Case Studies in SystemC

Hints and Tips for Exploiting the Latest Features of SystemC

John Aynsley, Doulos
Agenda

- sc_vector
- sc_event_or_list
- sc_writer_policy
- reset_signal_is
- reset() and kill()

Features first introduced in IEEE 1666™-2011 and SystemC-2.3.0

SystemC-2.3.1 is about to be released
struct Child: sc_module
{
    sc_vector< sc_out<int> > port_vec;
    Child(sc_module_name n)
    : port_vec("port_vec", 4)
    {
        ...
    }
}

struct Top: sc_module
{
    sc_vector< sc_signal<int> > sig_vec;
    Top(sc_module_name n)
    : sig_vec("sig_vec", 4)
    {
        c = new Child("c");
        c->port_vec.bind(sig_vec);
    }
    ...
}
sc_vector of Modules

```c
struct Child: sc_module
{
    sc_out<int> p;
    ...
}

struct Top: sc_module
{
    sc_vector< Child > mod_vec;
    sc_vector< sc_signal<int> > sig_vec;

    Top(sc_module_name n)
    : mod_vec("mod_vec")
    , sig_vec("sig_vec")
    {
        mod_vec.init(4);
        sig_vec.init(4);
        for (int i = 0; i < 4; i++)
            mod_vec[i].p.bind(sig_vec[i]);
    }
    ...
}
```

- Anything derived from sc_object
- Elements are named
- Size deferred
sc_vector methods

```cpp
struct M : sc_module {
    sc_vector< sc_signal<int> > vec;

    M(sc_module_name n)
    : vec("vec", 4) {
        SC_THREAD(proc)
    }
    void proc() {
        for (unsigned int i = 0; i < vec.size(); i++)
            vec[i].write(i);

        wait(SC_ZERO_TIME);

        sc_vector< sc_signal<int> >::iterator it;
        for (it = vec.begin(); it != vec.end(); it++)
            cout << it->read() << endl;

    ...
```
Binding Vectors

```cpp
sc_vector< sc_port<i_f> >::iterator it;
it = m1->port_vec.bind( m2->export_vec );

it = m1->port_vec.bind( m3->export_vec.begin(), m3->export_vec.end(), it );
```

1st unbound element

Start binding here
Constructor Arguments

```c
struct Child: sc_module
{
    Child(sc_module_name n, int a, bool b);
    ...
}

sc_vector<Child> child_vec;

static Child* creator_func( const char* name, size_t s )
{
    return new Child(name, 3, true);
}

child_vec.init(4, creator_func);
```

- Module ctor has args
- Pass args to constructor
Agenda

- sc_vector
- sc_event_or_list
- sc_writer_policy
- reset_signal_is
- reset() and kill()
Event List Objects

```c++
struct Child : sc_module
{
    sc_vector< sc_port<i_f> > ports;

    Child(sc_module_name n, int n_ports)
    : ports("ports", n_ports)
    {
        SC_HAS_PROCESS(Child);
        SC_THREAD(thread);
    }

    void thread()
    {
        sc_event_or_list or_list;
        sc_vector< sc_port<i_f> >::iterator it;
        for (it = port.begin(); it != port.end(); it++)
            or_list |= (*it)->default_event();
        for (;;)
        {
            wait(or_list);
            ...
        }
    }
};
```

Ctor arg
Setup once, keep it alive!
Agenda

- sc_vector
- sc_event_or_list
- sc_writer_policy
- reset_signal_is
- reset() and kill()
Modeling an Interrupt

```cpp
struct M: sc_module {
    sc_vector< sc_signal<bool, SC_MANY_WRITERS> > interrupt;

    M(sc_module_name n) : interrupt("interrupt", 4) {
        SC_HAS_PROCESS(M);
        SC_THREAD(proc1);
        SC_THREAD(proc2);
    }

    void proc1() {
        wait(100, SC_NS);
        interrupt[0].write(true);
        wait(1, SC_NS);
        interrupt[0].write(false);
    }

    void proc2() {
        wait(200, SC_NS);
        interrupt[0].write(true);
        wait(1, SC_NS);
        interrupt[0].write(false);
    }```

© 2014 Doulos
March 3, 2014
Agenda

- sc_vector
- sc_event_or_list
- sc_writer_policy
- reset_signal_is
- reset() and kill()
Processes Unified!

```
SC_CTHREAD(T, clk.pos());
    reset_signal_is(r, true);
    async_reset_signal_is(ar, true);

SC_THREAD(T);
    sensitive << clk.pos();
    reset_signal_is(r, true);
    async_reset_signal_is(ar, true);

SC_METHOD(M);
    sensitive << clk.pos();
    reset_signal_is(r, true);
    async_reset_signal_is(ar, true);
```

Reset restores the static sensitivity

```
void T() {
    if (r|ar)
        q = 0;
    while (1)
        { wait();
        ++q;
        }
}

void M() {
    if (r|ar)
        q = 0;
    else
        ++q;
}
Styles of Reset

SC_THREAD(target);
reset_signal_is(reset, active_level);
async_reset_signal_is(reset, active_level);

sc_spawn_options opt;
opt.reset_signal_is(reset, active_level);
opt.async_reset_signal_is(reset, true);

handle.reset();

handle.sync_reset_on();
...
handle.sync_reset_off();
Agenda

- sc_vector
- sc_event_or_list
- sc_writer_policy
- reset_signal_is
- reset() and kill()

Process control methods
reset and kill

```c
SC_THREAD(calling);
SC_THREAD(target);

int q;

void calling()
{
    wait(10, SC_NS);
    ev.notify();

    wait(10, SC_NS);
    t.reset();
    assert( q == 0 );

    wait(10, SC_NS);
    ev.notify();

    wait(10, SC_NS);
    t.kill();
    assert( t.terminated() );
}

void target()
{
    q = 0;
    while (1)
    {
        wait(ev);
        ++q;
    }
}
```

Wakes at 10 20 30
Terminated at 40
void target()
{
    q = 0;
    while (1)
    {
        try {
            wait(ev);
            ++q;
        }
        catch (const sc_unwind_exception& e)
        {
            reset();
        }
        kill();
    }
}
void target()
{
    q = 0;
    while (1)
    {
        try {
            wait(ev);
            ++q;
        }
        catch (const sc_unwind_exception& e)
        {
            sc_assert(sc_is_unwinding());
            if (e.is_reset()) cout << "target was reset";
            else cout << "target was killed";
        }
    }
    ...
void target()
{
    q = 0;
    while (1)
    {
        try {
            wait(ev);
            ++q;
        }
        catch (const sc_unwind_exception& e)
        {
            sc_assert( sc_is_unwinding() );
            if (e.is_reset()) cout << "target was reset";
            else cout << "target was killed";
            proc_handle.reset();
            throw e;
        }
    }
}

reset()  \rightarrow  kill()
reset_event & terminated_event

```cpp
SC_THREAD(calling);
SC_THREAD(target);
    t = sc_get_current_process_handle();

SC_METHOD(reset_handler);
    dont_initialize();
    sensitive << t.reset_event();

SC_METHOD(kill_handler);
    dont_initialize();
    sensitive << t.terminated_event();

void target()
{
    ...
    while (1)
    {
        wait(ev);
        wait(10, SC_NS);
        t.reset();
        t.kill();
        wait(10, SC_NS);
    }
    ...
}
```
For Further Information

http://www.doulos.com/knowhow/systemc/

http://www.accellera.org/community/systemc/

http://www.accellera.org/downloads/standards/

Thank you
Case Studies in SystemC

TLM Use Cases at Ericsson AB

Henrik Svensson, PhD, Ericsson AB
1. Develop Base Stations
Radio Base Stations

Ericsson base stations support all major 3GPP and 3GPP2 technology tracks:
- GSM/EDGE
- WCDMA/HSPA
- CDMA
- LTE

Antenna
- Radio
- Baseband
Radio Base Stations
A look inside

- Usually with the term system or embedded system we refer to hardware and software.
Develop base stations

System-On-Chip
Baseband, Radio, and Control SoCs

System-On-Board
Radio Unit and Digital Unit

System-In-Cabinet
RBS 6000 series of multi-standard base stations
System Design Process

THE DESIGN PROCESS AS A FLOW OF ACTIVITIES

- System
  - Hardware
  - Software

TLM

- Architect System
- HW IP cores D&V
- HW Chip D&V
- HW Board D&V
- SW TOOLS
- Platform & App

© 2014 Ericsson AB March 3, 2014
TLM advantages

Primary:
- Early Hardware Verification Environment
- Early Software Development and System Verification
- Enables early and massive System Exploration
- Enables early System Dimensioning

Secondary:
- IEEE Standard
- C++ offers a extensive code base
- Speed of Development
- Simulation and Verification Speed

POTENTIAL ADVANTAGES

MAIN ADVANTAGES: WITH ABSTRACTION

SECONDARY ADVANTAGES: E.G. SYSTEMC OVER OTHER ABSTRACT LANGUAGES.
Design flow with TLM

Functional model (C++) → LT model (SystemC) → AT model (SystemC) → RTL model (VHDL/…)

Algorithm D&V → Performance evaluation → SoC Design and Verification → HW Board D&V

SoC architects → HW designers → Platform SW dev. → Appl. SW dev. (RANs)

HW verifiers → Virtual prototyping → HW emulation

SW Tools → Platform SW (C/C++) → Application SW (C/C++)
Ericsson AB TLM

Community
- Ericsson joined Accellera as associate corporate level during 2012
- Ericsson became corporate level member 2014

Co-operating across projects and organization
- TLM steering group meetings every second week

Projects
- Methodology project – Apollo 2010
- Pilot project TLM for early SW development – Helios 2010
- Pilot project TLM for Architectural exploration – Vulcan 2011
- Sharp project TLM for early SW development – Ghost 2012
- Sharp project TLM for early HW verification – Atom 2012

Master thesis
- Accuracy of AT TLM models. How to compare RTL and TLM.
- TLM for virtual platforms
- TLM for verification
- SW statistics collection from TLM for HW exploration and dimensioning
2. SW Development
SW users at Ericsson

Today…

- Ericsson uses a TLM-2.0 LT based virtual platform for SW development
  - Hundreds of users
  - SW operates many months before chip and board is ready
  - Replaced legacy Virtual platform
SW Development

- Develop SW and run regressions
  - LT to get the speed
  - Memory/register accurate

- SW users operate at chip and board-level

- Standard debug tools are used independent if target is TLM or HW.

- More visibility with tools from EDA vendors.
System virtualization platform, single ASIC

- **Test framework support**
- **3rd party Debug tool support**

**DspTools support (Ericsson internal)**

- **Debugger**
- **MDI**
- **DspTools Core**
- **QSim**
- **Legacy Virtual platform**

**SVPIF client**

**TCP/IP socket**

**In-house CCI framework**

**SVP**

**Top**

**ASIC (TLM2)**

- **Accelerators**
- **Interfaces**
- **Control**
- **DSP cluster**

**I/O**

**Adapter**

**TCP/IP socket**

**SVPIF server**

**Debug**

**Attribute**

**Execute**

**Factory**
System virtualization platform, board

In-house CCI framework

SVP

Board

CPU

TLM2 IP

FPGA (TLM2)

ASIC (TLM2)

I/O

Adapter

TCP/IP socket

3rd party TLM2 Simulator support

Test framework support

Debug

Attribute

Execute

Factory

SVPIF server

1. TCP/IP socket
2. 3rd party TLM2 Simulator support
3. Test framework support
4. Debug
5. Attribute
6. Execute
7. Factory
8. SVPIF server

© 2014 Ericsson AB  March 3, 2014
In house CCI Module

- Must include header file
  ```
  #include "svp.h"
  ```

- Should use `svp_module`
  - provides attribute handler
  - dynamic creation facilities
  ```
  class Mymodule :
    public svp_module { public:
      Mymodule(svp_modulename nm) :
        svp_module(nm)
      {
      }
    }
  ```

- Otherwise regular `sc_modules`

- Must comply with modeling guidelines
In house CCI Attributes

- Store configuration data and state
- Are data members of SVP modules
- Can be used like regular data members inside the module
- Accessible from outside the class

Two ways to deploy
- Use ATTRIBUTE macro (preferred)
- Explicitly instantiate attribute class

Internally attributes are linked to AttributeContainers
- Used to form attribute trees
- Orthogonal to sc hierarchy
- svp modules are attribute containers

```cpp
class my_module : public svp_module {
  my_module(svp_module_name smn) 
    : svp_module(smn), a (123) 
    
  { b = 456; }

  SVP_BASIC_ATTRIBUTE(a, uint32_t);
  SVP_BASIC_ATTRIBUTE(b, uint32_t);
};
```
In house CCI registers

- Store SW visible data and state
- Are members of SVP modules
- Organized hierarchically
  - module.{bank.}register.field
- Support access restrictions (r/w/rw)
- Accessible from outside the class
  - by SVP handlers and tool clients
- Provide APIs for reset, field-wise access, ...
  - eases internal usage and access
  - full register or individual fields
- Optional creation from reg descriptions
  - SystemRDL, IP-XACT
  - Handy for large register files (100s of regs)

```cpp
// could be generated (MACRO|TOOL)
template <T> class RegXY
 : public er_register<T> {
  er_register_field<T,sBit,eBit,RW> f1;
  er_register_field<T,sBit,eBit,RW> f2;
  myReg(RegisterName an, T resetVal)
 : er_register<T>(an, resetVal)
   , f1("f1", this), ... 
  {} 
};

class my_module
 : public svp_module {
  RegXY<
```
3. Performance evaluation
Performance evaluation

Today…

- TLM-2.0 AT models used for HW IP level performance exploration
- TLM-2.0 LT models used at system-level to acquire SW load models
Roadmap for architectural exploration

- It is not in the current roadmap to develop AT models of complete ASICs or boards
- AT modeling used at IP level or subsystem level to do exploration or dimensioning
- LT level have shown accuracy enough to be used for
  - System level exploration
  - To acquire SW load model that is used in dimensioning or exploration
Performance Evaluation

- **Performance Evaluation** is to quantify the services given by the system as a function of scenarios:
  - Quantifying services = **Metric** or performance metric
  - Scenarios = **Stimuli** or load

**Goals are:**

- **Performance dimensioning**
  - Assure the selected system meets performance goals.
  - Metric **high accuracy**

- **Performance exploration**
  - Performance evaluation of multiple systems that are then compared.
  - Metric **medium accuracy** (Relative comparison)
SW load models acquired from TLM LT

- Use SVP to collect samples from SW applications.
  - Usage of buses
  - Usage of memories
  - Usage of HW cores
  - etc.

- Make statistical analysis of the samples

- Build a flexible TLM traffic generator targeting performance evaluation.

- Used also in static analysis
IP level exploration

AT Model vs. Original RTL

Later RTL vs. Original RTL after boosting

30% Performance boosting!
4. HW functional verification
HW Verification at Ericsson

- TLM-2.0 LT models are used as references at HW IP level
- TLM-2.0 LT models used as development platform for chip level verification
Chip level verification

1. Development phase
   - Test case SW
   - TLM

2. Regression phase
   - Test case SW
   - RTL
   - Emulator
   - Target

<table>
<thead>
<tr>
<th>Platform</th>
<th>Relative load time + execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL</td>
<td>15000x (days)</td>
</tr>
<tr>
<td>HW emulation</td>
<td>80x (minutes)</td>
</tr>
<tr>
<td>TLM LT</td>
<td>1x (seconds)</td>
</tr>
</tbody>
</table>

**Advantages**
- TLM available early and used as stable development platform for SW driven test cases.
- Develop and run design loop is fast.
TLM in chip level verification

- Software driven Verification
- Same verification platform for all abstractions, TLM, RTL, Emulation and HW.
- This setup allows us as well to verify and debug software.
- TLM model used for test software development in this setup
- The verification environment supports both test software development and design verification.
- Possibilities to run real software will improve our confidence and pave the wave for a smooth software integration.
- Regressions are run on all abstractions
HW IP verification

- Top-down design approach
  - TLM as DUT and reference REF
    - High abstraction level
  - DUT later exchanged to matured RTL
    - Low abstraction level
Challenges

- How smart should the scoreboard be to identify the mismatch type?
- Is the observability enough?
  - Internal state divergence due to different model accuracy
HW verification: LT or AT?

- How can we move the REF model to low abstraction level for achieving the verification goal?
- How can we maximize the model reusability?
HW verification: reusable LT module

- Drive TLM at critical condition.
  - Increase detail level with less modeling effort
- TLM behaves partly as a state observer and checker.
  - Increase observability for debugging

Interoperability?

Event driven LT verification mode
Summarize LT and AT

- LT adds value to several user groups in HW and SW.
- Verification requires refined LT but not AT.
- LT models are useful for system exploration and to acquire SW load models.
- AT models used for IP level or subsystem level exploration and dimensioning.
Thank you
Case Studies in SystemC

Efficient Abstractions for AMS System-level Design

Martin Barnasconi, SystemC AMS WG chair
Outline

- Introduction
- Language standard overview
- AMS models of computation
- Dynamic and reactive extensions in SystemC AMS 2.0
- Example: DC motor control with Pulse Width Modulator
- Conclusions
SystemC AMS objectives

- System-level modeling standard and methodology for analog/mixed-signal systems
- An architecture design language for AMS system-level design and verification
- A platform that facilitates AMS model exchange and IP reuse
- SystemC-centric methodology to integrate (abstract) AMS and digital HW/SW descriptions
- Efficient abstraction concepts and modeling formalisms using dedicated models of computation (MoC)
SystemC AMS applications

Communication systems

Imaging systems

Automotive systems
SystemC AMS – History

1999: Open SystemC Initiative (OSCI) announced

2000: SystemC 1.0 released (sourceforge.net)

~2000: First C-based AMS initiatives (AVSL, MixSigC)

2002: OSCI SystemC 1.0.2

2002: SystemC-AMS study group started

2005: IEEE Std 1666-2005 LRM

2005: OSCI AMSWG installed

2005: SystemC Transaction level modeling (TLM) 1.0 released

2005: First SystemC-AMS PoC released by Fraunhofer

2006: SystemC AMS Draft 1 LRM

2007: SystemC 2.2 released

2008: SystemC AMS Draft 1 LRM

2009: SystemC TLM 2.0 standard

2009: IEEE Std 1666-2011 LRM

2010: SystemC AMS 1.0 LRM standard

2009: SystemC Synthesizable Subset Draft 1.3

2010: SystemC AMS 1.0 PoC released by Fraunhofer IIS/EAS

2011: IEEE Std 1666-2011 LRM

2012: SystemC AMS 2.0 draft standard

2012: SystemC 2.3 PoC released by Accellera Systems Initiative

2013: SystemC AMS 2.0 PoC test version

2013: SystemC AMS 2.0 LRM standard

2014: IEEE 1666.1 (SystemC AMS) started
Positioning SystemC AMS

- Functional
- Architecture
- Implementation

SystemC AMS

VHDL-AMS, Verilog-AMS

SystemVerilog, VHDL, Verilog
**Example: Communication System**

- **Tight interaction** between digital HW/SW and AMS sub-systems
  - Signal path: Communication protocol stack – modeling including PHY layer
  - Control path: more and more HW/SW calibration and control of analog blocks

- **Architecture modeling using SystemC, TLM and AMS**
Industry requirements and needs

- **Design of True Heterogeneous Systems-on-a-chip**
  - Analog, Mixed-signal, RF, digital HW/SW (processor) interaction
  - Multi-domain, high frequencies, high bandwidth, configurable AMS components

- **Support different levels of design abstraction**
  - Functional modeling, architecture design, (abstract) circuit representations

- **Support different use cases – also for AMS!**
  - Executable specification, architecture exploration, virtual prototyping, integration validation

Need for Virtual Prototype Environments which enable inclusion of digital HW/SW and abstract AMS/RF system-level representations
SystemC AMS advantages

- **SystemC, thus C++ based**
  - Enjoy the power of C++ (and its wealth of libraries)
  - Object oriented – modular and extendable
  - AMS class libraries available for basic building blocks (analog primitives)
  - Tool independent / EDA-vendor neutral

- **Modeling in multiple abstractions using one simulator**
  - No need for complex multi-kernel/co-simulation
  - No difficult APIs
  - Converter models and ports are part of the language
  - Allows abstraction along four axis
    - structure, behavior, communication and time/frequency

- **Transparent modeling platform**
  - Access to simulation kernel to ease debugging and introspection
Model abstraction and formalisms

Use cases

- Executable specification
- Virtual prototyping
- Architecture exploration
- Integration validation

Model abstractions

- Discrete-time static non-linear
  - Non-conservative behavior
  - Timed Data Flow (TDF)

- Continuous-time dynamic linear
  - Conservative behavior
  - Linear Signal Flow (LSF)

- Electrical Linear Networks (ELN)

Modeling formalism
AMS models in Virtual Prototypes realistic?

Yes, as long as you use the right language and abstraction method

Expected simulation speed improvement [1]

SystemC AMS extensions LRM

- Language Reference Manual defines the standard of the SystemC AMS extensions

- Contents
  - Overview
  - Terminology and conventions
  - Core language definitions
  - Predefined models of computation
  - Predefined analyses
  - Utility definitions
  - Introduction to the SystemC AMS extensions (Informative)
  - Glossary (Informative)
  - Deprecated features (Informative)
  - Changes between SystemC AMS 1.0 and 2.0 standard (Informative)
SystemC AMS User’s Guide

- Comprehensive guide explaining the basics of the AMS extensions
  - TDF, LSF and ELN modeling
  - Small-signal frequency-domain modeling
  - Simulation and tracing
  - Modeling strategy and refinement methodology

- Many code examples

- Application examples
  - Binary Amplitude Shift Keying (BASK)
  - Plain-Old-Telephone-System (POTS)
  - Analog filters and networks

- Has proven it’s value: reference guide for many new users
# SystemC AMS language features

**Mixed-Signal Virtual Prototypes**
written by the end user

<table>
<thead>
<tr>
<th>SystemC methodology-specific elements</th>
<th>AMS methodology-specific elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction-level modeling (TLM), Cycle/Bit-accurate modeling, etc.</td>
<td>elements for AMS design refinement, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timed Data Flow (TDF)</th>
<th>Linear Signal Flow (LSF)</th>
<th>Electrical Linear Networks (ELN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td>modules</td>
<td>modules</td>
</tr>
<tr>
<td>ports</td>
<td>ports</td>
<td>terminals</td>
</tr>
<tr>
<td>signals</td>
<td>signals</td>
<td>nodes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scheduler</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Linear DAE solver</th>
</tr>
</thead>
</table>

**Time-domain and small-signal frequency-domain simulation infrastructure** (synchronization layer)

**SystemC Language Standard (IEEE Std. 1666-2011)**
SystemC AMS methodology elements

- Support design refinement using different models of computation
  - Timed Data Flow (TDF) - efficient simulation of discrete-time behavior
  - Linear Signal Flow (LSF) - simulation of continuous-time behavior
  - Electrical Linear Networks (ELN) - simulation of network topology & primitives

- Using namespaces
  - Clearly identify the used model of computation
  - Unified and common set of predefined classes, (converter) ports and signals

- Examples
  - Module
    - sca_tdf::sca_module
    - sca_lsf::sca_module
  - Input port
    - sca_tdf::sca_in
    - sca_lsf::sca_in
  - Output port
    - sca_tdf::sca_out
    - sca_lsf::sca_out
  - Signals
    - sca_tdf::sca_signal
    - sca_lsf::sca_signal
  - Nodes (electrical only)
    - sca_eln::sca_node
  - Terminal (in/output port, electrical only)
    - sca_eln::sca_terminal
Timed Data Flow (TDF)

- **TDF is based on synchronous dataflow**
  - A module is executed if enough samples are available at its input ports
  - The number of read/written samples are constant for each module activation
  - The scheduling order follows the signal flow direction

- **The function of a TDF module is performed by**
  - reading from the input ports (thus consuming samples)
  - processing the calculations
  - writing the results to the output ports

- **The TDF model of computation is a discrete-time modeling style**
Linear Signal Flow (LSF)

- Continuous-time behavior described in the form of block diagrams
  - LSF primitives describe relations between variables of a set of linear algebraic equations

- Only a single quantity is used to represent the signal
  - There is no dependency between flow (e.g. current) and potential (e.g. voltage) quantities
  - Uses directed real-valued signals, resulting in a non-conservative system description
Electrical Linear Networks (ELN)

- ELN modeling style allows the instantiation of electrical primitives
  - Connected ELN primitive modules will form an electrical network

- The electrical network is represented by a set of differential algebraic equations
  - following Kirchhoff’s voltage law (KVL) and Kirchhoff’s current law (KCL)

- ELN captures conservative, continuous-time behavior
Dynamic Timed Data Flow modeling

- Abstract modelling of sporadically changing signals
  - E.g. power management that switches on/off AMS subsystems

- Abstract description of reactive behaviour
  - AMS computations driven by events or transactions

- Capture behaviour where frequencies (and time steps) change dynamically
  - Often the case for clock recovery circuits or capturing jitter

- Modelling systems with varying (data) rates
  - E.g. multi-standard / software-defined radio (SDR) systems

This requires a *dynamic* and *reactive* Timed Data Flow modeling style
- Basically introduce variable time step instead of fixed/constant time step
Example: DC motor control

- Functional model in the Laplace domain modelled in SystemC AMS
- To achieve high accuracy, many module activations are necessary when using fixed time steps (AMS 1.0)
- Introducing *Dynamic TDF* to only compute when necessary, due to dynamic time step mechanism (AMS 2.0)
DC motor control loop behavior

$i_{meas}(t)$

$t_{ramp}$

$t_{duty}$

$t_{period}$

$v_{drv}(t)$
Example of Pulse Width Modulator (1)

```cpp
// pwm_dynamic.h

#include <cmath>
#include <systemc-ams>

SCA_TDF_MODULE(pwm) // for dynamic TDF, we can use the same helper macro to define the module class
{
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;

    pwm( sc_core::sc_module_name nm, ... )
        : in("in"), out("out") {}

    void set_attributes()
    {
        does_attribute_changes(); // module allowed to make changes to TDF attributes
        accept_attribute_changes(); // module allows attribute changes made by other modules
    }

    void change_attributes() // new callback to change attributes during simulation
    {
        double t = get_time().to_seconds(); // current time
        double t_pos = std::fmod( t, t_period); // time position inside pulse period
        ...
    }
```

Dynamic TDF features indicated in red
if ( t_pos < t_ramp ) {
    // rising edge
    request_next_activation( t_ramp - t_pos, sc_core::SC_SEC );
} else if ( t_pos < t_ramp + t_duty ) {
    // plateau
    request_next_activation( ( t_ramp + t_duty ) - t_pos, sc_core::SC_SEC );
} else if ( t_pos < t_ramp + t_duty + t_ramp ) {
    // falling edge
    request_next_activation( ( t_ramp + t_duty + t_ramp ) - t_pos, sc_core::SC_SEC );
} else {
    // return to initial value
    request_next_activation( t_period - t_pos, sc_core::SC_SEC );
}

void processing()
{
    ... // PWM behavior
}

private:
    ... // member variables
};
### TDF vs. Dynamic TDF comparison

<table>
<thead>
<tr>
<th>TDF model of computation variant</th>
<th>$t_{\text{step}}$ (ms)</th>
<th>$t_{\text{ramp}}$ (ms)</th>
<th>$t_{\text{period}}$ (ms)</th>
<th>Time accuracy (ms)</th>
<th>#activations per period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional TDF</td>
<td>0.01 (fixed)</td>
<td>0.05</td>
<td>5.0</td>
<td>0.01 ($= t_{\text{step}}$)</td>
<td>500</td>
</tr>
<tr>
<td>Dynamic TDF</td>
<td>variable</td>
<td>0.05</td>
<td>5.0</td>
<td>defined by sc_set_time_resolution()</td>
<td>4</td>
</tr>
</tbody>
</table>

- **Comparison of the two variants of the TDF model of computation**
  - Conventional PWM TDF model uses a fixed time step that triggers too many unnecessary computations
  - When using Dynamic TDF, the PWM model is only activated if necessary.
Summary and outlook

- SystemC AMS developments are fully driven and supported by European industry: NXP, ST, Infineon, and Continental
  - Applications: communication, automotive and imaging systems design

- SystemC AMS is a mature and proven standard
  - SystemC AMS 1.0 was released in March 2010, introducing efficient AMS modeling and system-level simulation
  - SystemC AMS 2.0 was released in March 2013, introducing reactive and dynamic behavior for AMS computations

- Third party Proof-of-Concept implementation for SystemC AMS 1.0 available under Apache 2.0 license
  - Thanks to Fraunhofer IIS/EAS Dresden

- Commercial design environments supporting SystemC AMS are available in the market
More information

- www.accellera.org
- www.accellera.org/downloads/standards/systemc/ams
- www.accellera.org/community/articles/amssspeed
- www.accellera.org/community/articles/amsdynamictdf
- www.systemc-ams.org
Thank you
Case Studies in SystemC

UVM for SystemC Users

John Stickley, Mentor Graphics
Gordon Allan, Mentor Graphics
UVM for SystemC users

- The case for hybrid testbenches
- Standards for hybrid testbenches
- What is UVM-Connect?
- UVM-Connect primer
- Reuse of legacy IP models
- Interchangeable testbench heads
- Dual use UVM drivers
- Hybrid testbench examples
- Summary
The case for hybrid testbenches

- Mounting use of TLM 2 as the common interconnect for multi-language HVL testbench environments:
  - TLM-2 standard is already “baked in” to IEEE 1666 SystemC and SV-UVM standards
  - Open-source public donations for “cross language connectivity fabric” are TLM-2 based and are already out there:
    - UVM-Connect - from Mentor Graphics
    - UVM-ML - from Accellera MLWG

- It makes sense to combine the strengths of 2 verification methodologies (SV-UVM, SystemC) into hybrid testbenches
  - Power of SystemVerilog UVM:
    - Constrained random traffic/sequence generation
    - Coverage, scoreboarding
  - Power of SystemC:
    - Good Linux host system interfacing capability which it gets for free simply by being written in C++
      - Direct access to host resources disks, networks, device drivers, X-windows displays, etc.
    - Stimulus via real host machine system interfaces and virtual platforms
The case for hybrid testbenches

- Types of hybrid testbenches:
  - Reuse of legacy IP models
    - Often RTL verification IP models come with C/C++ based APIs based on SystemVerilog P1800 DPI standard
    - Desirable to integrate this IP into SV-UVM testbench environments
  - Interchangeable testbench heads
    - A single source verification IP model can get reuse from multiple testbench modeling methodologies
    - Providing a TLM compliant socket API allows “interchangeable testbench heads”
  - Dual use UVM drivers
    - UVM drivers that provide interfaces to RTL models can be equipped with a 2nd TLM port (in addition to traditional sequencer port) to provide an extra inbound traffic channel from peer TLM compliant models – possibly cross language ones
  - Virtual platform (VP) hybrid testbenches
    - Starting to see a number of QEMU derivatives out there which pair fast host based virtual platforms with RTL designs under test (DUTs)
Standards for hybrid testbenches: Achieving Interop with Standard Interfaces

- To interoperate two components must agree on
  - information to exchange (i.e., the data type)
  - means of exchanging that information (i.e., the interface)

- To be reusable, easy to use, components must be
  - Independent of their context, not expose implementation

- Analogy: Media Server and TV
  - They don’t know about each other; independent design
  - They agree on common data (video) & interface (HDMI)
  - *Both can be connected to many other devices*
What is UVM-Connect?

Enabling Technology

- Enables TLM communication between SV+SC
  - Using native TLM calls
  - Implemented over SystemVerilog P1800 DPI standard

- Leverages UVM TLM 2.0 implementation
  - Also supports TLM 1.0 and Analysis ports

- Provided as separate SV and SC packages

- Cross language binding “rendezvous” via a uvmc::connect() method
  - Uses a string to identify corresponding sockets
What is UVM-Connect?
Trans-language TLM connections

- Connect SysC & SV-UVM models using standard TLM1, TLM2 interface ports
  - TLM GP handled automatically
- Access and control UVM from SystemC via command API
  - Messaging, configuration, and factory methods supported
  - Synchronize SystemC to UVM phases
- Standards based, available today
  - Vendor-independent
  - Fully open-sourced, *Apache licensed* package just like the UVM 1.1b base package is

Can obtain from Mentor’s Verification Academy here:

UVM-Connect primer
TLM Connection – SV using UVMC

Add a scoreboard, add a connection.

```cpp
#include "uvmc.h"
using namespace uvmc;
#include "consumer.h"

int sc_main(int argc,char* argv[]) {
    consumer cons("consumer");
    uvmc_connect(cons.in,"foo");
    uvmc_connect(cons.ap,"bar");
    sc_start();
    return 0;
}
```

```cpp
import uvm_pkg::*;
import uvmc_pkg::*;
`include "producer.sv"
`include "scoreboard.sv"

module sv_main;
    producer prod = new("prod");
    scoreboard sb = new("sb");
    initial begin
        prod.ap.connect(sb.expect_in);
        uvmc_tlm #(uvm_tlm_gp)::
            connect(prod.out, "foo");
        uvmc_tlm1 #(uvm_tlm_gp)::
            connect(sb.actual_in, "bar");
        run_test();
    end
endmodule
```
#include <systemc.h>
using namespace sc_core;
#include "producer.h"
#include "uvmc.h"
using namespace uvmc;

struct prod_alt : public producer {
    prod_alt(sc_module_name nm) :
        producer(nm) {
        SC_THREAD(objector);
    }
    SC_HAS_PROCESS(prod_uvm)
    void objector() {
        uvmc_raise_objection("run");
        wait(done);
        uvmc_drop_objection("run");
    }
};

int sc_main(int argc,char* argv[]) {
    prod_alt prod("producer");
    uvmc_connect(prod.in,"42");
    sc_start(-1);
    return 0;
}

module sv_main;
    consumer cons = new("cons");
    initial begin
        uvmc_tlm #()::connect(cons.in,"42");
        uvmc_init();
        run_test();
        end
endmodule
Reuse of legacy IP models

SV-UVM Test

TestbenchEnv

Scoreboard

UVM-Connect’ions

RandWriteReadSeq

APB Bus

Top

DPI based API

APB Master TlmDriver

APB Master DpiDriver

APB Slave DpiDriver

APB Monitor TlmDriver

APB Monitor Xactor

APB Slave Xactor

APB Master Xactor

= TLM-2.0 initiator -> target socket

= TLM analysis port -> subscribers

= Legacy DPI driver

March 3, 2014
Reuse of legacy IP models

TLM-2 initiator <-> target sockets

---

// UVM-Connect’ed SV-UVM uvm_driver “initiator”

class ApbBusMasterDriver
extends uvm_driver

    `uvm_component_utils(ApbBusMasterDriver)

protected string peerId; // UVM-Connect ID

uvm_tlm_b_initiator_socket #(uvm_tlm_generic_payload) initiatorSocket;

function new( string name, uvm_component parent );

function void connect_phase( uvm_phase phase );

task run_phase( uvm_phase phase );

endclass // }

// UVM-Connect’ed SystemC “target”

class Testbench : public sc_module {
    ApbMasterTlmDriver *apbMaster;
    Testbench( sc_module_name name );
};

class ApbMasterTlmDriver : public sc_module,
public virtual tlm::tlm_fw_transport_if<>

{ tlm::tlm_target_socket<32> socket;

    ApbMasterTlmDriver( 
        sc_module_name name, const char *transactorPath );

    void b_transport( 
        tlm::tlm_generic_payload &trans, 
        sc_time &delay );
}
Reuse of legacy IP models

TLM-2 initiator <-> target sockets

SV-UVM Testbench

TestbenchEnv

apbUve Agent

ApbTalker

Monitor

Driver

Monitor

Driver

APB Master

TlmDriver

APB Master

DpiDriver

SystemC Testbench

// UVM-Connect'ed SV-UVM uvm_driver "initiator"
class ApbBusMasterDriver
extends uvm_driver #(uvm_tlm_generic_payload); // {
  `uvm_component_utils(ApbBusMasterDriver)
protected string peerId; // UVM-Connect ID
uvm_tlm_b_initiator_socket#( uvm_tlm_generic_payload
initiatorSocket;
function new(string name, uvm_component parent);
  super.new(name, parent);
  initiatorSocket = new( "initiatorSocket", this );
endfunction
function void connect_phase( uvm_phase phase );
  super.connect_phase( phase );
  // Retrieve HDL path from UVM config DB
  assert( get_config_string( "peerId", peerId ) )
nvmc_tlm #( uvm_tlm_generic_payload,
    uvm_tlm_phase_e );
  ::connect( initiatorSocket, peerId );
endfunction
endclass // }
Reuse of legacy IP models
TLM-2 analysis broadcasters -> subscribers

// UVM-Connect’ed SV-UVM “subscriber”

class ApbMonitor
extends uvm_subscriber
  #(uvm_tlm_generic_payload); // {
  `uvm_component_utils(ApbMonitor)
  protected string peerId; // UVM-Connect ID
  uvm_analysis_port #( uvm_tlm_generic_payload )
    analysisPort;

  function new(string name, uvm_component parent);
  function void connect_phase( uvm_phase phase );
  function void write( uvm_tlm_generic_payload t );
endclass // }

// UVM-Connect’ed SystemC “broadcaster”

class Testbench : public sc_module {
  ApbMonitorTlmDriver *apbMonitor;
  Testbench( sc_module_name name );
};

class ApbMonitorTlmDriver : public sc_module,
  public tlm::tlm_analysis_port<tlm::tlm_generic_payload>
{
  tlm::tlm_generic_payload dMonitorRecordTrans;
  ApbMonitorTlmDriver(
    sc_module_name name, const char *transactorPath );
  void write( const svBitVecVal *monitorRecord );
}

// Import "DPI-C" function
extern "C" void ApbMonitorWrite(
    const svBitVecVal *monitorRecord ){
  ApbMonitorTlmDriver *me = (ApbMonitorTlmDriver *)
    svGetUserData( svGetScope(),
    (void *)&ApbMonitorWrite );
  me->write( monitorRecord );
}

= Legacy DPI driver

= TLM-2.0 initiator -> target socket
= TLM analysis port -> subscribers

March 3, 2014
Reuse of legacy IP models

TLM-2 analysis broadcasters -> subscribers

SV-UVM Testbench

// UVM-Connect'ed SV-UVM "subscriber"
class ApbBusMonitor
extends uvm_subscriber
    #(uvm_tlm_generic_payload);

protected string peerId; // UVM-Connect ID

uvm_analysis_port #( uvm_tlm_generic_payload
    analysisPort; // { 

function new(string name, uvm_component parent);
    super.new(name, parent);
    analysisPort = new("analysisPort", this);
endfunction

function void connect_phase( uvm_phase phase );
    super.connect_phase( phase );
    // Retrieve peer ID from UVM config DB. If cannot
    // be found, assume this TB is not interested in
    // hooking up the monitor. Else, UVM-Connect it.
    // Note reference to 'analysis_export' data member
    // of base class uvm_subscriber.
    if( get_config_string( "peerId", peerId ) )
        uvmc_tlm1 #(uvm_tlm_generic_payload)
            ::connect( analysis_export, peerId );
endfunction

function void write( uvm_tlm_generic_payload t );
    analysisPort.write( t );
endfunction

endclass // }

// UVM-Connect'ed SystemC "broadcaster"
class Testbench : public sc_module {
    ApbMonitorTlmDriver *apbMonitor;

    Testbench( sc_module_name name ) : sc_module(name)

    apbMonitor = new ApbMonitorTlmDriver( 
        "apbMonitor", "Top.apbMonitor" );

    uvmc_connect( *apbMonitor, "monitor" );
}

class ApbMonitorTlmDriver
    : public sc_module,
    public tlm::tlm_analysis_port<tlm::tlm_generic_payload { 

    tlm::tlm_generic_payload dMonitorRecordTrans;

    ApbMonitorTlmDriver( 
        sc_module_name name, const char *transactorPath ) 
    : sc_module( name ),
        tlm::tlm_analysis_port<
            tlm::tlm_generic_payload>(name),
        dTransactorPath( transactorPath )
    { }

    void write( const svBitVecVal *monitorRecord ){
        dMonitorRecordTrans.set_data_ptr( 
            reinterpret_cast<const unsigned char *>(
                monitorRecord) );
        tlm::tlm_analysis_port<
            tlm::tlm_generic_payload>::write( 
                dMonitorRecordTrans );
    }

};
Interchangeable testbench heads

UART transactor example

UartTargetTransactor has 4 interchangeable initiator clients:
- SV-UVM TLM 2 test sequence client
- SV-UVM TLM 2 file i/o client
- SystemC TLM-2.0 xterm client
- SystemC TLM-2.0 file i/o client
This configuration demonstrates a dual use SV-UVM driver equipped with both a sequencer port and a TLM-2 port.

In this case the TLM-2 port in both the SV-UVM driver and the slave memory (back-door port) are unused.
This configuration demonstrates a dual use SV-UVM driver equipped with both a sequencer port and a TLM-2 port.

- In this case the sequencer port is unused.
Hybrid testbenches: Virtual platform example

Guest OS (Android/Linux)
Virtual Machine (FastModel FVP Simulator)
HVL Testbench (SystemC and/or SystemVerilog)

TestbenchEnv
AxiTalker
AxiCover

AXI Agent
Config
 disc
S
sequences
D
Monitor
Driver

AXI "Agent"
AXI Master
Driver+Xactor

AXI Bridge
Tlm2Rtl

AXI Slave
Driver+Xactor

SystemC

TBX
Co-model
Channel (DPI)

Veloce

soc_hdl_top

soc_dut_wrapper

axi_dut_wrapper

Clock & Reset
Generation
Graphics
Subsystem
Interrupt
Monitor

ARM FastModel
Virtual Platform

RTSM CortexA8 Versatile EB System

Top Level:
Builds an Emulation Baseboard RTSM,
containing an ARM CortexA8 Core Tile
Hybrid testbenches: Ethernet packet router example

This configuration uses SystemC DPI based legacy models, ...

Each “dotted line” TLM crossing is a UVM-Connect’ion!
Summary

- There is a good case to made for hybrid SV-UVM SystemC testbenches:
  - Supporting legacy IP models
  - Interchangeable testbenches can be coupled with reusable verification IP models
  - UVM drivers can be designed for "dual use" to accommodate alternative TLM channels for input
  - Hybrid testbenches allow taking combining of strengths of SystemC …
    - Virtual platforms, "real system" interfaces
    - Constrained random sequences, coverage, scoreboarding

- The TLM 1 and 2 standards are well supported by both SV-UVM and SystemC methodologies
  - There are now open-source "cross-language" TLM connectivity packages readily available - and being considered for standardization
Thank you