Lecture 4 – Language Semantics

Andreas Gerstlauer
Electrical and Computer Engineering
University of Texas at Austin
gerstl@ece.utexas.edu

Lecture 4: Outline

- **Language semantics**
  - Models of concurrency & time
  - Discrete event model

- **SpecC semantics**
  - Simulation semantics
  - Formal semantics
### System-Level Language Semantics

- **Language concepts (syntax)**
  - Behavioral and structural hierarchy
  - Concurrency and time
  - Synchronization and communication
  - Exception handling
  - State transitions

- **Language semantics needed to define the meaning**
  (unambiguous & formal for modeling/simulation/synthesis)
  - Denotational semantics
    - Mathematical semantics, e.g. as functions
  - **Operational semantics**
    - Semantics of execution, e.g. in the form of a simulation algorithm

Source: R. Doemer, UC Irvine

### Recap: Models vs. Languages

- **Two level of models**
  - Models of behavior/structure (capture concepts)
  - Language semantics (execution/simulation model)
    - Dedicated syntax/semantics vs. universal language
      - Tasks/processors/busses vs. behaviors/channels vs. functions/vars/events
    - Synthesis vs. simulation
      - Some languages have 1:1 mapping
  - **System-level design languages (SLDLs)**
    - Capture concepts from computation to architectures
    - Universal underlying simulation model
    - Concurrency & time
Models of Time (Order)

- **Physical**
  - Continuous, total order
    - Physically concurrent components naturally interleaved in (very fine) time

- **Logical**
  - Discrete, partial order
    - Discrete instants of time (time tags $t_0 < t_1 < \ldots$)
    - No events (state changes) between time instants
    - Unspecified interleaving of concurrent events (same time tag)
      - Freedom of implementation

- **Untimed**
  - Partial order based on causality only
    - No ordering in time, explicit dependencies only
      - Free of implementation (purely behavioral)

Logical Concurrency

- Events/actions happening “at the same (logical) time”
  - Communication/synchronization establishes internal order
  - Implementation/simulation determines actual interleaving

- Non-determinism due to undefined order
  - Outputs in relation to other outputs and inputs (behavior of system?)
  - Explicit dependencies to define required order
  - Order in the presence of causality loops?
Synchronous Reactive (SR) Model

- **Synchronous hypothesis**
  - Sequence of input events (total order)
  - Discrete steps (logical clock)
  - Reactions are instantaneous (zero time)
  - Simultaneous (broadcast)
    - Deterministic, static verifiable (independent of actual delays)

- **Synthesis challenges:**
  - Semantics: causality loops, conflicts?
  - Implementation: ensure that $\sum \Delta t << \text{clock}$, realize broadcast events, requires global clock (over-specification?)

- **Synchronous languages**
  - Imperative (control) [Esterel] or declarative (data) [Lustre] style
  - Reject cycles [Lustre] or require unique fixed-point [Esterel]
  - Hardware (FSMs) or software (safety critical) compilers

Discrete Event (DE) Model

- **Asynchronous, relax relations**
  - Signals = streams of events
    - Event $e_i = (value, tag)$, discrete time tags
  - Global event and evaluation order
    - Execute block on input event
    - Process input and generate output events with $tag + \Delta t$
  - Flexible
    - arbitrary dynamic delays
  - Efficient
    - only evaluate when necessary (event)

- **Synthesis challenges**
  - Semantics: simultaneous events, zero-delay cycles (non-determinism)
  - Implementation: global order (maintain global notion of time)

- **Execution and simulation model**
  - General, universal model for system simulation (multi-scale)
  - Hardware-description [VHDL, Verilog], system-level [SpecC, SystemC]
Discrete Event Semantics

• Motivating example 1
  • Given:

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

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

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

    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

Source: R. Doemer, UC Irvine

Discrete Event 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!)

Source: R. Doemer, UC Irvine
Discrete Event 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

Source: R. Doemer, UC Irvine

---

Discrete Event 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!)

Source: R. Doemer, UC Irvine
Discrete Event 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

Source: R. Doemer, UC Irvine

Simultaneous Events

- Depending on simulator
  - Process C might be invoked once, observing both inputs in one invocation
  - Process C might be invoked twice, processing events one at a time
    - Non-deterministic order of event processing

```
behavior C(event a, event b) {
    void main(void) {
        while(true) {
            wait(a, b); // a || b
            ...
        }
    }
}
```

Suppose B is invoked first

Source: M. Jacome, UT Austin.
Delta (Superdense) Time

- Two-level model of time
  - Break each time instant into multiple delta steps
  - Each "zero" delay event results in a delta step
  - Delta time has zero delay but imposes semantic order

➢ Resolve some non-determinism
  - Ambiguity still exists
    - Shared memory accesses in same delta cycle
      - B and C accessing same x?
    - Undefined order, non-deterministic!

Discrete Event Semantics

- Motivating example 5
  - Given:

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

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

```c
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
Discrete Event 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;
    }
  }
  
  void main(void) {
    par{b1; b2;}
  }
  ```

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

Source: R. Doemer, UC Irvine

Discrete Event Semantics

- Motivating example 7
  - Given:
  
  ```
  behavior B1(int x, event e) {
    void main(void) {
      notify e;
      x = 4;
      notify e;
      x = 5;
    }
  }
  
  behavior B2(int x, event e) {
    void main(void) {
      wait e;
      x = 6;
      wait e;
      x = 7;
    }
  }
  
  void main(void) {
    par{b1; b2;}
  }
  ```

  - What is the value of x after the execution of B?
  - Answer: B2 never terminates (x = 6)

Source: R. Doemer, UC Irvine
Discrete Event Semantics

• Motivating example 8
• 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;
    }
};
```

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

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

Source: R. Doemer, UC Irvine

Discrete Event Semantics

• Motivating example 9
• 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;
    }
};
```

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

• What is the value of x after the execution of B?
• Answer: B2 never terminates!
  (the event is lost)

Source: R. Doemer, UC Irvine
Zero-Delay Feedback Loops

- **Causality loop**
  - Where to start & end?

  ➢ **Reject zero-delay cycles**
    ➢ Forbid (every $\Delta t$ must be strictly $> 0$) [DEVS]
    ➢ Detect (compile error on zero cycle)

  ➢ **Delta cycle semantics**
    ➢ Oscillate

  ➢ **Other approaches based on topological sorting**
    ➢ Establish precedence relationship
    ➢ Annotate feedback arcs to “break” for ordering purposes

Source: M. Jacome, UT Austin.

### Deterministic Communication

- **Given**

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

  behavior B2(
      in int x, event e)
  {
    int y, z;
    void main(void)
    {
      y = x;
      wait e;
      z = x;
    }
  };

  behavior B
  {
    int x = 0;
    event e;
    B1 b1(x,e);
    B2 b2(x,e);
    void main(void)
    {
      par{b1; b2;}
    }
  };
  ```

- What is the value of $(y,z)$ at the end of execution?
- **Answer:** $z = 5$, $y$ is non-deterministic (0 or 5)
Deterministic Communication

- Delta-semantics for events & variables [VDHL, Verilog]
- Signals combine event with current/new value updates

![](image1)

- What is the value of \((y, z)\) at the end of execution?
- Answer: \(z = 5, y = 0\)
- Resolves concurrent read & writes
- Concurrent writes still a problem
  - Non-deterministic [SpecC], resolution functions [VHDL, Verilog]

Deterministic Communication

- Avoid event loss by combining event with flag

![](image2)

- What is the value of \(x\) at the end of execution?
- Answer: actually, can end up being \(x = 5\), why?
  - Race conditions, interleaving of B1 and B2 if \(D1 == D2\)
- Encapsulate in channel
  - Code in SpecC channel methods is atomic!
Lecture 4: Outline

- Language semantics
  - Models of concurrency & time
  - Discrete event model

- SpecC semantics
  - Simulation semantics
  - Formal semantics

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)

Source: R. Doemer, UC Irvine
SpecC Language Semantics

- **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)

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

Source: R. Doemer, UC Irvine
Simulation Semantics

- Abstract simulation algorithm for SpecC

```
Select thread t ∈ READY, execute t

• notify
  - Add notified events to N

• wait
  - Move t ∈ READY to WAIT

• waitfor
  - Move t ∈ READY to WAITFOR

• 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
```

Source: R. Doemer, UC Irvine
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 use cooperative multi-threading
    - A single thread is active at any time!
    - Cannot exploit multiple parallel cores
  - Example: SystemC

---

Parallel Discrete Event Simulation (PDES)

- **SLDL semantics**
  - SystemC prescribes *Cooperative Multi-Threading*
    - SystemC LRM defines: "process instances execute without interruption"
    - Preemptive interleaving forbidden
    - Parallelizing not possible
  - SpecC specifies *Preemptive Multi-Threading*
    - SpecC LRM defines: "preemptive execution", "No atomicity is guaranteed"
    - Preemptive interleaving assumed
    - Can be parallelized
    - Need critical regions with mutually exclusive access: Channels!
      - Locks inserted by compiler in parallel mode

---

Source: R. Doemer, UC Irvine
Parallel Discrete Event Simulation (PDES)

- **Parallel DE simulation algorithm**
  - Threads managed in READY queue
  - Parallel delta cycle
    - Scheduler picks \( N \) threads and executes them in parallel
    - \( N \) = number of available CPU cores
  - Time advances
    - In delta-cycle
    - In timed-cycle

  ![Parallel DE simulation algorithm diagram]

- **Parallel DES**
  - Threads execute in parallel iff
    - in the same delta cycle, \textit{and}
    - in the same time cycle
  - Significant speed up!
  - But: Amdahl’s Law still applies!
    - Cycle bounds are absolute barriers

  ![Parallel DES diagram]

- **Aggressive PDES**
  - Optimistic
    - Let threads run ahead in time
    - Rollback if conflict detected (event in the past)
  - Conservative
    - Only run ahead if guaranteed no conflicts
    - E.g. static code/dependency analysis

Source: R. Doemer, UC Irvine
Lecture 4: Outline

- Language semantics
  - Models of concurrency & time
  - Discrete event model
- SpecC semantics
  - Simulation semantics
  - Formal semantics

Formal Execution Semantics

- Two examples of semantics definition:
  1) Time-interval formalism
     - Definition of execution semantics of SpecC V2.0 [SpecC LRM'02]
     - Formal definition of timed execution semantics
     - Sequentiality, concurrency, synchronization
     - Allows reasoning over execution order, dependencies
  2) Abstract State Machines
     - Formalism of Evolving Algebras [Gurevich'87]
     - Complete execution semantics of SpecC V1.0 [Mueller'02]
       - 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

Source: R. Doemer, UC Irvine
Lecture 4: Summary

- **Discrete event (DE) model of computation**
  - Universal model for simulation of systems
    - Event = (value, tag)
    - Concurrency, time (ordering)

- **SpecC language semantics**
  - Simulation semantics
    - Simulation algorithm
    - Parallel discrete event simulation
  - Formal semantics