Lecture 4 – Language Semantics

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

Lecture 4: Outline

• **Language semantics**
  • Programming models
  • Concurrency models
  • Discrete event models

• **SpecC semantics**
  • Simulation semantics
  • Formal semantics
**Design Methodology**

- **Capture**
  - Specification model
  - Computation refinement
  - Communication refinement

- **Algorithmic (Algor) IP**
  - Compilation
  - Validation
  - Analysis
  - Estimation

- **Computation (Comp) IP**
  - Compilation
  - Validation
  - Analysis
  - Estimation

- **Communication (Comm) IP**
  - Compilation
  - Validation
  - Analysis
  - Estimation

- **Hardware Synthesis**
  - Implementation model
  - Compilation
  - Validation
  - Analysis
  - Estimation

- **Software compilation**
  - Implementation model
  - Compilation
  - Validation
  - Analysis
  - Estimation

- **RTOS IP**
  - Implementation model
  - Compilation
  - Validation
  - Analysis
  - Estimation

**System-level Language Semantics**

- **Specification concepts**
  - 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

Source: R. Doerner, UC Irvine
Programming Models

- **Imperative programming models**
  - Statements that manipulate program state, control flow
    - Sequential programming languages [C, C++, …]
- **Declarative programming models**
  - Rules for data manipulation, data flow
    - Functional programming [Haskell, Lisp, Excel]
    - Logic programming [Prolog]
- **Van Neuman execution**
  - Implicit concurrency (dependency analysis)
  - Implicit ordering (time)
  - Implicit state (set of variables)
  - Implicit termination (Turing complete)

Control and Data

- **Control/Data Flow Graph (CDFG)**
  - **Control**
    - Branches, loops
    - Dependencies, order of basic blocks (BB) of computation
    - Operations
  - **Data**
    - Basic blocks (BBs) with expressions
    - Directed, acyclic graph of operations
    - Values

- **Graphical representation (syntax) of program**
Reactive Systems

- **Transformative**
  - Output = F(Input)
  - Data processing

- **Reactive**
  - Continuous interaction with environment
    - Interactive (servers, GUIs) and constrained (embedded)
  - Ordering, interleaving of inputs and outputs (when)?
    - Timing defines reactive behavior (e.g. in case of feedback)

Concurrency

- Interaction, ordering relationship between blocks
  - Control or data dependencies, communication (what)
    - Synchronization, time (when)

- Physical: total order (real time)
  - Distinct components naturally interleaved in (very fine) time

- Logical: partial order (logical time)
  - Specification of interleaving based on causality only (implementation freedom)
Synchronous Model

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

- **Challenges**:
  - Semantics: cycles, conflicts?
  - Implementation: $\Delta t << \text{clock}$, global clock, shared events

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

Discrete Event

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

- **Challenges**
  - Semantics: simultaneous events, zero-delay cycles?
  - Implementation: maintain delays (global time)

- **Design language execution and simulation semantics**
  - General, universal modeling and simulation (multi-scale)
  - Hardware-description [VHDL, Verilog], system-level [SpecC, SystemC]
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;
    }
}
```

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

What is the value of x after the execution of B?

Answer: x = 6

Source: R. Doemer, UC Irvine

---

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)
    {
        b1; b2;
    }
}
```

```
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
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;
      }
    };
    
    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

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)
      {
        x = 6;
      }
    };
    
    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
Execution and Simulation Semantics

- **Motivating example 5**
  - Given:

    ```c
    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;
        }
    };
    
    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

---

Execution and Simulation Semantics

- **Motivating example 6**
  - Given:

    ```c
    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
**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;
    }
    }
    }
    
    void main(void) {
    par{b1; b2;}
    }
    ```
  - What is the value of x after the execution of B?
  - Answer: x = 6

- **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: B never terminates! (the event is lost)
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

Suppose B is invoked first

A
B
C

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

Source: M. Jacome, UT Austin.

Delta 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?

Source: M. Jacome, UT Austin.
Lecture 4: Outline

- Language semantics
  - Programming models
  - Concurrency models
  - Discrete event models

- 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)

Source: R. Doemer, UC Irvine

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 \( \text{wait} \) (initially \( \emptyset \))
    - **WAITFOR:** set of threads suspended by \( \text{waitfor} \) (initially \( \emptyset \))

  - Notified events are stored in a set \( N \)
    - \( \text{notify} \ e_1 \) adds event \( e_1 \) to \( N \)
    - \( \text{wait} \ e_1 \) will wakeup when \( e_1 \) is in \( N \)
    - Consumption of event \( e \) means event \( e \) is taken out of \( N \)
    - Expiration of notified events means \( N \) is set to \( \emptyset \)

Source: R. Doemer, UC Irvine
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

Source: R. Doemer, UC Irvine
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

```c
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

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

source: R. Doemer, UC Irvine

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

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

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

source: R. Doemer, UC Irvine

```
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!

---

Formal Execution Semantics

- **Time-interval formalism**
  - **Synchronization**

Source: R. Doemer, UC Irvine
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

Source: R. Doemer, UC Irvine

Abstract State Machine (ASM)

<table>
<thead>
<tr>
<th>Algebra A</th>
<th>Algebra A'</th>
</tr>
</thead>
<tbody>
<tr>
<td>g = 0</td>
<td>g = 0</td>
</tr>
<tr>
<td>f(0) = undef</td>
<td>f(0) = 42</td>
</tr>
<tr>
<td>f(0,0) = 23</td>
<td>f(0,0) = 0</td>
</tr>
<tr>
<td>f(0,1) = 6</td>
<td>f(0,1) = 6</td>
</tr>
</tbody>
</table>

**Rules**
- if f(0) = undef then f(0) := 42 else f(0) := 77
- if f(0,0) = 0 then f(0,0) := 23 else f(0,0) := 0
- f(0) := 42
- f(0,0) := 0

**Update Set**
- if f(0) = undef then f(0) := 42 else f(0) := 77
- if f(0,0) = 0 then f(0,0) := 23 else f(0,0) := 0
ASM: SpecC Kernel Semantics

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

ASM: SpecC Behavior Semantics

\( p \in \text{BEHAVIOR}: \)  
\[ \text{status}(p) \in \{\text{running, waiting, interrupted, completed}\} \]
ASM: SpecC Statement Semantics

- **Modelling execution of statements of behavior “Self”**
  
  Self executes \(<\text{statement}>\) if
  
  \[ \text{programCounter}(\text{Self}) = \text{statement} \land \text{status}(\text{Self}) = \text{running} \]

- **Wait statement**
  
  if Self executes \(<\text{wait}(\text{EventList})>\)
  
  then \(\text{status}(\text{Self}) := \text{waiting}\),
  
  \(\text{sensitivity}(\text{Self}) := \text{EventList}\),
  
  \(\text{programCounter}(\text{Self}) := \text{nextStmt}(\text{Self})\)
  
  endif;

- **Notify statement**
  
  if Self executes \(<\text{notify}(\text{EventList})>\)
  
  then \(\forall e \in \text{EventList}: \text{notified}(e) := \text{true}\),
  
  \(\text{programCounter}(\text{Self}) := \text{nextStmt}(\text{Self})\)
  
  endif;

- **The simulation kernel sets each behavior to**
  
  \(\text{status}(\text{b}) := \text{running}\) if \(\exists e: \text{notified}(e) = \text{true} \land e \in \text{sensitivity}(\text{b})\)

---

**Lecture 4: Summary**

- **Programming models**
  - Imperative, declarative: C, C++, Haskel, …
  - Synchronous, reactive: Esterel, Lustre
  - Discrete event: HDLs, SpecC

- **Language semantics**
  - Simulation semantics
    - Simulation algorithm
  - Formal semantics
    - Operational semantics via ASMs