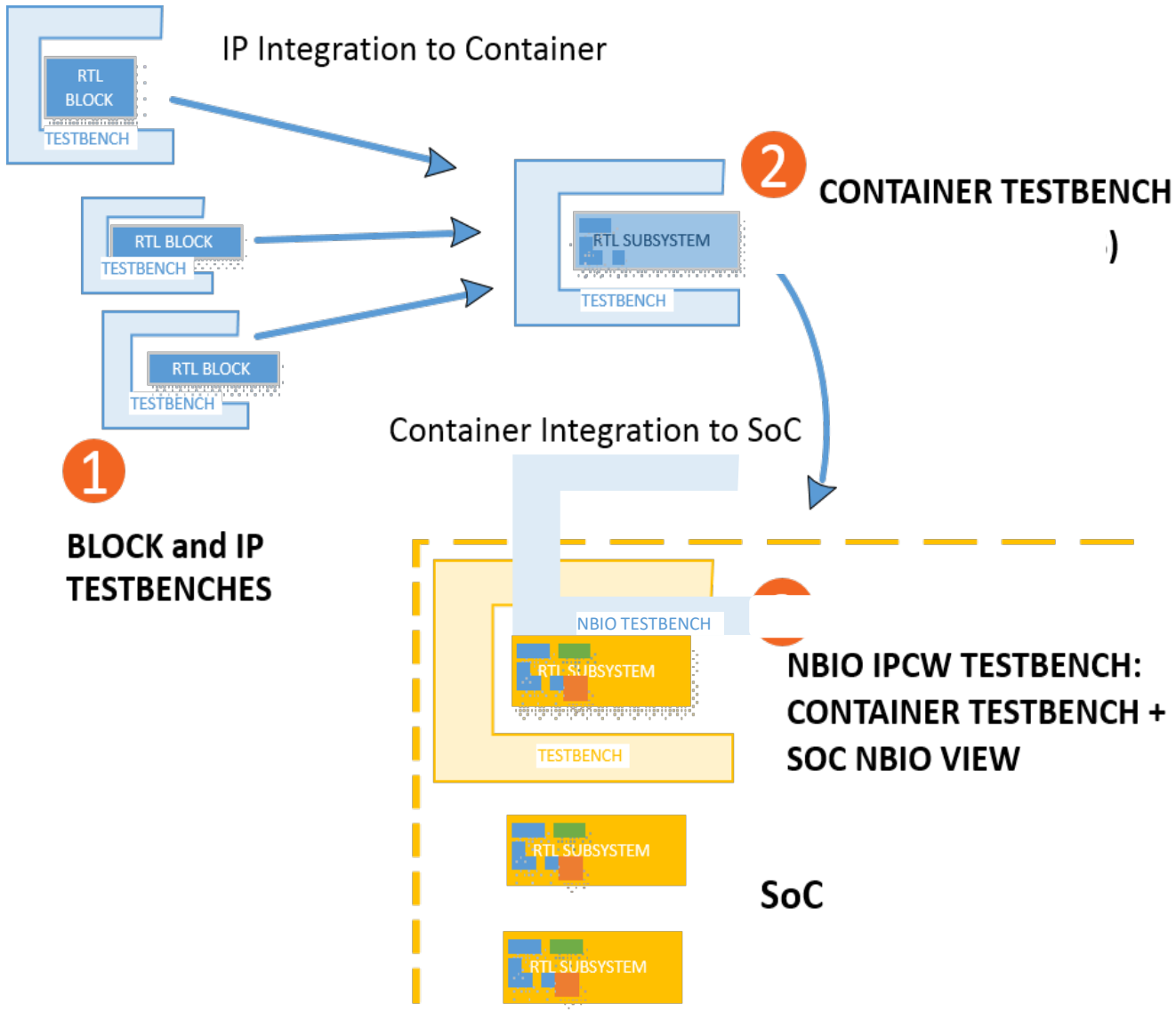
Theta Yang, Evean Qin

Advanced Micro Devices

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Problem

- Hierarchical Verification
  - IP level
  - Subsystem level
  - SOC level
- Challenge of vertical Verification Reuse
  - Each level compose simulation test independently
  - Change SOC/Subsystem level tests scenarios manually once IP logic/Testbench has changed
- Solution
  - Specify the SOC/Subsystem level tests in high level of abstraction of Portable Stimulus Script
  - Synthesis PSS into concrete simulation tests through tool set
- Challenge
  - Simulation speed is not good to cover large number of scenarios



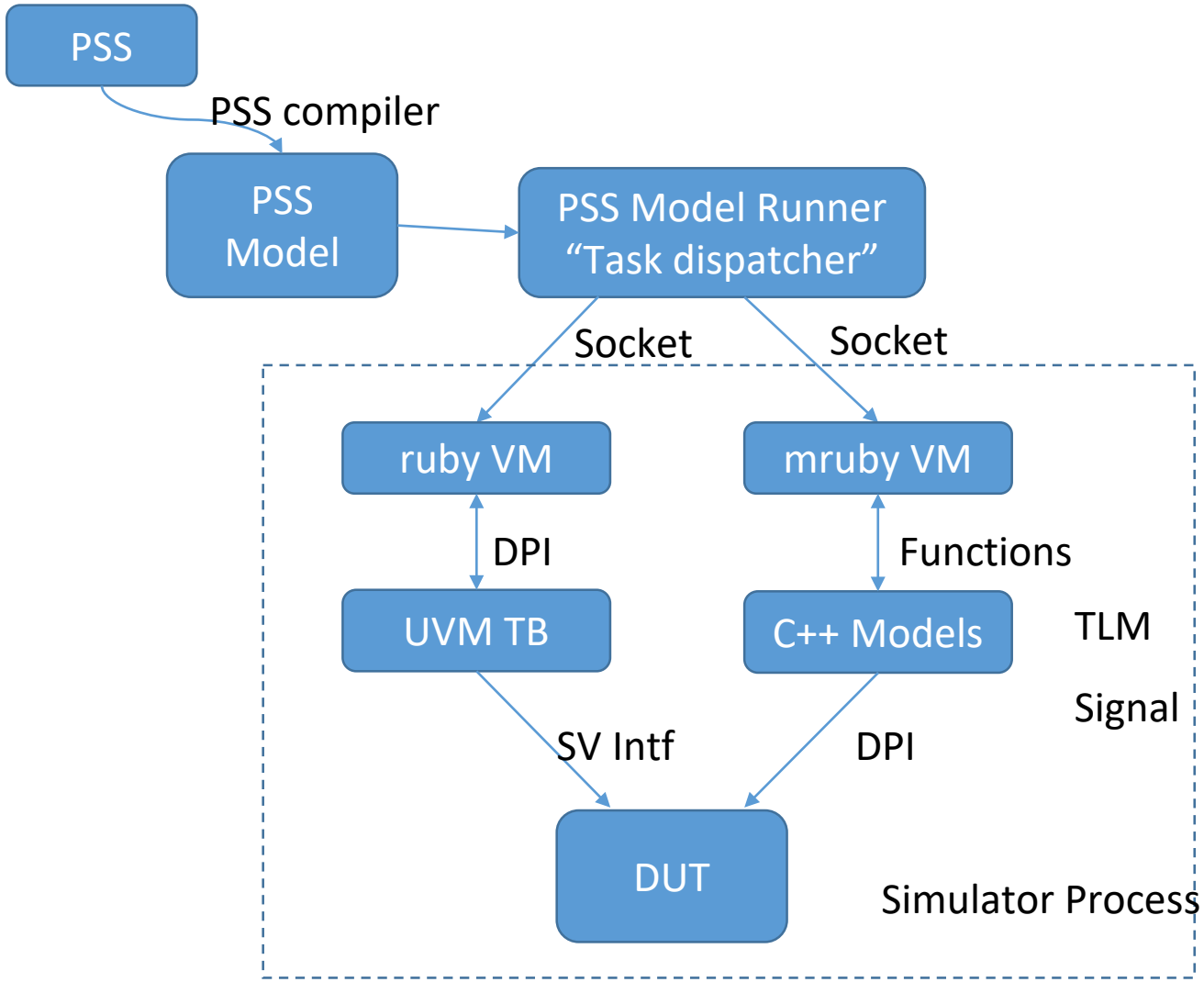
Handle DV complexity with hierarchical verification

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

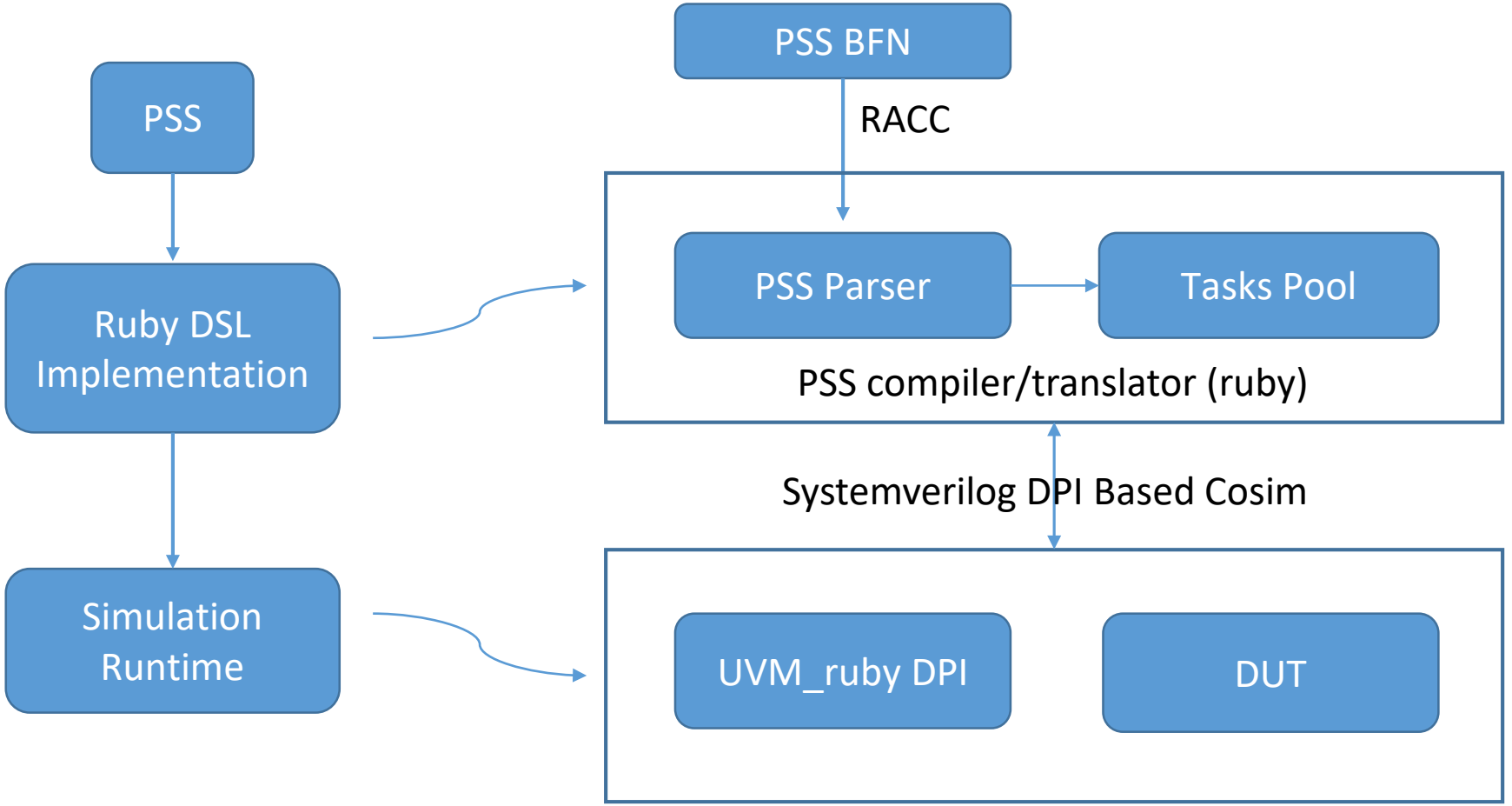
Portable Stimulus Enabling

- Descriptions of Requirement
  - Container/SOC reuses test sequence from IP level, a test sequence exerciser which provides test scenarios' parameterization, randomization and coverage is required to instrument the DUT thoroughly and leverage the IP common sequence efficiently.
  - Portable stimulus specification (PSS) from Accellera provides standard way of test reuse in language and simulation flow level.
- Goal,
  - Create testbench infrastructure will enable PSS as scenario description
  - PSS to test script conversion tools is under development, which supports,
    - Sequence exerciser
    - Randomization scenarios generation
    - System scenario coverage as DV metrics

PSS Verification Infrastructure

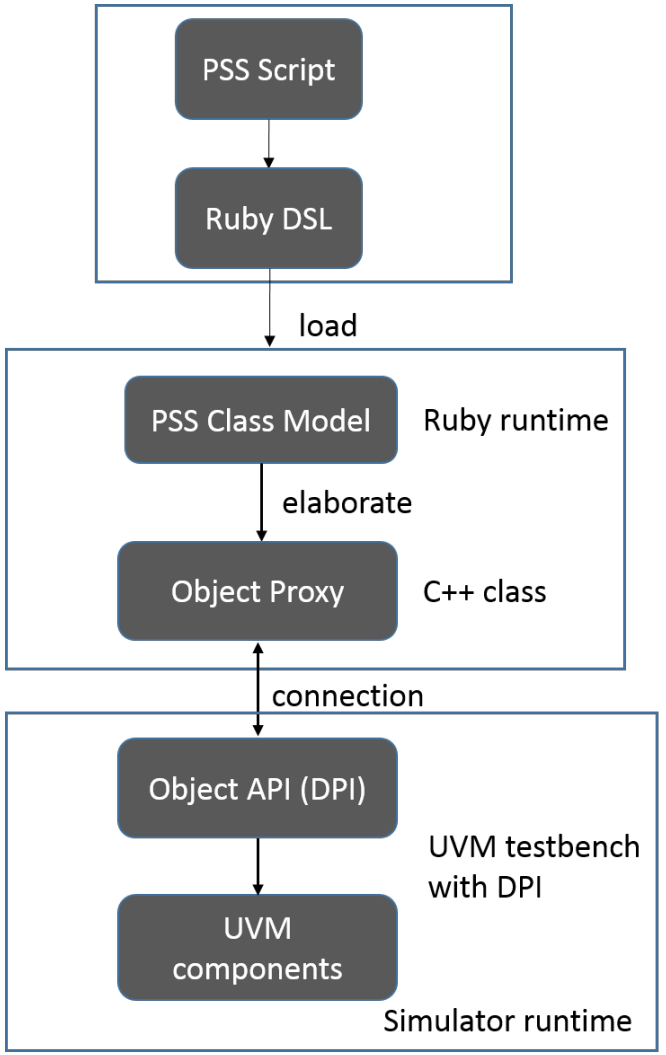


- Ruby Interpreter is embedded in simulator extension to accept PSS task
  - UVM sequence can be invoked through ruby run time through DPI to execute PSS
- PSS is converted into tasks which depend on each other
  - Tasks is managed with a runtime library which handle depend-graph
- Multiple mruby VM is supported as multiple agents to accept PSS tasks.
  - Agent is in ruby script


PSS
Ruby DSL Implementation
Simulation Runtime
PSS BFN
RACC
PSS Parser
Tasks Pool
PSS compiler/translator (ruby)
Systemverilog DPI Based Cosim
UVM_ruby DPI
DUT

PSS Parser

- The PSS script is compiled and converted into its ruby script equivalence.
- After evaluated by a ruby interpreter, the PSS translated ruby script will create ruby object model, which is the internal representation of the PSS abstract test intend.
- The PSS object model is based on a set of predefined ruby classes and implements the PSS functionality such as PSS component or PSS action.
- The PSS object model can run on and interact with a UVM testbench, with the help of ruby_uvm simulation runtime library.

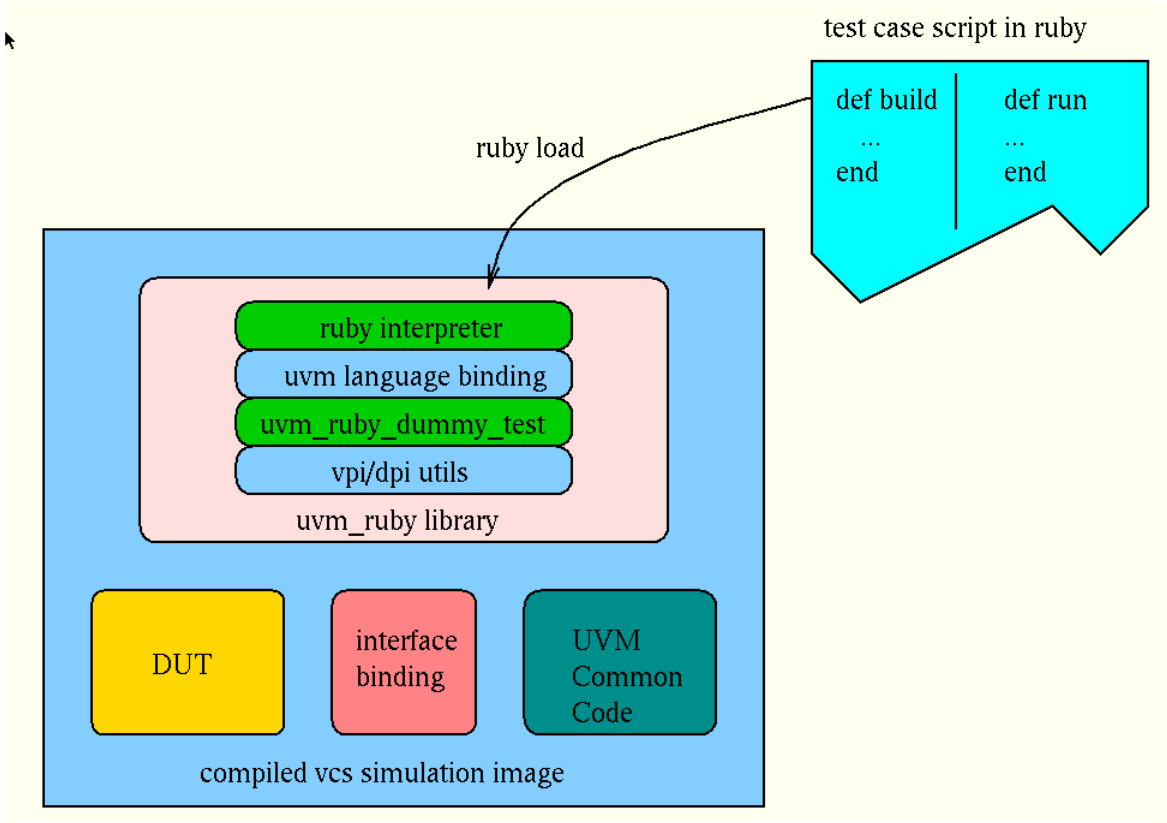


PSS_UVM Runtime library

- A ruby extension library is proposed to provides language binding with uvm, the popular hardware verification framework.
  - The library augments uvm with ruby's capability such as dynamic programming, easy to debug, and the comprehensive ruby library.
  - By using ruby on the top of UVM, the UVCs can be stitched and configured in ruby, and the test case can be developed in ruby as well.
  - As an interpretive language and it’s quick development nature of ruby, the efficiency the verification can be improved largely.
- The functions of major classes of uvm are exported to C through DPI, and though DPI functions are in turn binding to ruby.
  - With this uvm-ruby library, the functionalities of uvm such as uvm factory, phases callback, config_db, sequence and RAL access are provided in ruby, so that the test case can be developed in ruby instead of system Verilog.
  - The uvm-ruby library also provide the tlm connection API, combined with config_db and factory API, the uvm components used to construct a uvm testbench can be created and connected dynamically in ruby.
  - The library also provides DUT and uvm object inspection API, which enables use to debug the ruby testbench or testcase interactively.

Block diagram of UVM runtime

- Testcase in ruby specify how the test environment is constructed (in build callback function), and the contents of the test (in run callback function)
- The testcase script is loaded in simulation run time and uvm_env and uvm_test objects are created with the configuration from the script



Create ruby interpreter inside systemverilog

- Ruby interpreter creation DPI is called in an *initial* block of testbench systemverilog code
- Ruby interpreter is created by a C++ function which is spawned as a unix pthread. The pthread creation is imported to system Verilog as a DPI task and is invoked in an *initial* block of testbench.

SystemVerilog

```
import "DPI-C" context
task uvmc_init_ruby();


initial begin
        uvmc_init_ruby()
end
```

C function to create ruby interpreter

```
static void *init_ruby(void *) {
    ruby_init();
    ruby_init_loadpath();
    rb_require("uvmc.rb");
    return NULL;
}
void uvmc_init_ruby() {
    pthread_t init_ruby_pthread;
    pthread_create(&init_ruby_pthread, NULL, init_ruby, NULL);
    pthread_join(init_ruby_pthread, NULL);
    return;
  }
```

Feed the ruby uvm test to ruby interpreter

- When the simulation is running, the ruby interpreter is created and then the top level of ruby script is loaded. The ruby script defines callback functions corresponding to each uvm phase, these callback functions are registered into a predefined dummy uvm test.
- When the dummy uvm test is running, for each uvm_phase, the phase callback function will be called.
- A C++/ruby cross language library called “ruby rice” is used to pass the function call from C++ side to ruby side. Ruby rice uses multiple inherit to supports polymorphic calls travelling between C++ and Ruby seamlessly.
- Rice::Director is used to build a proxy class to properly send execution up or down the object hierarchy for our uvm_test class.

Feed the ruby uvm test to ruby interpreter


```
class VirtualBaseTest {
public:
  VirtualBaseTest();
  virtual int build() = 0;
  virtual int connect() = 0;
  virtual int run() = 0;
};
```

bind as base class of test in ruby,
the build/connect/run will be overloaded

```
class VirtualBaseTestProxy : public VirtualBaseTest,
                             public Rice::Director {
  public:
    VirtualBaseTestProxy(Object self) : Rice::Director(self) { }
    virtual int build() {
      return from_ruby<int>(getSelf().call("build") );     }
    virtual int connect() {
      return from_ruby<int>(getSelf().call("connect") );     }
    virtual int run() [
      return from_ruby<int>(getSelf().call("run") );     }

    int do_build() [return VirtualBaseTest::build();]
    int do_connect() {return VirtualBaseTest::connect();}
    int do_run() {return VirtualBaseTest::run();}
};
```

The ruby rice director class will tranverse
the call from C++ side to ruby side

```
VirtualBaseTestProxy *ProxyPtr;

int do_build() {return ProxyPtr->do_build();}
int do_connect() {return ProxyPtr->do_connect();}
int do_run() {return ProxyPtr->do_run();}
```

The test case in ruby is registered in ProxyPtr
The do_build/do_connect/do_run is added into
phase callback in uvm_ruby_dummy_test

Export UVM functionality to ruby using ruby rice

- Rice is a C++ interface to Ruby's C API. It provides a type-safe and exception-safe interface in order to make embedding Ruby and writing Ruby extensions with C++ easier. It is similar to Boost.Python in many ways, but also attempts to provide an object-oriented interface to all of the Ruby C API.
- What Rice gives you:
  - A simple C++-based syntax for wrapping and defining classes
  - Automatic conversion of exceptions between C++ and Ruby
  - Smart pointers for handling garbage collection
  - Wrappers for most builtin types to simplify calling code

Export uvm factory API to ruby

```
export "DPI-C" function UVMC_set_factory_type_override;
export "DPI-C" function UVMC_debug_factory_create;
export "DPI-C" function UVMC_find_factory_override;
.....

function automatic void
UVMC_print_factory (int all_types=1);
   uvm_factory factory = uvm_factory::get();
   factory.print(all_types);
endfunction

function automatic void
UVMC_set_factory_inst_override(string requested_type,
                               string override_type,
                               string contxt);
   uvm_factory factory = uvm_factory::get();
   factory.set_inst_override_by_name(requested_type,
                               override_type,contxt);
endfunction
```

1. Export sv uvm factor functions to C

```
class UvmFac {
public:
  create_instance_by_name(string type,
                          string inst_path,
                          string name);

  create_component_by_name(string type,
                           string inst_path,
                           string name);
  ...
};


define_class<UvmFac>("UvmFac")
          .define_method("create_instane_by_name",
                         &UvmFac::create_instance_by_name)
          .define_method("create_component_by_name",
                         &UvmFac::create_component_by_name)
```

2. Wrap the functions inside a C++ class, and create ruby wrapper with ruby rice

```
fac = UvmFac.new
fac.create_instance_by_name("bus_monitor", "*", "x_monitor");
```

3. Use the ruby class in ruby script

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Export uvm config db API to ruby

```
export "DPI-C" function UVMC_set_config_int;
export "DPI-C" function UVMC_set_config_string;
export "DPI-C" function UVMC_get_config_int;
export "DPI-C" function UVMC_get_config_string;



function
bit UVMC_get_config_string (string contxt,
                            string inst_name,
                            string field_name,
                            output string value);
   uvm_component comp;

   comp = get_context(contxt, "UVMC_GET_CFG_STR", 0);
   if (comp == null) begin
      if (inst_name == "")
        inst_name = contxt;
      else if (contxt != "")
        inst_name = {contxt, ".", inst_name};
   end
   UVMC_get_config_string=uvm_config_db#(string)
       ::get(comp, inst_name, field_name, value);

endfunction // UVMC_get_config_string
....
```

1. Export sv config db functions

```
#define __UVMC_DB_HPP_

class UvmDb {
public:
  set(string path, string field, p_vpi_vecval v);
  get(string path, string field, p_vpi_vecval v);
};
#endif //_UVMC_DB_HPP_


...
  define_class<UvmSignal>("UvmDb")
    .define_method("set", &UvmDb::set)
    .define_method("get", &UvmDb::get)
```

2. Wrap the functions inside a C++ class, and create ruby wrapper with ruby rice

```
db = UvmDb.new
db.get("*.comp.filed", value)
```

3. Use the ruby class in ruby script

Manipulate uvm sequence in ruby

- In System Verilog side, UVM Sequence is create and run in run_phase, the phase callback function is a task and consumes simulation time. In Uvm_ruby library we handle uvm_sequence class is similar to other uvm class like the uvm_config_db or uvm_factory, but…
- The uvm sequence is create to be executed, after created, the object and the sequencer on which the sequence will be run is registered into the uvm_ruby_dummy_test class.
- When we trigger the sequence run in ruby (by *seq.run*), the task “run_sequences” in uvm_ruby_dummy_test is trigerred. The task will executed all the registered sequence.
- Once the registered sequence is finished, the ruby function (seq.run) will return And after the all tasks ruby test’s run phase is complete, the callback function will return to uvm_ruby_dummy_test (run_phase) function.

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

conclusions

- The uvm_ruby library exposes the uvm API and utilities to ruby, and make the verification engineers can use ruby to create testbench and write testcases. As ruby is an interpretation based script language, it is much easier for engineer to stitch a uvm testbench and develop testcase in ruby than in system verilog natively.
- Uvm_ruby libraries improve the uvm library in a nonintrusive way. The runtime scope of uvm_ruby created objects are under the dummy uvm_test. Which means the deployment of uvm_ruby library will not affect the current uvm based test environment. If there are already parts of testcase being developed in native uvm, they will works as usual.
- Since the contents of ruby tests are not compiled into simulation execute image, if DUT and the uvm common code are not changed between simulation, but only the ruby code changes, we needn't to recompile it when run new tests.

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Conclusions-cond

- Ruby as a morden script language provides powerful debug environment, we can even do interactive simulation debug in the simulator independence way. We can halt the simulations, inspect into ruby program or check the signal value of DUT and they restore the simulation.
- Systemverilog language doesn't provide a mechanism to create new class through PLI/VPI, as a result, we can not create new system verilg class in C and ruby as well. But the uvm framework provides factory, config db, uvm_object registration and etc to make current uvm class instances configurable. The uvm_ruby library is based on those uvm configurable feature, as well as the system verilog's cross language support (VPI, DPI), which is powerful enough to do hange verification for complex SOC chips.

Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION