Hardware-Software Codesign

6. System Simulation

Lothar Thiele
System Design

specification

system synthesis

estimation

system simulation (this lecture)

(worst-case) perf. analysis (lectures 10-11)

SW-compilation

instruction set

HW-synthesis

machine code

net lists

intellectual prop. code

intellectual prop. block

System Design
Outline

- System classification
- Discrete event simulation
- Illustration: SystemC simulation
- Simulation at high abstraction levels
System and Model

- **A system** is a combination of components that act together to perform a function not possible with any of the individual parts.  
  \[\text{IEEE Standard Dictionary of Electrical and Electronic Terms}\]

- **A model** is a formal description of the system (or subsystem) which covers selected information.
System and Model - Example

Network Processor Example

- system
- model
The **state** of a system model at time $t_0$ contains all information necessary to determine the output at all $t \geq t_0$, from this information and from the input for all $t \geq t_0$.

The set $\mathbf{x}$ of possible states of a system is called its **state space**.

*Example*: state space modeling of continuous time driven systems.
Discrete State/Continuous State

- In **discrete state models**, the state space $\mathbf{x}$ is isomorphic to the set of integers, i.e., it is countable

- ...while other models are termed **continuous state models**

![Diagram of discrete state model](image1)

- ![Diagram of hybrid model](image2)
Time

- In a **continuous time model**, the set $T$ of admissible time values is isomorphic to the set of real numbers, i.e., $T \rightarrow \mathbb{R}$

- In a **discrete time model**, the set $T$ of admissible time values is isomorphic to the set of integer numbers, i.e., $T \rightarrow \mathbb{Z}$
Discrete/Continuous State/Time Systems

Some examples

- **Continuous state systems:** physical processes (usually), electrical networks, mechanical systems
- **Discrete state systems:** finite state machines, queuing systems, computer systems
- **Continuous time systems:** physical processes (usually), electrical circuits, asynchronous systems
- **Discrete time systems:** digital clocked system, equidistant sampling (z-transform), synchronous system models
Events

- An event \( e = (v, t) \) is a tuple of a value \( v \) and a tag \( t \) (tags are usually totally ordered)

  - If the tag denotes **time**, then the event is a **timed event**
  - If the tag only denotes (only) a **sequence number**, the event is an **untimed event**
Discrete Event Systems (DES)

- A DES is an **event-driven** system i.e., its state evolution depends entirely on occurrence of discrete events over time (or the tag system, in general), and not by the evolution of time.

- As in time-driven systems, a **DES model** can be defined in **continuous or discrete time**, depending on whether the admissible time instances are taken from a continuous or discrete set.

- The **state space of a DES model** can be either discrete or continuous, depending on $X$. 

Discrete Event Systems (DES) (contn.)

DES modeling objects:

- **Signals** or **streams** represent ordering/timing of events, i.e., based on ordered sequences.

- **Processes** are represented as functions acting on signals or streams.
Example: Queuing System

System

Model

State trajectory

$x(t)$: stored token

$t_1, t_2, t_3, t_4, t_5, t_6, t_7, t$
**Time-Driven vs. Event-Driven Simulation**

**Discrete-time, time-driven simulation**

- The simulated time is partitioned into (equidistant) time intervals
- The lengths of time intervals are determined by the simulated system (e.g., clock period), by the intended precision (discretization loss), or by the simulation effort
- A simulation step is performed even if nothing happens

\[ x(t) \]

\[ t \]
Time-Driven vs. Event-Driven Simulation

- Event-driven simulation
  - State changes only at events
  - Analysis and simulation possible in discrete or continuous time

![Diagram showing continuous and discrete time with events at specific points](image-url)
Outline

- System classification
- Discrete event simulation
- Illustration: SystemC simulation
- Simulation at high abstraction levels
Discrete-Event Modeling and Simulation

- Concurrent processes are modeled using modules
  - The behavior is described using logic and/or algebraic expressions
  - The state is described using persistent variables inside these modules
  - The communication between modules is done through ports, via signals
  - The synchronization is done through abstract events

- Modules can be hierarchical
- The system behavior is governed by events → event-driven simulation
Components of Discrete-Event Simulation

- Simulation clock
  - The simulation clock represents the current value of the simulation time
  - In discrete-event simulations, the clock advances to the next event start-time during simulation

- Event list
  - Events are processed in order, by the simulation engine
  - The event list is typically organized as a priority queue
  - Event lists may include simulation times when events will happen (in this case, lists are sorted by event times)

- System modules
  - System modules model subsystems of the simulated system
  - System modules are called by the simulation engine if an event relevant to the respective subsystem is scheduled
  - System modules process events, manipulate the event queue (add or remove events), and contain and manipulate the system state
Discrete-Event Simulation Engine

**Initialization routine**
- Initialize the simulation model: set initial states of subsystem modules, fill the event queue with initial events

**Timing routine**
- Determine the next event from the event queue
- Advance the simulation clock $clk$ to the time when the event is to occur

**Event routine**
- Update the system state when a particular type of event occurs
In a simulation cycle

- The events with the next time in the event queue are processed
- All modules sensitive to these events are executed \(\rightarrow\) this may “produce” new events

Problem:

- Within the same simulation cycle (same simulation clock), “cause” and “effect” events may share the same time of occurrence!
Discrete-Event Simulation

Solution:
- The simulator uses a zero duration virtual time interval, called *delta-cycle* ($\delta$)
- The role of a *delta-cycle* is to order “simultaneous” events within a simulation cycle, i.e., identifying which event caused another
  → in this way, “causes” and “effects” are separated by *delta-cycles*
- Simulation cycles may be composed of several delta-cycles ($\delta$)
Outline

- System classification
- Discrete event simulation
- Illustration: SystemC simulation
- Simulation at high abstraction levels
SystemC in a Nutshell

- System-level modeling language
  - Several levels of abstraction: from purely functional to cycle-accurate/pin-accurate
  - Special attention to systems with embedded software

- Library of C++ templates and classes for modeling concurrent systems
  - Hardware-oriented data types
  - Communication mechanisms
  - Concurrency modeling

- An event-driven simulation kernel for executing models

- … and available for free (Windows & Linux)
SystemC Principle

User module #1

User module #2

......

User module #N

Event & signal I/F

C++ class library

Events

Simulation kernel
(Event scheduler)

SystemC

Executable specification
**SystemC Language Architecture**

- **Methodology**
  - **specific channels**
  - master/slave lib, etc.

- **Standard channels**
  - for various MoC’s
  - Kahn process networks, static dataflow, etc.

- **Elementary channels**
  - signal, timer, mutex, semaphore, FIFO, etc.

- **Core language**
  - modules
  - ports
  - processes
  - interfaces
  - channels
  - events

- **Data types**
  - logic type (0’XZ)
  - logic vectors
  - bits and bit vectors
  - fixed point numbers
  - C++ built-in types
    - (int, char, double, etc.)
  - C++ user-defined types

- **Data types**
  - (separated from core lang.)

**On top of core language:**

- **Communication mechanisms** (e.g. signals, FIFOs)
- **Models of computation** (MoCs)
- **SystemC builds on C++**
- **Upper layers built on top of lower layers**

*Note: if desired, the lower layers within the diagram can be used without the upper layers*
Example NAND

```c++
#include "systemc.h"

SC_MODULE(nand) {
    // declare a NAND sc_module
    sc_in<bool> A, B;  // input signal ports
    sc_out<bool> F;    // output signal ports

    void do_it() {  // a C++ function
        F.write(!((A.read()) && (B.read())));
    }

    SCCTOR(nand) {  // constructor for the module
        SC_METHOD(do_it);  // register do_it() w/ kernel
        sensitive << A << B;  // sensitivity list
    }
};
```
Example EXOR

```
#include "nand.h"

SC_MODULE(exor) {
    sc_in<bool> A, B;
    sc_out<bool> F;
    nand n1, n2, n3, n4;
    sc_signal<bool> S1, S2, S3;

    SC_CTOR(exor) : n1("N1"), n2("N2"), n3("N3"), n4("N4") {
        n1.A(A);
        n1.B(B);
        n1.F(S1);

        n2 << A << S1 << S2;

        n3(S1);
        n3(B);
        n3(S3);

        n4 << S2 << S3 << F;
    }
};
```

alternatives to define connections between modules
Processes

Processes are the basic units of functionality

**SC_THREADS**
- Typically called once, run forever in a `while(true)` loop
- Can be suspended by calling the `wait()` function
- Keep the state of execution implicitly

**SC_METHODs**
- Execute repeatedly from the beginning to end
- Simulate faster
- Do not keep the state of execution implicitly

Processes must be contained in a module
- But not every member function is a process
**Modules**

- **Modules**: building blocks of SystemC models
  - Hierarchy
  - Abstraction
  - IP reuse

---

SC_MODULE

![Diagram showing SystemC module structures](image)
Module Template

```c
SC_MODULE(Module_name) {
    // Declare ports, internal data, etc.
    // Declare and/or define module functions

    SC_CTOR(Module_name) {
        // Body of the constructor

        // Process declarations and sensitivities
        SC_METHOD(function1);
        sensitive << input1 << input2;

        SC_THREAD(function2);
        sensitive << input1 << clk;
    }
};
```
Inter-Process Communication

- Processes can communicate directly through *signals*

![Diagram of inter-process communication with input ports, I/O ports, sensitivity, internal signals, and output ports between Process 1 and Process 2.](image-url)
Advanced Communication

- **Event**
  - Flexible, low-level synchronization primitive

- **Channel**
  - Container for *communication* and *synchronization*
  - e.g. can have state/private data, transport data, transport events
  - Channels implement one or more *interfaces*

- **Interface**
  - Set of access methods to the channel
  - Interface methods need to be implemented

*Other communication & synchronization models can be built based on the above primitives*
Channels and Interfaces

Module1

Interfaces

Channel

Module2

Events

Ports to Interfaces
Ex.1: Simple Producer-Consumer Application

‘Producer’ communicates with ‘consumer’ via a FIFO channel
Ex.1: Simple FIFO – Interface

class write_if : public sc_interface
{
  public:
    virtual void write(char) = 0;
    virtual void reset() = 0;
};

class read_if : public sc_interface
{
  public:
    virtual void read(char&) = 0;
    virtual int num_available() = 0;
};
Ex.1: Simple FIFO – Implementation

class fifo: public sc_channel,
    public write_if,
    public read_if
{
    private:
        enum e {max_elements=10};
        char data[max_elements];
        int num_elements, first;
        sc_event write_event,
        read_event;
        bool fifo_empty() {...};
        bool fifo_full() {...};

    public:
        fifo() : num_elements(0),
        first(0);

    void write(char c) {
        if (fifo_full())
            wait(read_event);
        data[ <you calculate> ] = c;
        ++num_elements;
        write_event.notify();
    }

    void read(char &c) {
        if (fifo_empty())
            wait(write_event);
        c = data[first];
        --num_elements;
        first = ...;
        read_event.notify();
    }

Ex.1: Simple FIFO – Implementation

class fifo: public sc_channel,
        public write_if,
        public read_if
{
    private:
        enum e {max_elements=10};
        char data[max_elements];
        int num_elements, first;
        sc_event write_event,
        read_event;
        bool fifo_empty() { }
        void reset() {
            num_elements = first = 0;
        }
        int num_available() {
            return num_elements;
        }
    } // end of class declarations

    void write(char c) {
        if (fifo_full())
            wait(read_event);
        data[<you calculate>] = c;
        ++num_elements;
        write_event.notify();
    }

    void read(char &c) {
        if (fifo_empty())
            wait(write_event);
        c = data[first];
        --num_elements;
        first = ...;
        read_event.notify();
    }
}
Ex. 1: Simple Producer-Consumer

```
SC_MODULE(producer) {
    public:
        sc_port<write_if> out;

    SC_CTOR(producer) {
        SC_THREAD(main);
    }

    void main() {
        char c;
        while (true) {
            out.write(c);
            if(...) 
                out.reset();
        }
    }
};

SC_MODULE(consumer) {
    public:
        sc_port<read_if> in;

    SC_CTOR(consumer) {
        SC_THREAD(main);
    }

    void main() {
        char c;
        while (true) {
            in.read(c);
            cout<<
            in.num_available();
        }
    }
};
```
Ex. 1: Simple Producer-Consumer

SC_MODULE(producer) {
    public:
        sc_port<write_if> out;
    
    SC_CTOR(producer) {
        SC_MODULE(top) {
            public:
                fifo afifo;
                producer *ppproducer;
                consumer *pconsumer;
            
            SC_CTOR(top) {
                pppproducer=new producer(“Producer”);
                pppproducer->out(afifo);
                pconsumer=new consumer(“Consumer”);
                pconsumer->in(afifo);
            }
        }
    
    void main() {
        char c;
        while (1) {
            c = (char) getchar();
            out.put(c);
            if (c=='$
                break;
        }
    }
};

SC_MODULE(consumer) {
    public:
        sc_port<read_if> in;
    
    SC_CTOR(consumer) {
        SC_MODULE(top) {
            public:
                fifo afifo;
                producer *ppproducer;
                consumer *pconsumer;
            
            SC_CTOR(top) {
                pppproducer=new producer(“Producer”);
                pppproducer->out(afifo);
                pconsumer=new consumer(“Consumer”);
                pconsumer->in(afifo);
            }
        }
    
    void main() {
        char c;
        while (1) {
            c = (char) getchar();
            in.get(c);
            if (c=='$
                break;
        }
    }
};
Ex. 2: Kahn Process Network - Generator

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);
        }
    }
}
Ex.2: Kahn Process Network - Adder

```c
adder.h

SC_MODULE(add) {

    sc_fifo_in<float> input1, input2;
    sc_fifo_out<float> output;
    SC_CTOR(add) {
        SC_THREAD(adding);
    }
    void adding() {
        while (true) {
            output.write(input1.read() + input2.read());
        }
    }
}
```
Ex.2: Kahn Process Network - Forker

KPN will deadlock unless an initial token is put in the loop:

```cpp
output1.write(0.0);
```

```cpp
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);
            output1.write(value);
        }
    }
};
```
Ex.2: Kahn Process Network - Printer

**printer.h**

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

    SCCTOR(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)
    }
}
```
Ex.2: Kahn Process Network – Top

Code (Putting it all together)

```c
#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");

    Prn.input(fork_print);
    sc_start(); // run forever
    return 0;
}
```
SystemC Principle

Event & signal I/F

User module #1
User module #2
......
User module #N

C++ class library

Events

Simulation kernel (Event scheduler)

SystemC

Executable specification
SystemC Scheduler

- initialize
- execute all processes until blocking point
- update signals
- compute the set of "ready" processes
  - compute the number of "ready" processes
    - clock cycle
      - == 0
        - advance simulation time
      - > 0
        - delta cycle
          - execute all processes until blocking point
          - update signals
Wait and Notify

- **wait**: halt process execution until an event is raised
  - `wait()` with arguments ⇒ dynamic sensitivity
    - `wait(sc_event)`
    - `wait(time)`
    - `wait(time_out, sc_event)`

- **notify**: raise an event
  - `notify()` with arguments ⇒ delayed notification
    - `my_event.notify();` //notify immediately
    - `my_event.notify(SC_ZERO_TIME);` //notify next delta cycle
    - `my_event.notify(time);` //notify after `time`
Outline

- System classification
- Discrete event simulation
- Illustration: SystemC simulation
- Simulation at high abstraction levels
Multiple Levels of Abstraction

- **(Untimed) functional level**
  - *Use*: model (un-)timed functionality
  - *Communication*: shared variables, messages
  - *Typical languages*: C/C++, Matlab

- **Transaction level**
  - *Use*: MPSoC architecture analysis, early SW development, timing estimation
  - *Communication*: method calls to channels
  - *Typical languages*: SystemC

- **Register transfer level /pin level**
  - *Use*: HW design and verification
  - *Communication*: wires and registers
  - *Typical languages*: Verilog, VHDL