EECS 222A: System-on-Chip Description and Modeling Lecture 4

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Lecture 4: Overview

- Homework Assignment 2
  - Discussion, Q&A
- Execution and Simulation Semantics
  - System-level Language Semantics
  - Motivating Examples
  - Simulation Semantics
  - Formal Execution Semantics

Homework Assignment 2

- Administration
  - Server
    - `epsilon.eecs.uci.edu`
    - Intel Pentium CPU, 3.0 GHz, 1GB RAM
    - RedHat Linux (Fedora Core 4)
    - Access via secure shell protocol (`ssh`)
  - Accounts
    - User ID same as your UCI net ID
    - Password as discussed in class
  - SpecC Software (© by CECS, UCI)
    - System-on-Chip Environment
      - `/opt/sce-20041007/bin/setup.csh`
      - `/opt/sce-20060301/bin/setup.csh`

Homework Assignment 2

- Task
  - Become familiar with
    - the System-on-Chip Environment (SCE)
  - Follow the initial steps in the SCE Tutorial
    - `/opt/sce-20041007/doc/SCE_Tutorial/sce-tutorial.pdf`
- Deliverables
  - none (but be prepared for the next assignment)
- Due
  - next week (Week 4)

System-level Language Semantics

- Concepts found in Embedded Systems
  - Behavioral and structural hierarchy
  - Concurrency
  - Synchronization and communication
  - Exception handling
  - Timing
  - State transitions
- System-level language must support these concepts
- Language semantics needed to define the *meaning*
  - Semantics of execution (modeling, simulation, synthesis)
  - Deterministic vs. non-deterministic behavior
  - Preemptive vs. non-preemptive concurrency
  - Atomic operations

System-level Language Semantics

- Language semantics are needed for
  - System designer (understanding)
  - Tools
    - Validation (compilation, simulation)
    - Formal verification (equivalence, property checking)
    - Synthesis
  - Documentation and standardization
- Objective:
  - Clearly define the execution semantics of the language
- Requirements and goals:
  - completeness
  - precision (no ambiguities)
  - abstraction (no implementation details)
  - formality (enable formal reasoning)
  - simplicity (easy understanding)

System-level Language Semantics

- Example: SpecC language
  - Documentation
    - Language Reference Manual (LRM)

      ⇒ set of rules written in English (not formal)
    - Abstract simulation algorithm

      ⇒ set of valid implementations (not general)
  - Reference implementation
    - SpecC Reference Compiler and Simulator

      ⇒ one instance of a valid implementation (not general)
    - Compliance test bench

      ⇒ set of specific test cases (incomplete)
  - Formal execution semantics
    - Time-interval formalism

      ⇒ rule-based formalism (incomplete)
    - Abstract State Machines

      ⇒ fully formal approach (not easy to understand)

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 7

Execution and Simulation Semantics

- Motivating Example 1
  - Given:

```
behavior B1(int x)
{
  void main(void)
  {
    x = 5;
  }
};
```

```
behavior B2(int x)
{
  void main(void)
  {
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  B1 b1(x);
  B2 b2(x);

  void main(void)
  {
    b1; b2;
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: x = 6

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 8

Execution and Simulation Semantics

- Motivating Example 2
  - Given:

```
behavior B1(int x)
{
  void main(void)
  {
    x = 5;
  }
};
```

```
behavior B2(int x)
{
  void main(void)
  {
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  B1 b1(x);
  B2 b2(x);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: The program is non-deterministic!
    (x may be 5, or 6, or any other value!)

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 9

Execution and Simulation Semantics

- Motivating Example 3
  - Given:

```
behavior B1(int x)
{
  void main(void)
  {
    waitfor 10;
    x = 5;
  }
};
```

```
behavior B2(int x)
{
  void main(void)
  {
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  B1 b1(x);
  B2 b2(x);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: x = 5

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 10

Execution and Simulation Semantics

- Motivating Example 4
  - Given:

```
behavior B1(int x)
{
  void main(void)
  {
    waitfor 10;
    x = 5;
  }
};
```

```
behavior B2(int x)
{
  void main(void)
  {
    waitfor 10;
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  B1 b1(x);
  B2 b2(x);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: The program is non-deterministic!
    (x may be 5, or 6, or any other value!)

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 11

Execution and Simulation Semantics

- Motivating Example 5
  - Given:

```
behavior B1(
  int x, event e)
{
  void main(void)
  {
    x = 5;
    notify e;
  }
};
```

```
behavior B2(
  int x, event e)
{
  void main(void)
  {
    wait e;
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  event e;
  B1 b1(x,e);
  B2 b2(x,e);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: x = 6

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 12

Execution and Simulation Semantics

- Motivating Example 6
  - Given:

```
behavior B1(
  int x, event e)
{
  void main(void)
  {
    notify e;
    x = 5;
  }
};
```

```
behavior B2(
  int x, event e)
{
  void main(void)
  {
    wait e;
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  event e;
  B1 b1(x,e);
  B2 b2(x,e);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: x = 6

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 13

Execution and Simulation Semantics

- Motivating Example 7
  - Given:

```
behavior B1(
  int x, event e)
{
  void main(void)
  {
    waitfor 10;
    x = 5;
    notify e;
  }
};
```

```
behavior B2(
  int x, event e)
{
  void main(void)
  {
    wait e;
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  event e;
  B1 b1(x,e);
  B2 b2(x,e);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: x = 6

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 14

Execution and Simulation Semantics

- Motivating Example 8
  - Given:

```
behavior B1(
  int x, event e)
{
  void main(void)
  {
    x = 5;
    notify e;
  }
};
```

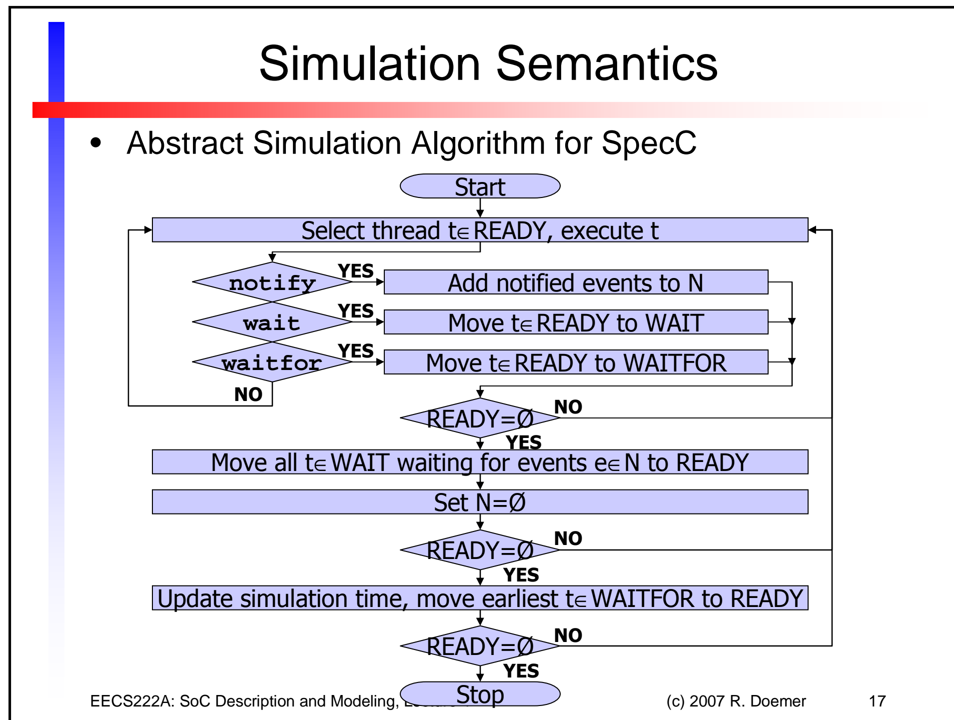
```
behavior B2(
  int x, event e)
{
  void main(void)
  {
    waitfor 10;
    wait e;
    x = 6;
  }
};
```

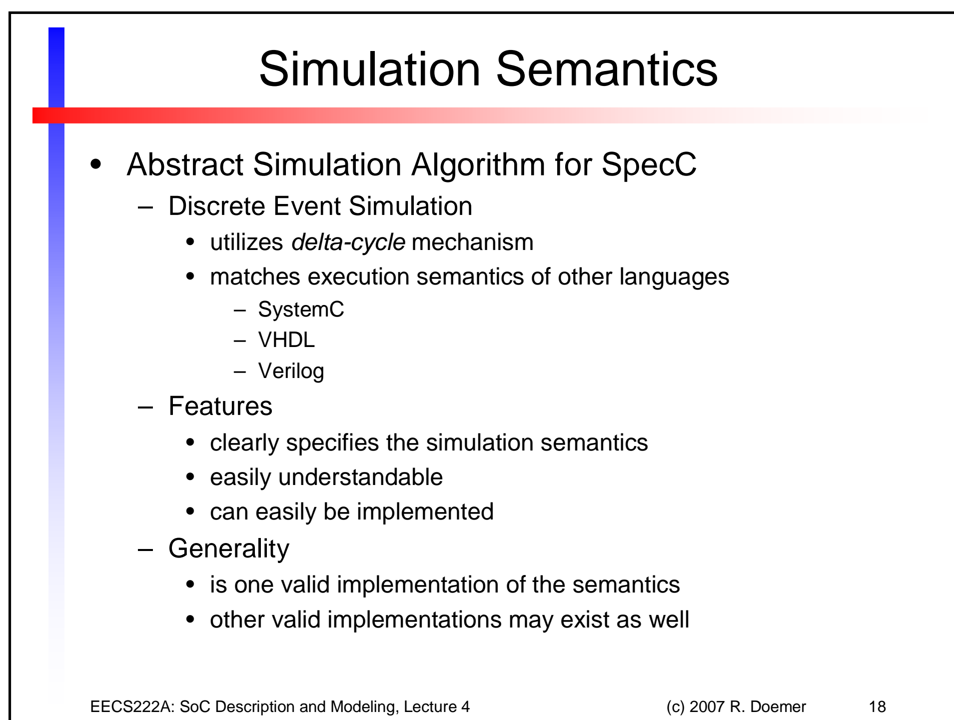
```
behavior B
{
  int x;
  event e;
  B1 b1(x,e);
  B2 b2(x,e);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: B never terminates!
    (the event is lost)

Simulation Semantics

- Abstract Simulation Algorithm for SpecC
  - available in LRM (appendix), good for understanding
  - ⇒ set of valid implementations
  - ⇒ not general (possibly incomplete)
- Definitions:
  - At any time, each thread t is in one of the following sets:
    - **READY**: set of threads ready to execute (initially root thread)
    - **WAIT**: set of threads suspended by `wait` (initially Ø)
    - **WAITFOR**: set of threads suspended by `waitfor` (initially Ø)
  - Notified events are stored in a set **N**
    - `notify e1` adds event `e1` to **N**
    - `wait e1` will wakeup when `e1` is in **N**
    - Consumption of event e means event e is taken out of **N**
    - Expiration of notified events means **N** is set to Ø


Simulation Semantics
• Abstract Simulation Algorithm for SpecC
Start
Select thread t∈READY, execute t
notify
YES
Add notified events to N
wait
YES
Move t∈READY to WAIT
waitfor
YES
Move t∈READY to WAITFOR
NO
READY=Ø
NO
YES
Move all t∈WAIT waiting for events e∈N to READY
Set N=Ø
READY=Ø
NO
YES
Update simulation time, move earliest t∈WAITFOR to READY
READY=Ø
NO
YES
Stop
EECS222A: SoC Description and Modeling,
(c) 2007 R. Doemer
17


Simulation Semantics
• Abstract Simulation Algorithm for SpecC
– Discrete Event Simulation
• utilizes delta-cycle mechanism
• matches execution semantics of other languages
– SystemC
– VHDL
– Verilog
– Features
• clearly specifies the simulation semantics
• easily understandable
• can easily be implemented
– Generality
• is one valid implementation of the semantics
• other valid implementations may exist as well
EECS222A: SoC Description and Modeling, Lecture 4
(c) 2007 R. Doemer
18

Formal Execution Semantics

- Two examples of semantics definition:
  1) Time-interval formalism
     - formal definition of timed execution semantics
     - sequentiality, concurrency, synchronization
     - allows reasoning over execution order, dependencies
  2) Abstract State Machines
     - complete execution semantics of SpecC V1.0
       - wait, notify, notifyone, par, pipe, traps, interrupts
       - operational semantics (no data types!)
     - influence on the definition of SpecC V2.0
     - straightforward extension for SpecC V2.0
     - comparable to ASM specifications of SystemC and VHDL 93

Formal Execution Semantics

- Time-interval formalism
  - Definition of execution semantics of SpecC 2.0
    - sequential execution
    - concurrent execution (semantics of **par**)
    - synchronization (semantics of **notify**, **wait**)
  - Sequential execution

```
behavior B1
{ void main(void)
  { a;
    b;
    c;
  }
};
```

Tstart(B1) <= Tstart(a) < Tend(a) <=
Tstart(b) < Tend(b) <=
Tstart(c) < Tend(c) <= Tend(B1)

B1
a
b
c
time

Formal Execution Semantics

- Time-interval formalism
  - Sequential execution
    - waitfor rule:
      - only waitfor increases simulation time
      - other statements execute in zero simulation time

```
behavior B
{ void main(void)
  { a;
    waitfor 10;
    b;
  }
};
```

0 <= Tstart(a) < Tend(a) < 1
0 <= Tstart(w) < Tend(w) = 10
10 <= Tstart(b) < Tend(b) < 11

Formal Execution Semantics

- Time-interval formalism
  - Concurrent execution

Preemptive or non-preemptive scheduling:
No atomicity guaranteed!

```
behavior B
{ void main(void)
  { par{ b1; b2;}
  }
};
```

```
behavior B1
{ void main(void)
  { a; b; c; }
};
```

```
behavior B2
{ void main(void)
  { d; e; f; }
};
```

Tstart(B) <= Tstart(a) < Tend(a) <=
Tstart(b) < Tend(b) <=
Tstart(c) < Tend(c) <= Tend(B)
Tstart(B) <= Tstart(d) < Tend(d) <=
Tstart(e) < Tend(e) <=
Tstart(f) < Tend(f) <= Tend(B)

Possible Schedule

Formal Execution Semantics

- Time-interval formalism
  - Atomicity
    - Since there is no atomicity guaranteed, a safe mechanism for mutual exclusion is necessary
    - SpecC 2.0:
      - A mutex is implicitly contained in each channel instance
      - Each channel method implicitly acquires the mutex when it starts execution and releases the mutex again when it finishes
      - An acquired mutex is also released at **`wait`** and **`waitfor`** statements and will be re-acquired before execution resumes
    - This easily enables safe communication without unnecessary restrictions to the implementation!

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 23

Formal Execution Semantics

- Time-interval formalism
  - Synchronization

```
behavior B
{ void main(void)
  { par{ b1; b2;}
  }
};
```

```
behavior B1
{ void main(void)
  { a; wait e;    b; }
};
```

```
behavior B2
{ void main(void)
  { c; notify e; d; }
};
```

Tstart(B) <= Tstart(a) < Tend(a) <=
Tstart(w) < Tend(w) <=
Tstart(b) < Tend(b) <= Tend(B)

Tstart(B) <= Tstart(c) < Tend(c) <=
Tstart(n) < Tend(n) <=
Tstart(d) < Tend(d) <= Tend(B)

Tend(w) >= Tend(n)

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 24

Formal Execution Semantics

- Abstract State Machine (ASM)
  - aka. Evolving Algebras (Y. Gurevich, 1987)
  - ASM semantics already exist for
    - Prolog, Concurrent Prolog
    - C, C++, Java
    - VHDL, VHDL-AMS, SystemC
  - ASM semantics for SpecC published at ISSS'02
- ASM components
  - Sequence of algebras (functions over domains): *states*
  - Rules define updates of functions: *state transitions*

Abstract State Machine (ASM)

ASM: SpecC Kernel Semantics

- Phase 1: at least one BEHAVIOR is running
- Phase 2: all BEHAVIORs are not running

ASM: SpecC Behavior Semantics

p ∈ BEHAVIOR:
status(p) ∈ {running, waiting, interrupted, completed}

- modelling execution of statements of behavior "Self"
  Self executes <statement> ≡
  programCounter(Self) = <statement> ∧ status (Self) = running
- wait statement
  if Self executes <wait(EventList)>
  then status(Self) := waiting,
  sensitivity (Self) := EventList,
  programCounter(Self) := nextStmt(Self)
  endif;

ASM: SpecC Statement Semantics

- modelling execution of statements of behavior "Self"

  Self executes <statement> ≡
  programCounter(Self) = <statement> ∧ status (Self) = running

- wait statement

  if Self executes <wait(EventList)>
  then status(Self) := waiting,
  sensitivity (Self) := EventList,
  programCounter(Self) := nextStmt(Self)
  endif;

- notify statement

  if Self executes <notify(EventList)>
  then ∀ e ∈ EventList: notified(e) := true,
  programCounter(Self) := nextStmt(Self)
  endif;

- The simulation kernel sets each behavior to
  status(b):= running if ∃e: notified(e) = true ∧ e ∈ sensitivity (b)

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 29

ASM: SpecC Summary

- Formal Semantics of SpecC Execution
  - complete execution semantics of SpecC V1.0 by ASMs
    - wait, notify, notifyone, par, pipe, traps, interrupts
    - operational semantics (no data types!)
  - can be easily extended to V2.0
  - influenced the definition of SpecC V2.0
  - SpecC ASM specification is comparable to other ASM specifications
    - SystemC
    - VHDL 93

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2007 R. Doemer 30