Augmenting a C++/PLI/VCS Based Verification Environment With SystemC

Dr. Ambar Sarkar
Paradigm Works Inc.
Andover MA
Agenda

• Why integrate?
• C++/PLI Based Environments
  – Verilog-on-top approach
• SystemC/SCV Environments
  – SystemC-on-top approach
• Integrated Environment
  – Incorporate SystemC transactors into legacy environment
• Integration Challenges
  – Interaction between legacy objects and SystemC transactors
• Conclusions
Why Integrate?

- HVLs (e, Vera, SystemC/SCV) now state-of-the-art
- Projects want to adopt HVLs for their next project
  - Preserve existing verification code
- SystemC is C++ code
- Allows a evolutionary path towards adopting a HVL
Augmenting a C++/PLI/VCS Based Verification Environment With SystemC

C++/PLI Based Environments

- PLI based C/C++
- side of transactors
- Test Driven
- Test

- IPC
- PLI Code/Low-level BFM
- Simulator Driven

- C/C++
- Verilog
- Verilog Top-level
- DUV
C++/PLI Based Environments -- Issues

- Indirect interaction with Verilog
- Unavailability of HVL features at all levels of abstraction
- Synchronizing simulation time
- Verilog-on-top model
PLI Environment Issues -- Indirect interaction with Verilog

- Fine grain control of stimulus not easily attained
  - Needed for verification

- C code does not directly manipulate DUT signals
  - High level transaction requests are queued from the C side
  - Only the verilog code directly manipulates the signal
  - Significant effort needed to observe/manipulate Verilog signals directly from C side

- Hard to synchronize with events on the Verilog side
PLI Environment Issues -- Unavailability of HVL features at all levels of abstraction

- Verilog used at low level transactor implementation
- Following advanced features thus not available
  - Pointers
  - Standard Template Library
  - Reentrant tasks
PLI Environment Issues -- Synchronizing simulation time

• C++ side has no notion of time
  – Blocking calls
  – Non-blocking calls

• Hard to synchronize based on simulation time

• Requires extra work
PLI Environment Issues -- Verilog-on-top model

• Verilog side is the topmost object hierarchically
  – Controls advancement of time
  – C side initiated with PLI call from Verilog side

• Top-level testbench code is Verilog
  – Typically requires more work compiling test code
  – Less convenient than writing top-level code using HVL
SystemC/SCV Environments

• SystemC
  – A C++ based hardware verification language
  – Has built in notion of simulation time
  – Standardized, non proprietary
  – Has constructs that can mimic hardware languages such as Verilog
  – More like a HDL

• SCV (SystemC Verification Library)
  – Additional library dedicated to verification
  – Full support for constrained randomization
SystemC/SCV Environments – Typical Implementations

Test → SystemC Transactors → DUV

Verilog Top-level
SystemC/SCV Environments – Advantages

- Direct interaction with DUV
- HVL features available at all levels of abstraction
- Built-in synchronization between Verilog and SystemC
- Follows HVL-on-top model
Ambar Sarkar
Paradigm Works

Augmenting a C++/PLI/VCS Based Verification Environment With SystemC

Integrated Environment

Test

IPC

SystemC Transactors

PLI based C/C++
side of transactors

C/C++ Domain

PLI Code/ Low-level BFM

Verilog Domain

Verilog Top-level

DUV
Integration Challenges

- Linking with VCS
- Compiling with legacy transactors
- Instantiating SystemC Transactors
- Multiple instantiations
- Communicating between SystemC and PLI transactors
- Error Reporting
- .. and so on
Integration Challenges – Linking with VCS

• Need to integrate OSCI reference implementation with VCS
  – Download reference implementation from OSCI
  – Minor source code changes
    Couple of files provided by Synopsys
    Provides DKI based interaction between
    SystemC reference implementation kernel
    VCS Simulator

• Different simulators have different approaches
Integration Challenges – Compiling with Legacy Transactors

- **syscan utility**
  - Generates Verilog wrapper code
  - Compiles the relevant SystemC files
  - Links resulting object files with the objects generated from the other verilog files.

```
syscan -cpp <cpp> -Mdir=<work_csrc_path> <srcfiles> -cflags <cflags>
```

Where:
- `<cpp>` is the path to the SystemC compatible C++ compiler
- `<work_csrc_path>` is the path to the output directory for the VCS compilation and should be the same as the one used for the legacy environment
- `<srcfiles>` list of files that declare and define the SystemC modules
- `<cflags>` any relevant C compilation flags including those used in the legacy environment
Integration Challenges – Compiling with Legacy Transactors

• Several gotchas
  – C++ compiler version
    Older version used by legacy
    Different versions needed for Solaris/Linux
  – Incompatibility of legacy code with SystemC

```c++
scvproxy.h

#ifndef __SCVPROXY_H__
#define __SCVPROXY_H__
#include "scv.h"

void RegisterWrite(uint addr);

#endif
```

```c++
scvproxy.cc

#include "scvproxy.h"
#include "Register.h"
void RegisterWrite() {
    Register::Write(addr);
}
```

```c++
....
```
Integration Challenges – Instantiating SystemC Transactors

- Transactors instantiated in Verilog Domain (PLI code)
  - Instance handle not available from C side directly
    - Transactors instantiated by Verilog through PLI, not C
    - Cannot access handle of Verilog PLI created transactor from C/C++ testbench code
  - Cannot just create an instance on C side and expect it to work
    - Instance created from C side cannot access Verilog signals
    - Need to use the instance created by Verilog PLI
- Use C++ static methods to pass handles between domains
  - static ScXactor * ScXactor::GetHandle();
Integration Challenges – Instantiating SystemC Transactors

ScXactor.h

```c
SC_MODULE(ScXactor) {
public:

    SC_CTOR(ScXactor) { …
        m_pScXactor = this;
    }

    // Access this instance
    static ScXactor * GetHandle() {
        return m_pScXactor;
    }

    void DoFoo();

private:
    static ScXactor *m_pScXactor;
};
```

ScXactor.cc

```c
// This will get initialized only when the
// constructor is called
ScXactor * ScXactor::m_pScXactor = 0;
```

Test.cc

```c
#include “ScXactor.h”

ScXactor *pXtr;

…

    pXtr = ScXactor::GetHandle();
    if (!pXtr) {
        // Print error
        exit();
    }

    pXtr->DoFoo();
```
Integration Challenges – Multiple SystemC Transactors

• Multiple instantiation of same transactors quite common
  – Different instances for each interface port

```c
SC_MODULE(ScXactor) {
  public:
    sc_in <sc_logic> id, clk;
    sc_inout <sc_lv<16>> data;

  SC_CTOR(ScXactor):
    id("id"), clk("clk"), data("data")
    {
      // The following would not work
      m_pScXactor0 = (id = 0) ? this:0;
      m_pScXactor1 = (id = 1) ? this:0;
    }
  private:
    static ScXactor *m_pScXactor0;
    static ScXactor *m_pScXactor1;
}
```

testbench.v

```c
ScXactor ScXactor0(.id (1'b0), .clk(clk),
  .data(data));
ScXactor ScXactor1(.id (1'b1), .clk(clk),
  .data(data));
```

Does not work!

• Wrapper code initialization not complete
• Crash or hang!
Integration Challenges – Multiple SystemC Transactors

- Checking instance id done outside constructor
- Need to wait before the passed in port id can be read by transactor

**ScXactor.h**

```cpp
SC_CTOR(ScXactorT):
    id("id"), clk("clk"), data("data")
{
   // The following workaround is needed
   // in case of multiple instantiations. The
   // thread will eventually read the id port
   // and set the pointers to transactor
   // instances accordingly
   SC_THREAD(detect_id_thread);
   ...
}
```

```cpp
void detect_id_thread();
```

**ScXactor.cc**

```cpp
// The following thread reads the assigned
// value to the id port and sets the instance
// pointers accordingly
void ScXactorT::detect_id_thread()
{
   // Wait for a small time to make sure the
   // SystemC reads the value of id
   wait(1, SC_NS);
   if (id == 0)
      m_pScXactor0 = this;
   else
      m_pScXactor1 = this;
}
```
Integration Challenges – Communicating between SystemC and PLI Transactors

- Testbench code can call both transactors
- Legacy transactor can call SystemC
  - Simulation time can advance in SystemC transactor method
- SystemC transactor can call Legacy
  - Except for blocking calls
    - Simulation time cannot advance in legacy transactor method
  - Workaround using asynchronous calls and polling

Instead of:

```cpp
read_data = legacyTransactor.read_blocking(addr);
```

Use:

```cpp
legacyTransactor.read_non_blocking(addr);
while (!LegacyTransactor.read_done()) wait();
```
Integration Challenges – Error Reporting

• Need a common error reporting API
  – SystemC/SCV has extensive support
  – Legacy environment as its own error reporting

• Need to forward SystemC/SCV system error messages as well
  – Failures during randomization

• Override default SystemC report handler class
Integration Challenges – Error Reporting

• Override default SystemC report handler

```cpp
class ProjectReportHandler: public scv_report_handler {
public:
  static void report(
    scv_severity severity,
    scv_msg_type msg_type,
    const char * msg,
    const char *file, int line
  ) ;
};

ProjectReportHandler project_report_handler;
scv_report_handler::set_handler(project_report_handler);
```
Integration Challenges – Error Reporting

```c++
void ProjectReportHandler::report(scv_severity severity, ... {
    string tmStmp = sc_time_stamp().to_string();
    switch (severity) {
    case SCV_INFO:
        LEGACY_INFO("%s:Time: %s: %s\n", msg_type, tmStmp.c_str(), msg);
        break;
    case SCV_WARNING:
        LEGACY_WARNING("%s:Time: %s: %s\n", msg_type, tmStmp.c_str(), msg);
        break;
    case SCV_ERROR:
        LEGACY_ERROR("%s:Time: %s: %s\n", msg_type, tmStmp.c_str(), msg);
        break;
    case SCV_FATAL:
        LEGACY_FATAL("%s:Time: %s: %s\n", msg_type, tmStmp.c_str(), msg);
        break;
    }
```
Conclusions

- Feasible to incorporate SystemC support into Legacy environments
- Takes effort to integrate SystemC transactors to the fullest extent
- Integrated environment has all benefits of SystemC while preserving legacy vestments