EECS 222: Embedded System Modeling Lecture 10

Rainer Dömer

doemer@uci.edu

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

Lecture 10: Overview

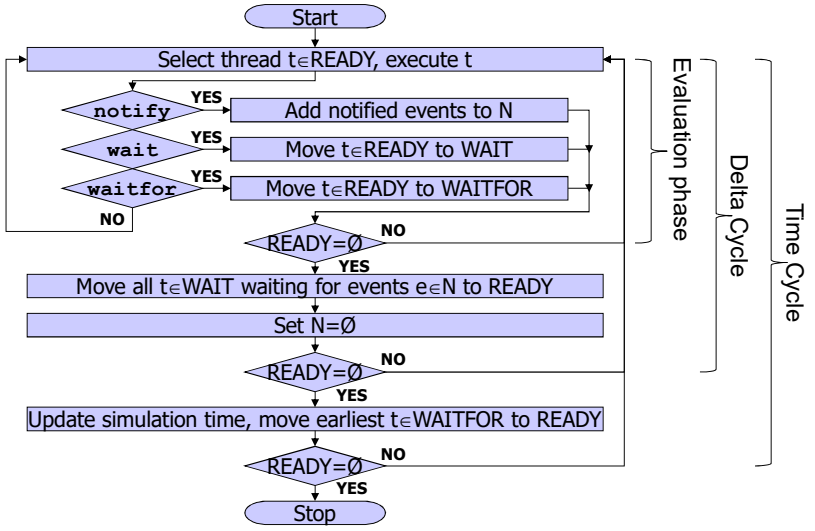
- Simulation Semantics
  - Discrete Event Simulation
  - Parallel Discrete Event Simulation
  - Out-of-Order Parallel Discrete Event Simulation
- Formal Execution Semantics
  - Time-Interval Formalism
- Homework Assignment 4
  - SLDL Model of the Canny Edge Detector

Simulation Semantics

- Discrete Event Simulation Algorithm for SpecC
  - available in LRM (appendix), good for documentation
  - ⇒ abstract definition (defines a 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

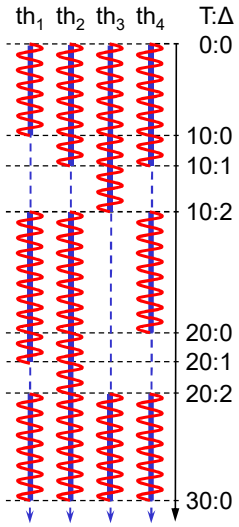
Simulation Semantics

- Discrete Event Simulation Algorithm for SpecC



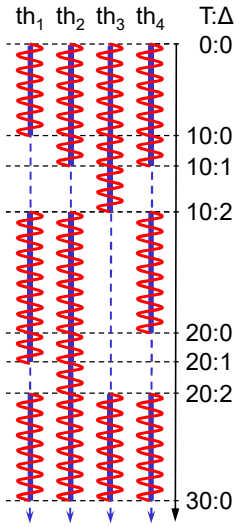
Discrete Event Simulation (DES)

- Traditional DES
  - Concurrent threads of execution
  - Managed by a central scheduler
  - Driven by events and time advances
    - Delta cycle
    - Time cycle
  - ➢ Partial temporal order with barriers
- Reference Simulators
  - Both SystemC and SpecC implement cooperative multi-threading
  - ➢ A single thread is active at any time!
  - ➢ Cannot exploit multiple parallel cores
  - Example: Execution of four threads



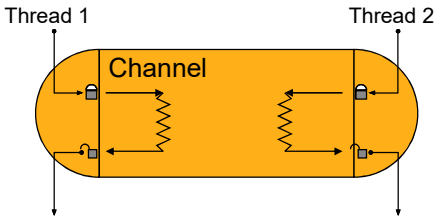
Discrete Event Simulation (DES)

- ➢ Parallel Simulation!?
- SLDL Execution Semantics
  - SystemC prescribes *Cooperative Multi-Threading*
    - SystemC LRM defines: *"process instances execute without interruption"*
    - ➢ Preemptive scheduling forbidden!
  - SpecC specifies *Preemptive Multi-Threading*
    - SpecC LRM defines: *"preemptive execution", "No atomicity is guaranteed"*
    - ➢ Preemptive scheduling assumed!
    - ➢ Need critical regions with mutually exclusive access: Channels!



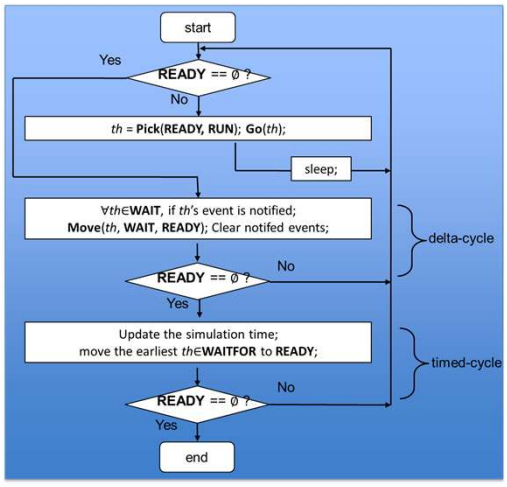
Discrete Event Simulation (DES)

- Parallel Simulation!?
- Safe Communication in Parallel Execution Context
  - Requires protection of inter-thread communication!
  - SpecC
    - Preemptive multi-threading mandates channels as *"monitors"*
  - SystemC
    - Cooperative multi-threading assumes execution *"without interruption"*
  - Protection Idea: Insert a mutex lock into channel instances
    - Lock the channel on thread entry
    - Unlock the channel on thread exit



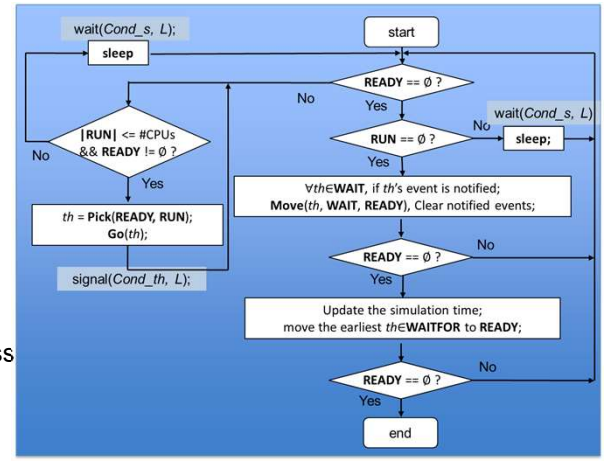
Parallel Discrete Event Simulation (PDES)

- Traditional DES Algorithm
  - Active Threads are managed in READY queue
  - Scheduler picks a *single* thread and executes it
  - Simulation progress
    - Delta cycle
    - Time cycle



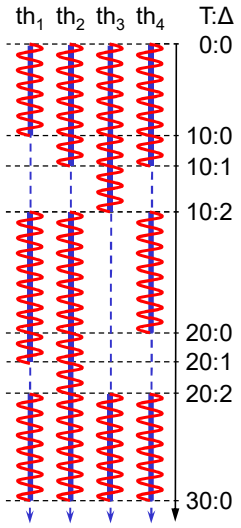
Parallel Discrete Event Simulation (PDES)

- Parallel DES Algorithm
  - Active threads are managed in READY queue
  - Scheduler *picks N threads* and executes them *in parallel*
  - *N* = number of available CPU cores
  - Simulation progress
    - Delta cycle
    - Time cycle



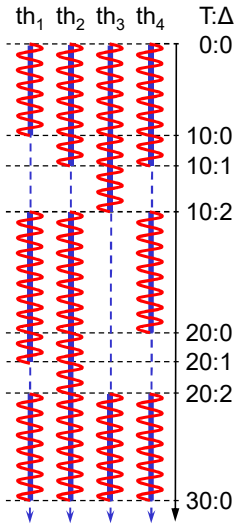
Parallel Discrete Event Simulation (PDES)

- Parallel DES
  - Threads execute in parallel *iff*
    - in the same delta cycle, *and*
    - in the same time cycle
  - ➢ Significant speed up!
  - Cycle boundaries are absolute barriers
- Aggressive Parallel DES
  - Conservative Approaches
    - Careful static analysis prevents conflicts
  - Optimistic Approaches
    - Conflicts are detected and addressed (*roll back*)



Out-of-Order Parallel DES

- Out-of-Order PDES
  - Threads execute in parallel *iff*
    - in the same delta cycle, *and*
    - In the same time cycle,
    - *OR* if there are no conflicts!
  - Needs compiler support for static data conflict analysis!
  - ➢ Preserves the accuracy of event handling and simulation time
  - ➢ Allows as many threads in parallel as possible
  - ➢ Results in higher speedup!
    - Reference: [DATE'12]



Formal Execution Semantics

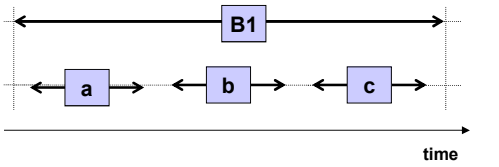
- Examples of Formally Defined Semantics
  1) Time-interval formalism
     - Formally defines timed execution semantics of SpecC
     - Covers sequentiality, concurrency, synchronization
     - Allows reasoning over execution order, dependencies
     - ➢ Discussed in the following slides!
  2) Abstract State Machines (ASM)
     - Complete execution semantics of SpecC
       - wait, notify, notifyone, par, pipe, try-trap-interrupt
       - Operational semantics only (no data types!)
     - Abstract model closely matches SystemC
     - Abstract model closely matches VHDL, Verilog
     - ➢ Not discused in this course

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)

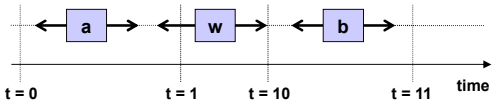


Formal Execution Semantics

- 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!

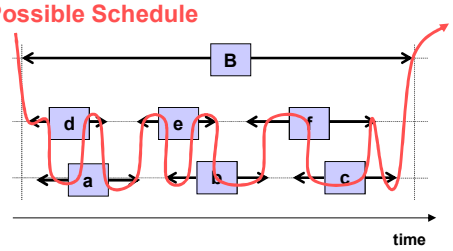
```
behavior B
{ void main(void)
  { par{ b1; b2;}
  }
};
```

```
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
  - Synchronization

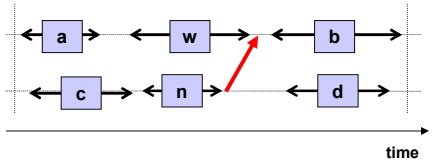
```
behavior B
{ void main(void)
  { par{ b1; b2;}
  }
};
```

```
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)



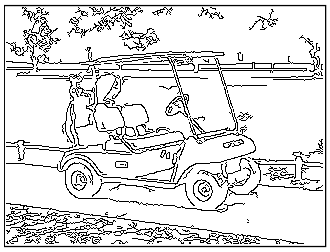
Formal Execution Semantics

- Time-interval formalism
  - Atomicity
    - Since there is generally no atomicity guaranteed, a safe mechanism for mutual exclusion is necessary
    - SpecC 2.0: Channels behave as *Monitors*!
      - 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
      - **`wait`** and **`waitfor`** statements implicitly (and atomically!)
        - » *release* an acquired mutex in a channel, and
        - » *re-acquire* the mutex before execution resumes
      - ➢ This easily enables safe communication without heavy restrictions to the implementation!

EECS 222 Project

- Application Example: Canny Edge Detector
  - Embedded system model for image processing: Automatic Edge Detection in a Digital Camera

golfcart.pgm



golfcart.pgm_s_0.60_l_0.30_h_0.80.pgm

  - Application Source and Documentation:
    - http://marathon.csee.usf.edu/edge/edge_detection.html
    - http://en.wikipedia.org/wiki/Canny_edge_detector

Homework Assignment 4

- Task: SLDL Model of the Canny Edge Detector
  - Convert ANSI-C source code into SLDL model
  - Choose either SpecC or SystemC for simulation
- Steps
  1. Fix the off-by-one bug in the **`non_max_supp`** function
  2. Clean-up the code for compilation without warnings
  3. Fix configuration parameters to compile-time constants
  4. Remove or replace dynamic memory allocation
- Deliverables
  - **`Canny.sc`** or **`Canny.cpp`** (choose one!)
  - **`Canny.txt`**
- Due
  - By next week: May 8, 2017, 12pm (noon!)