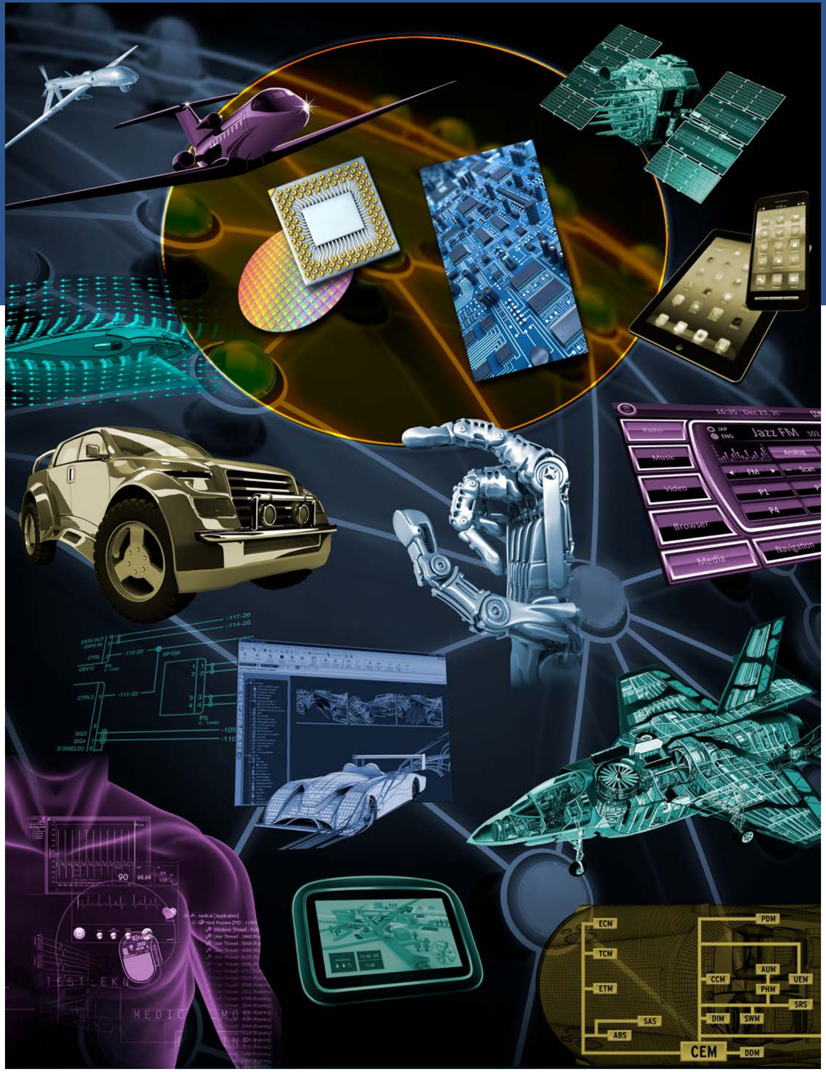
Presentation Copyright Permission

- A non-exclusive, irrevocable, royalty-free copyright permission is granted by **Mentor Graphics(a Siemens business)** to use this material in developing all future revisions and editions of the resulting draft and approved Accellera Systems Initiative **"SystemC"** standard, and in derivative works based on the standard.



Throughput accurate modeling and synthesis of abstract interfaces

Kunal Bindal

Calypto Systems Division

October 18, 2017

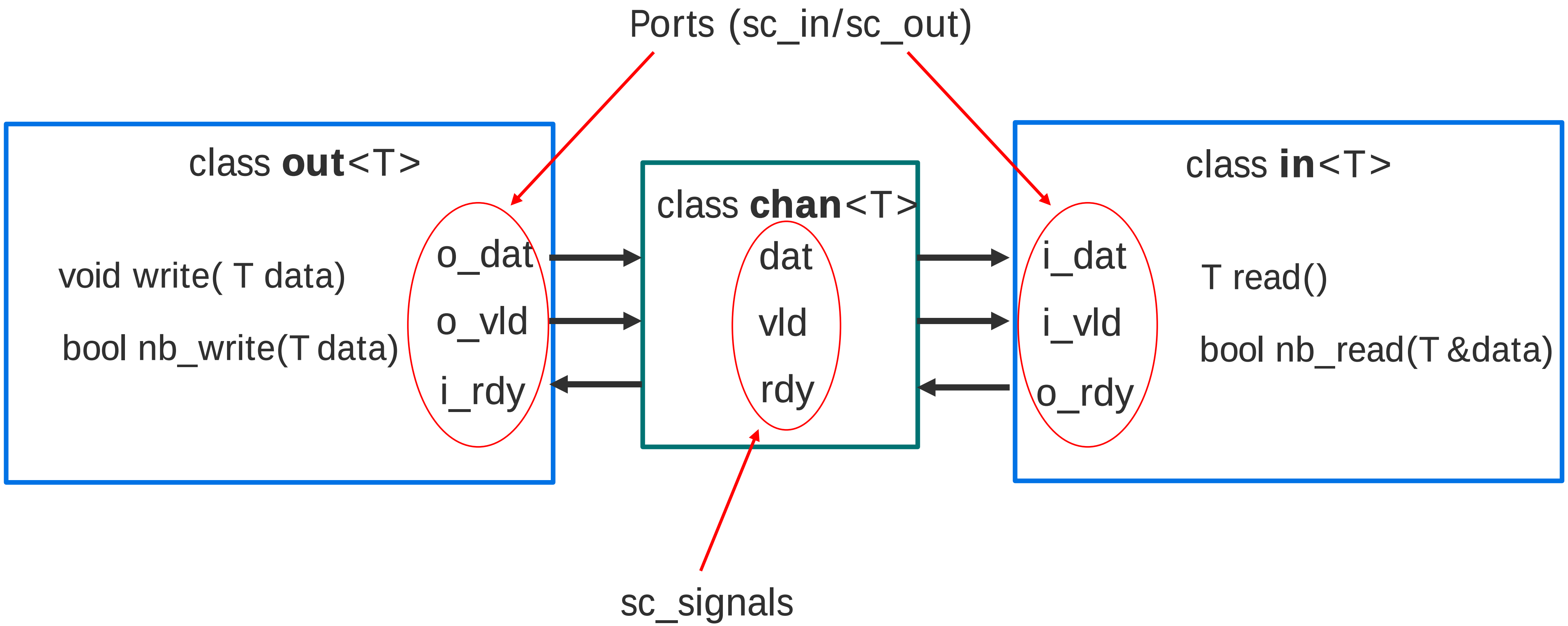
Mentor®
A Siemens Business

Motivation

- Modeling of pin and cycle accurate interfaces
  — Required to connect to RTL models
- Want abstract view (function call) view to use interfaces to model behavior:
  — r = in1.read() + in2.read()
- Modular IO: abstract view encapsulates pin and cycle accurate details of protocol
  — C++ class encapsulate ports and/or signals and provide abstracts functions such as read, nb_read, write, nb_write etc.

Example: p2p wire interface

- Protocol similar to AXI Streaming



P2P Example

```
SC_MODULE(DUT) {
  sc_in<bool> clk, rst;
  p2p<>::in<Data> in1, in2;
  p2p<>::out<Data> out1;
  SC_CTOR(DUT) : ... {  SC_CTHREAD(process, clk.pos(); reset_signal_is(rst,true); }
  void process() {
      in1.reset_read();  in2.reset_read(); out1.reset_write();
      while(1) {
          wait();
          out1.write( in1.read() + in2.read());
      }
   }
 };
```

Basics of the Protocol

- Transfer of data occurs when both vld and rdy are high in the same cycle
  — No COMBINATIONAL between vld and rdy
- To read:
  — o_rdy is set to high
  — Synchronously wait for i_vld to become high
  — get data from i_dat
  — Set o_rdy back to low

```
T read() {
   do {  o_rdy.write(true); wait(); } while (i_vld.read() != true);
   o_rdy.write(false);
   return i_dat.read();
}
```

Advantages

- Encapsulating protocols allow the behavior to remain more abstract:
  - Better separation of computation and communication
  - Opportunity to abstract behavior for faster simulation
- Synthesis has a more abstract view that allows it to understand Modular IO ports as being independent.
- Better for visualization and debug
- Library of p2p can provide different protocols that can be reused
  - fifos, events

Challenges

- There is no easy way to express concurrency between transactions:
  - r = in1.read() + in2.read()
  - Each read() encapsulates sequential behavior (pin wiggles) and consumes one or more cycles.
  - The two read() calls will be sequential, not concurrent
- Synthesis can treat transactions on different ports as concurrent
  - RTL implementation from HLS runs faster than SystemC model
- If SystemC model is to become "sign-off" point need to be able to have it run with the same concurrency of IO transactions:
  - Loosely speaking "throughput accurate"

Mentor
A Siemens Business

Throughput Accurate

- If SystemC model is able to run IO transactions as concurrent
- SystemC model can run as fast as fastest RTL implementation
- Add cycle latency to match the RTL interface behavior
  - Can exercise same cycle-accurate IO access patterns as RTL
- Three approaches explored for modeling throughput accurate IO
  - Forking to achieve concurrency
  - Concurrent Blocking for IO
  - Emulating concurrency in one thread

Approach 1: Forking to Achieving Concurrency

- Use SystemC functionality to spawn threads for transaction calls

rd = **sc_spawn**(&rd.data, sc_bind(&**in::read**, this), name, &spawn_opt);

- **sc_spawn** returns a handle of type **sc_process_handle**.
  - Construct and object **rd** of type **ac_fork_d** that stores
    - Process handle
    - has datamember **data** where result of **in::read()** is stored
  - Wait on completion of process using process handle
    - wait(process_handle.**terminated_event**())
  - Can define the operator()() to wait on the completion of process

Approach 1: Forking to Achieving Concurrency

- Use SystemC functionality to spawn threads for transaction calls

```
template <class T>
class in<T> {
  ...
  T read() {   ....;   return data; }    // synthesizable

  // ac_fork holds data of type T and process handle
  //     operator ()() waits for completion of spawned process
  typedef ac_fork_d<T> rd_t;
  rd_t rd;

  const rd_t &read_f() {        // forked version of read() that initiates read
#ifndef __SYNTHESIS__
    rd = sc_spawn(&rd.data, sc_bind(&in::read, this), name, &spawn_opt);
#else
    rd.data = read();
#endif
    return rd;
  }
};
```


Mentor
A Siemens Business

Approach 1: Forking to Achieving Concurrency

- Requires to split initiation and completion of transaction for transactions that return a result:
  - a.read_f(); b.read_f(); // initiate (spawn) reads for inputs a and b
  - Sum = a.rd() + b.rd(); // wait for completion

- Disadvantages
  - Split transactions are not desirable. Harder to manage cleanly
  - Impact on runtime
  - Harder to debug

Approach 2: Concurrent Block for IO

- Concurrent process
  - Does the protocol signaling
  - Transaction methods interact with it using signals
  - Contains buffer
    - Read is pre-fetched
    - Write is buffered
- Disadvantages
  - Since transaction methods are called from process sensitive to clock, input to output is registered.
    - Hard to debug behavior that interacts correctly with concurrent process
    - Runtime impact due to additional thread and signaling
  - Concurrent process needs to inherit clock and reset behavior from main process using the p2p IO
    - No easy syntax to hook them up correctly by default

Approach 3: Emulate Concurrency

- "Register" every p2p port from "reset" method of p2p port
  - Places it in a list of p2p port for that process (SC_THREAD or SC_CTHREAD)
- "Intercept" every wait() to call specialized wait_mio function that calls:
  - update_pre for every registered p2p port
  - ::wait()
  - update_post for every registered p2p port
- Since the update functions execute on every wait(), it models a concurrent block that actively interacts with the environment
  - Does all the pin and cycle accurate protocol signaling

Advantages

- Update functions are executed in the context of the process (SC_THREAD, SC_CTHREAD) that owns the p2p port
  - No context switch: minimizes runtime overhead
  - Clean interface between signals used for protocol and variables that are used to interact with transactions functions (e.g. read(), write())
  - Clock and reset are implicitly handled by parent process
- Since every p2p port is guaranteed to be updated for every WAIT, it is
  - Easy to add instrumentation to aid debugging, gathering of statistics etc.
  - Add forced stalls for:
    - coverage
    - alignment with RTL simulation

Example of p2p::out<T>

```
template <class T>
class out <T> : public MIO_Base {       // MIO_Base::wait() is special wait
    sc_out<T> o_dat; sc_out<bool> o_vld; sc_in<bool> i_rdy;        // Ports/signals
    T dat_in; bool dat_vld;  bool buffer_full;     // Variables
    // Update functions interact with the environment (port/signals)
    //     read/update variables
    void update_pre();
    void update_post();
    // Transaction functions
    //     Interact with update functions using variables
    bool ready() { return !buffer_full; }
    void write (T data);
    bool nb_write(T data);
}
```

Update functions for p2p::out<T>

```
void update_pre {
    if(!buffer_full & dat_vld) {
        buffer_full = true;
        o_dat.write(dat_in);
    }
    o_vld.write(buffer_full);
    dat_vld = false;
  }
}
```

```
void update_post {
    if(buffer_full) {
        if(i_rdy.read())
            buffer_full = false;
    }
}
```

Transaction functions for p2p::out<T>

```
bool ready() { return !buffer_full; }

void write (T data) {
    while(!ready())
        wait();          // MIO_Base::wait()
    dat_vld = true;
    data_in = data;
}
```

```
bool nb_write(T data) {
    if(ready()) {
        data_vld = true;
        dat_in = data;
        return true;
    }
    return false;
}
```

Update Function

- All signaling (sc_in, sc_out, sc_signal) is encapsulated in the update functions (pre and post).
- Update functions are called for every process wait
  - update_pre is immediately before the wait
  - update_post is called immediately after the wait
- Member functions for transactions (write, nb_write etc.) only use variables (rather than signals or ports)
- Member functions for blocking transactions calls special wait
  - Provided by MIO_Base

Mentor
A Siemens Business

Registering of p2p and Special wait()

- P2p port is automatically registered by process that calls “reset” method for it
  - in1.reset_read();
  - assert is triggered if port not registered
- Special wait cycles through the execution of all update_pre and update_post functions before and after an actual ::wait()
- Special wait is provided by
  - MIO_Base for wait() called from p2p class
  - sc_module2 base class (other mechanisms are possible)
    - Provides wait() for calls from thread functions (SC_THREAD and SC_CTHREAD)
- Run time checks (assert) to identify calls to non special wait (::wait)

Data Buffering

- Without true concurrency (spawning processes to initiate protocol) data buffering in the p2p class is required
- Without buffering for read:
  - Can take current valid data
  - Signal to environment data was taken
    - Environment will see it on next cycle
  - Can only perform 1 read for every 2 cycles
- FIFO buffering can be reduced due to buffering in the ports

Preliminary Results

- Used JPEG example with AXI bus interfacing to memory as a testcase
- The cycle throughput of the simulation improved as expected due to the concurrency between IO transactions
- The runtime was 1.27x compared to the original
  - Original runtime: 4.17s
  - Runtime with new p2p: 5.31s

Conclusions

- Implemented approach for throughput accuracy for point to point interfaces
  - P2P are pin accurate and implement protocol for port
  - P2P interfaces are modular and have transaction functions that can be called from behavior that is modeled at an abstract level
  - Concurrency of transactions is obtained by
    - Buffering reads and writes
    - Having a special wait that execute update function in every call
      - Responds to environment as a concurrent process would
  - P2P are “registered” during their reset calls
  - Update functions run as part of the process that owns the P2P port/channel

Mentor
A Siemens Business

Conclusions

- Runtime overhead is minor
- It provides a way to enable debug, gather activity information, force stalls
- It should enable closer match between SystemC model and Synthesized RTL using HLS
  - Goal is to enable sign-off point at SystemC
- Could be considered for standards
  - Modeling of Interfaces is an important TODO item in the Synthesis Subset Standardization

Mentor®

A Siemens Business

www.mentor.com