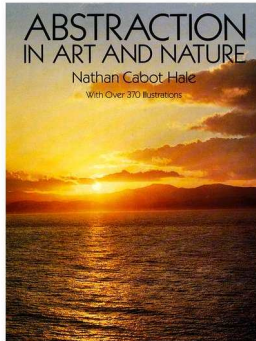


Introduction to SystemC, part 2

Deian Tabakov

Rice University
Houston, TX

September 21, 2007



From last time...

Fundamental concepts

- Modules
- Channels
- Signals
- Sensitivity

Other important concepts

- Hierarchical models
- Simulation kernel
- Delta cycles

Today:

- Primitive and hierarchical channels
- Modeling at various levels of abstraction: Parameterization and MoC

From last time...

Fundamental concepts

- Modules
- Channels
- Signals
- Sensitivity

Other important concepts

- Hierarchical models
- Simulation kernel
- Delta cycles

Today:

- Primitive and hierarchical channels
- Modeling at various levels of abstraction: Parameterization and MoC

From last time...

Fundamental concepts

- Modules
- Channels
- Signals
- Sensitivity

Other important concepts

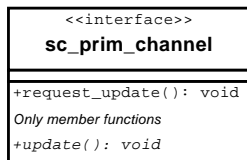
- Hierarchical models
- Simulation kernel
- Delta cycles

Today:

- Primitive and hierarchical channels
- Modeling at various levels of abstraction: Parameterization and MoC

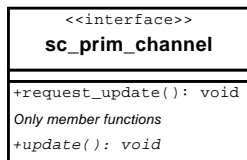
Primitive channels: simple communication

- Simulate concurrency: serialization and arbitration
- Delay any operation that may potentially change the state of the channel



Primitive channels: simple communication

- Simulate concurrency: serialization and arbitration
- Delay any operation that may potentially change the state of the channel



Examples

- `sc_signal<T> : sc_signal_inout_if<T>`
- `sc_fifo<T> : sc_fifo_in_if<T>, sc_fifo_out_if<T>`
- `sc_mutex : sc_mutex_if`

Hierarchical channels: complex communication structures

- May contain processes, ports, modules, other channels
- Use more memory than primitive channels
- “...model complex communication within a design in a much more elegant way than is possible in languages such as VHDL and Verilog”

Primitive vs. Hierarchical Channels

Hierarchical channels: complex communication structures

- May contain processes, ports, modules, other channels
- Use more memory than primitive channels
- “...model complex communication within a design in a much more elegant way than is possible in languages such as VHDL and Verilog”

Example: On-chip bus

- Arbiter
- Control programming unit
- Decoder unit

Hierarchical channels: complex communication structures

- May contain processes, ports, modules, other channels
- Use more memory than primitive channels
- “...model complex communication within a design in a much more elegant way than is possible in languages such as VHDL and Verilog”

Example: On-chip bus

- Arbiter
- Control programming unit
- Decoder unit

Another example: FIFO buffer with handshaking

- Refine `sc_fifo`
- Add functions for handshaking

Inside the `sc_signal<T>` primitive channel: the interfaces

Code (`sc_signal_in_if`)

```
template <class T> class sc_signal_in_if
    : virtual public sc_interface {
public:
    // get the value changed event
    virtual const sc_event& value_changed_event() const = 0;

    // read the current value
    virtual const T& read() const = 0;
};
```

Code (`sc_signal_inout_if`)

```
template <class T> class sc_signal_inout_if
    : public sc_signal_in_if<T> {
public:
    // write the new value
    virtual void write(const T&) = 0;
};
```

Code (sc_signal)

```
template <class T> class sc_signal
    : public sc_signal_inout_if<T>,
      public sc_prim_channel {
public:
    virtual const sc_event& value_changed_event() const {
        return m_value_changed_event;
    }

    virtual const T& read() const {
        return m_cur_val;
    }

    virtual void write(const T& value_) {
        m_new_val = value_;
        if (! (m_new_val == m_cur_val))
            request_update();
    }
    ..... //continued on next page
```

Code (sc_signal)

```
template <class T> class sc_signal
    : public sc_signal_inout_if<T>,
      public sc_prim_channel {

    .....// continued from previous page

protected:
    virtual void update() {
        if ( !(m_new_val == m_cur_val)) {
            m_cur_val = m_new_val;
            m_value_changed_event.notify(SC_ZERO_TIME);
        }
    }

    T        m_cur_vale;
    T        m_new_vale;
    sc_event m_value_changed_event;
};
```

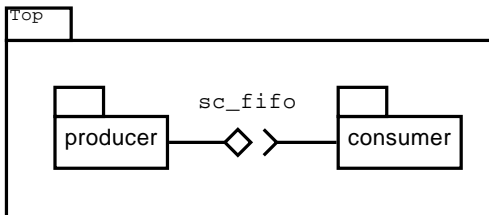
Another example: `sc_signal_resolved`

- One-big wide signal
- Multiple drivers; values in $\{0, 1, X, Z\}$

Implementation details (no code):

- Inherit from `sc_signal<sc_logic>`
- Overwrite `write()`:
 - Store the current driving value for each driving process
 - Call `request_update()`
- Overwrite `update()`:
 - Compute a new value deterministically
 - Notify the value-changed event

Other examples: `sc_fifo`, `sc_mutex...`



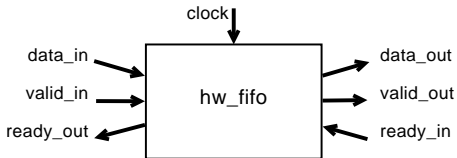
Producer:

- Writes characters to the channel
- One character at a time
- 50% probability
- Wait for 1 nanosecond between writes

Consumer:

- Reads characters to the channel
- One character at a time
- 50% probability
- Wait for 1 nanosecond between writes

Refining the design



Code (hw_fifo)

```
template <class T> class hw_fifo : public sc_module {
public:
    sc_in<bool>    clk;
    sc_in<T>      data_in;
    sc_in<bool>   valid_in;
    sc_out<bool>  ready_out;

    sc_out<T>     data_out;
    sc_out<bool>  valid_out;
    sc_in<bool>   ready_in;
    ....// more code
};
```

Code (hw_fifo)

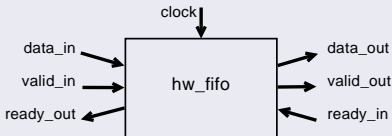
```
template <class T> class hw_fifo : public sc_module {
    ....// continued from previous page
    SC_HAS_PROCESS(hw_fifo);

    hw_fifo(sc_module_name name, unsigned size)
        : sc_module(name), _size(size) {
        assert(size > 0)
        _first = _items = 0;
        _data = new T[_size];

        SC_METHOD(main);
        sensitive << clk.pos();

        ready_out.initialize(true);
        valid_out.initialize(false)
    }

    ~hw_fifo() {delete[] _data;}
    ....// more code
};
```



Hierarchical Channels

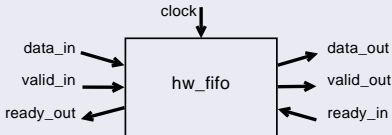
Code (hw_fifo)

```
template <class T> class hw_fifo : public sc_module {
    ...// continued from previous page
protected:
    void main() {
        if (valid_in.read() && (_items < _size)) {
            // new data item is ready to be stored
            _data[( _first + _items) \% _size] = data_in;
        }

        if (ready_in.read() && (_items > 0)) {
            _items--;
            _first = (_first + 1) \% _size;
        }

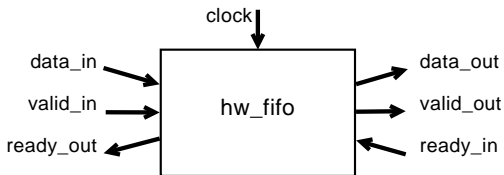
        // update all output signals
        ready_out.write(_items < _size);
        valid_out.write(_items > 0);
        data_out.write(_data[_first]);
    }

    unsigned _size, _first, _items;
    T* _data;
};
```

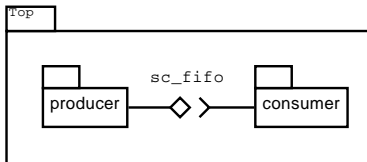


Hierarchical Channels

Can we plug this module...

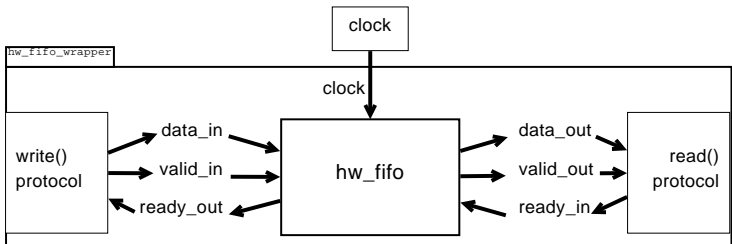


...in the place of `sc_fifo` in the Producer-Consumer model?

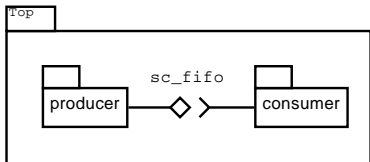


Hierarchical Channels

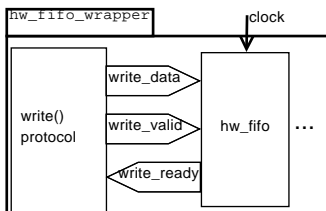
Can we plug this module...



...in the place of `sc_fifo` in the Producer-Consumer model?



Hierarchical Channels



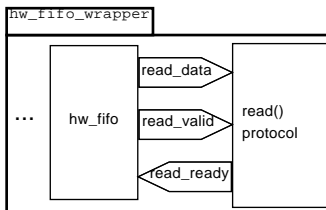
Code (hw_fifo_wrapper)

```
template <class T> class hw_fifo_wrapper
: public sc_module, public sc_fifo_in_if<T>, public sc_fifo_out_if<T> {

public:
    virtual void write(const T& data) {
        write_data = data;
        write_valid = true;

        do {
            wait(clk->posedge_event());
        } while (write_ready != true);
        write_valid = false;
    }
};
```

Hierarchical Channels



Code (hw_fifo_wrapper)

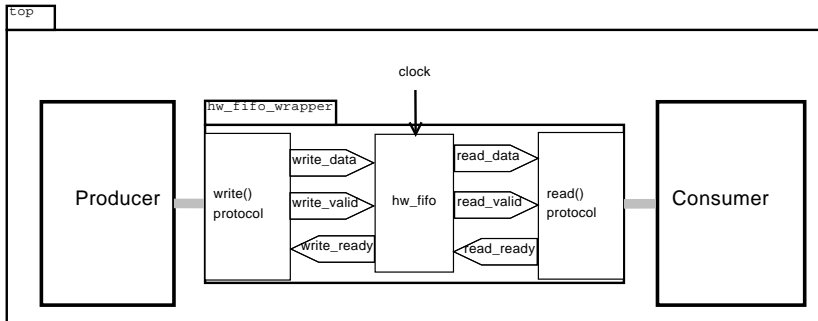
```
template <class T> class hw_fifo_wrapper
: public sc_module, public sc_fifo_in_if<T>, public sc_fifo_out_if<T> {

virtual T read() {
    read_ready = true;

    do {
        wait(clk->posedge_event());
    } while (read_valid != true);

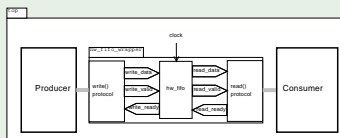
    read_ready = false;
    return read_data.read();
}
};
```

Putting it all together



Example (Putting it all together)

```
#include 'hw_fifo_wrapper.h'  
#include 'producer.h'  
#include 'consumer.h'  
  
int sc_main(int argc, char* argv[]) {  
    hw_fifo_wrapper HW_fifo('clocked_fifo');  
  
    Producer prod('Producer');  
    Consumer cons('Consumer');  
  
    // connect the modules via the channel  
    prod.out(HW_fifo);  
    prod.in(HW_fifo);  
  
    sc_start(); // run forever  
    return 0;  
}
```



Parameterized modules and channels

- Leverage powerful language constructs provided by C++
- Promote re-usability
- One-up VHDL and Verilog

Parameterized modules and channels

- Leverage powerful language constructs provided by C++
- Promote re-usability
- One-up VHDL and Verilog

Examples of what can be parameterized in SystemC

- Parameterized values within a design
 - Depth of FIFO channel (recall `hw_fifo_wrapper`)
 - Constant data values within read-only memory
- Parameterized types and attributes
 - Data type of a channel
 - Width of a fixed-point type
- Parameterized design structure
 - Parameterized width carry-lookahead adder
 - Conditionally instantiate parts of the adder based on the value of the width parameter

Resolving parameter values

- At compile time
 - Resolved when the C++ compiler is run
 - C++ template parameters
- At elaboration time
 - Resolved when the SystemC design hierarchy is constructed (after `sc_main()` and before `sc_start()`)
 - C++ constructor arguments for modules and channels
 - *Advantage*: Can be specified via command line arguments
- At simulation time
 - Resolved as the design is simulated
 - May change during simulation
 - SystemC's channels can carry updated parameters values where they are needed

Which one should you use?

- Elaborate at the earliest possible time
 - Design errors usually caught early
 - C++ compiler catches most type incompatibilities when using compile-time resolution
 - Compiler has a better shot at optimizing code
 - Synthesis and analysis tools may not fully support elaboration time and simulation time resolution

Which one should you use?

- Elaborate at the earliest possible time
 - Design errors usually caught early
 - C++ compiler catches most type incompatibilities when using compile-time resolution
 - Compiler has a better shot at optimizing code
 - Synthesis and analysis tools may not fully support elaboration time and simulation time resolution
- Exceptions to the above rule
 - Hard to protect IP (template code in the .h files)
 - C++ restrictions: only types or simple integer arguments in templates
 - The architecture of the model cannot be changed (as of SystemC 2.0.1)
 - The code may compile, but be hard to read
 - Confusing error messages

SystemC can emulate virtually any discrete time system

- RTL hardware modeling
- Static/Dynamic multirate dataflow
- Kahn process network
- Transaction-based

Multiple different MoCs in the same system

- Natural way of expressing complex systems
- MoC can “evolve” during refinement process
- Communication/synchronization between different MoCs via channels (adapters)

Register Transfer Level

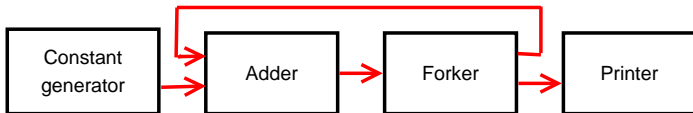
- Pin and cycle accurate
- Communication through signals
 - Sequential logic: sensitive to the clock
 - Combinational logic: sensitive to inputs

Comparison with HDL simulators

- HDL simulators: order of evaluation determined before start of simulation
- SystemC RTL simulation: dynamic order of evaluation

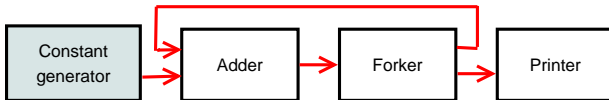
Kahn Process Networks (KPN)

- Processes execute concurrently
- Data exchanged through infinite-length FIFO channels
- Time delays can be added if necessary
- Scheduling order does not change observable behavior



Static Dataflow (SDF) Networks

- Special case of KPN
- Activation rules known at compile time
- FIFO length can be computed at compile time

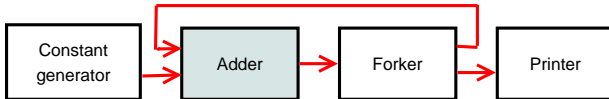


constgen.h

```
SC_MODULE(constgen) {
    sc_fifo_out<float> output;

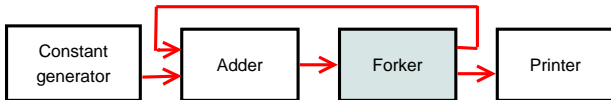
    // The constructor
    SC_CTOR(constgen) {
        SC_THREAD(generating());
    }

    void generating() {
        while (true) {
            output.write(0.7);
        }
    }
}
```

adder.h

```
SC_MODULE(adder) {  
  
    sc_fifo_in<float> input1, input2;  
    sc_fifo_out<float> output;  
    SC_CTOR(adder) {  
        SC_THREAD(adding());  
    }  
    void adding() {  
        while (true) {  
            output.write(input1.read() + input2.read());  
        }  
    }  
}
```



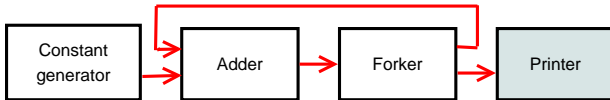
forker.h

```
SC_MODULE(forker) {
    sc_fifo_in<float> input;
    sc_fifo_out<float> output1, output2;

    SC_CTOR(forker) {
        SC_THREAD(forking());
    }

    void forking() {
        while (true) {
            float value = input.read();
            output1.write(value);
            output2.write(value);
        }
    }
}
```

Kahn Process Networks



printer.h

```
SC_MODULE(printer) {
    sc_fifo_in<float> input;

    SC_CTOR(printer) {
        SC_THREAD(printing());
    }

    void printing() {
        for (unsigned int i = 0; i < 100; i++) {
            float value = input.read();
            printf("%f\n", value);
        }
        return; // this indirectly stops the simulation
               // (no data will be flowing any more)
    }
}
```

Code (Putting it all together)

```
#include "constgen.h"  
#include "adder.h"  
#include "foker.h"  
#include "printer.h"
```



```
int sc_main(int argc, char* argv[]) {  
  
    // The FIFO channels  
    sc_fifo<float> gen_add, add_fork, fork_add, fork_print;  
  
    // The modules  
    constgen Gena("Generator");  
    adder Addy("Adder");  
    forker Forky("Forker");  
    printer Prn("Printer");  
  
    // connect the modules  
    Gena.output(gen_add);  
    Addy.input1(gen_add);  
    Addy.input2(fork_add);  
    Forky.output1(fork_add);  
    Forky.output2(fork_print);  
  
    Prn.input(fork_print);  
    sc_start(); // run forever  
    return 0;  
}
```

- Smaller and faster than RTL
- Model only necessary details
- Communication through interface function calls
 - Ignore implementation details: how data moves, protocol rules
 - Emphasize functionality: where data is transferred to and from
- Data exchanged through shared data variables (call by reference)

Definition (Transaction [NH07])

A *transaction* $t \in T$ is a pair (m_t, sig) , where $m_t \in M_T$ is the target module on which the transaction is being executed, and $sig \in N$ is the signature of the transaction.

Definition (Signature [NH07])

The *signature* of a transaction is a triple $sig = (c, d_{req}, d_{resp})$ where c denotes a computation request, d_{req} is the request part of the communication payload data, and d_{resp} is the response part of the communication payload data.

- Fast and accurate enough to validate software
- Easy to develop and use
- Accurately models both hardware and software components

very_simple_bus_if

```
// First define the interfaces
class very_simple_bus_if : virtual public sc_interface {

public:
    virtual void burst_read (char* data,
                            unsigned addr,
                            unsigned length) = 0;

    virtual void burst_write (char* data,
                              unsigned addr,
                              unsigned length) = 0;
};
```

very_simple_bus

```
class very_simple_bus {

public:
    // The constructor
    very_simple_bus( sc_module_name nm, unsigned mem_size,
                    sc_time cycle_time) :
        sc_channel(nm),
        _cycle_time(cycle_time) {

        // Model bus memory access using an embedded mem array
        _mem = new char [mem_size];

        // Initially the memory contains 0s
        memset(_mem, 0, mem_size);
    }
    ....// more code on the next page
}
```


burst_read

```
void burst_read(char* data, unsigned addr,
                unsigned length) {

    // Model bus contention using mutex w/out arbitration
    _bus_mutex.lock();

    // Block caller for length of burst transaction
    wait(length * _cycle_time);

    // Copy the data from memory to requester
    memcpy(data, _mem + addr, length);

    // Unlock mutex to allow others to access bus
    _bus_mutex.unlock();
}
```

burst_write

```
void burst_write(char* data, unsigned addr,
                unsigned length) {

    // Model bus contention using mutex w/out arbitration
    _bus_mutex.lock();

    // Block caller for length of burst transaction
    wait(length * _cycle_time);

    // Copy the data from requester to memory
    memcpy(_mem + addr, data, length);

    // Unlock mutex to allow others to access bus
    _bus_mutex.unlock();
}
```

Transaction Level Modeling: Summary

- Communication through function calls
- Fast, compact
- Integrate hardware and software models
- Early platform for software development, easy to distribute
- Early system exploration and verification
- Verification reuse
- Widely used, standard APIs and guidelines (TLM 2.0)

Summary

- Fundamental constructs of SystemC: time, modules, interfaces, ports, channels, methods and threads, events, sensitivity, instantiation, simulation semantics
- Primitive and hierarchical channels
- Parameterization of channels and modules, resolution issues and caveats
- Modeling abstraction levels: RTL, KPN, TLM

Connection to Moy's Thesis

- Main goal: formal verification of SystemC models
- Syntactic extraction (Pinapa) and semantic extraction (Bise) of relevant information from the SystemC code
- A series of transformations (Birth) and output of modelchecker-specific code

- EDA Electronic Design Automation
- SoC System-on-chip
- TLM Transaction-Level Model(ing)
- PV Programmer's View
- PVT Programmer's View with Time
- HDL Hardware Description Language
- RTL Register Transfer Level
- ASIC Application-Specific Integrated Circuit
 - IP Intellectual Property
- DSP Digital Signal Processing
- DUV Design Under Verification
- MoC Model of Computation
- KPN Kahn Process Network