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

- Language Semantics
- Execution and Simulation Semantics
  - Motivating Examples
- Simulation Semantics
  - Discrete Event Simulation
  - Parallel Discrete Event Simulation
- Formal Execution Semantics
  - Time-Interval Formalism
  - Abstract State Machines
- Project Discussion
  - Assignment 2
  - Assignment 3

Language Semantics

- Concepts found in Embedded Systems
  - Behavioral and structural hierarchy
  - Concurrency
  - Synchronization and communication
  - Exception handling
  - Timing
  - State transitions
- SLDL 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

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)

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)

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

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!)

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

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!)

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

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

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

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;}
  }
};
```

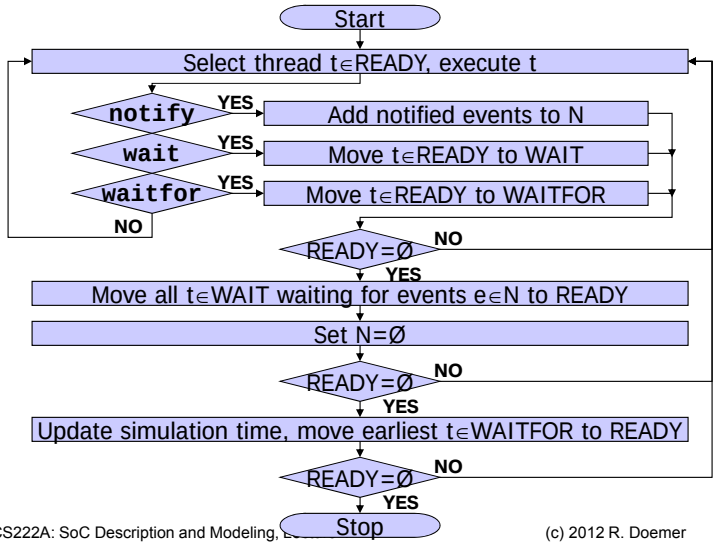
  - What is the value of x after the execution of B?
  - Answer: B never terminates!
    (the event is lost)

Simulation Semantics

- Discrete Event 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

- Discrete Event Simulation Algorithm for SpecC

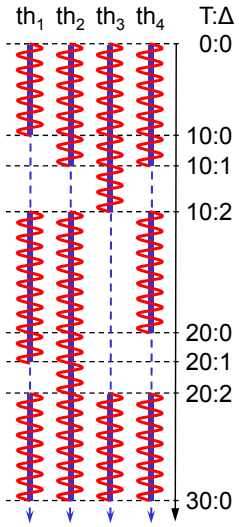


Simulation Semantics

- Discrete Event Simulation Algorithm for SpecC
  - Conforms to general Discrete Event (DE) Simulation
    - utilizes *delta-cycle* mechanism (i.e. inner event loop)
    - matches execution semantics of other languages
      - SystemC
      - VHDL
      - Verilog
  - Features
    - clearly specifies the simulation semantics
    - is easily understandable
    - can be easily implemented
  - Generality
    - is one valid implementation of the semantics
    - other valid implementations may exist as well

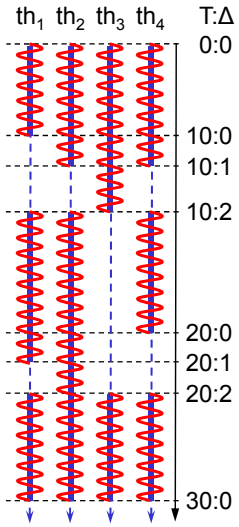
Discrete Event Simulation (DES)

- 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 TLM Simulators
  - Both SystemC and SpecC use cooperative multi-threading
  - ➢ A single thread is active at any time!
  - ➢ Cannot exploit multiple parallel cores
  - Example: SystemC



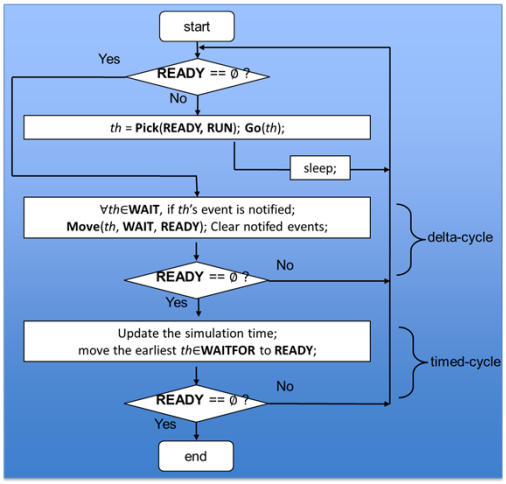
Discrete Event Simulation (DES)

- ➢ Parallel Multi-threading!?
- 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!



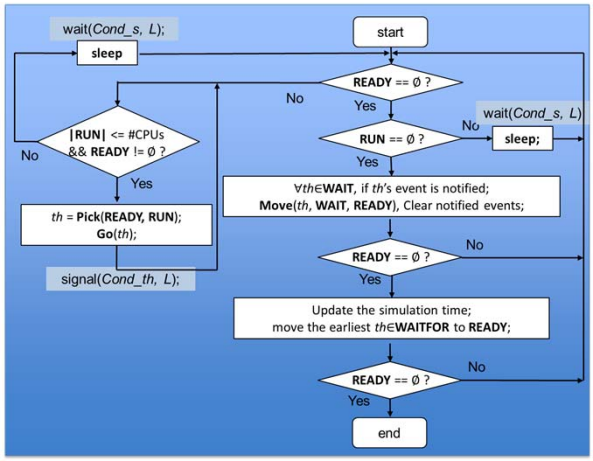
Parallel Discrete Event Simulation (PDES)

- Traditional DE Simulation Algorithm
  - Threads managed in READY queue
  - Scheduler picks a *single* thread and executes it
  - Time advances
    - In delta-cycle
    - In timed-cycle



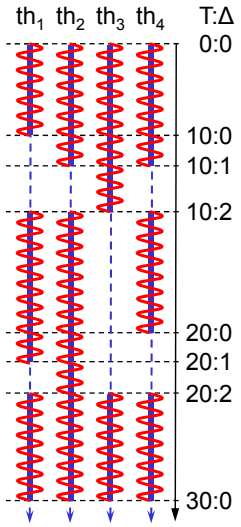
Parallel Discrete Event Simulation (PDES)

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



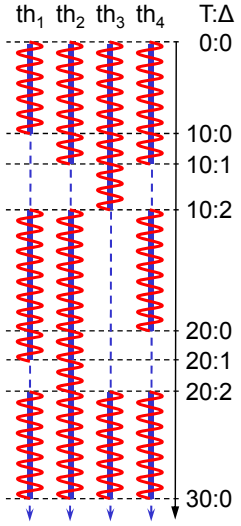
Parallel Discrete Event Simulation (PDES)

- 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

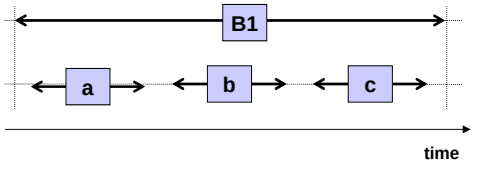
- Two examples of semantics definition:
  1) Time-interval formalism
     - formal definition of timed execution semantics
     - sequentiality, concurrency, synchronization
     - allows reasoning over execution order, dependencies
  2) Abstract State Machines
     - complete execution semantics of 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

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)

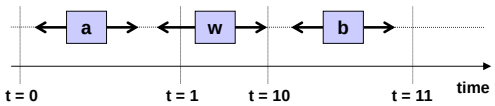


Formal Execution Semantics

- 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!

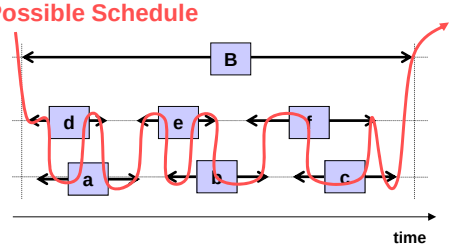
```
behavior B
{ void main(void)
  { par{ b1; b2;}
  }
};
```

```
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

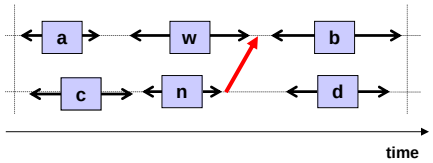
```
behavior B
{ void main(void)
  { par{ b1; b2;}
  }
};
```

```
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)



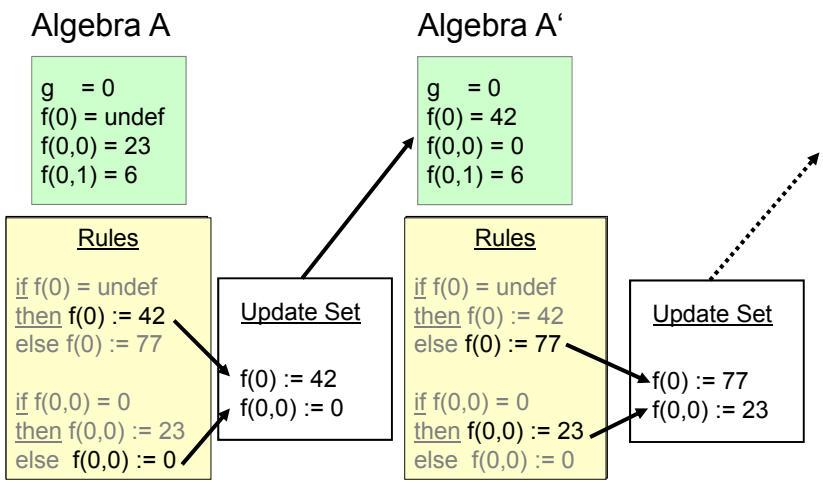
Formal Execution Semantics

- Time-interval formalism
  - Atomicity
    - Since there is 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 implicity (and atomically!)
        - » *release* an aquired mutex in a channel, and
        - » *re-acquire* the mutex before execution resumes
      - ➢ This easily enables safe communication without heavy restrictions to the implementation!

Formal Execution Semantics

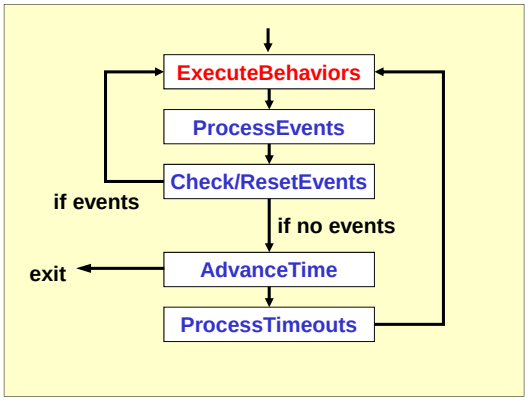
- Abstract State Machine (ASM)
  - aka. Evolving Algebras (Y. Gurevich, 1987)
  - ASM semantics already exist for
    - Prolog, Concurrent Prolog
    - C, C++, Java
    - VHDL, VHDL-AMS, SystemC
  - ASM semantics for SpecC published at ISSS'02
- ASM components
  - Sequence of algebras (functions over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

ASM: Abstract State Machine



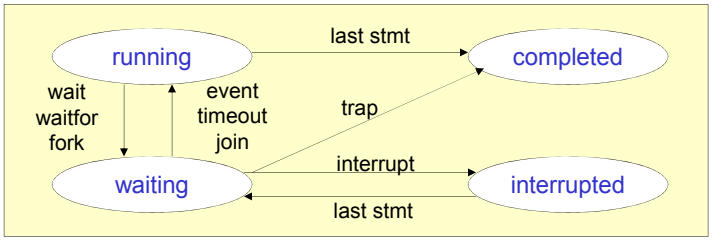
ASM: SpecC Kernel Semantics

- Phase 1: at least one BEHAVIOR is running
- Phase 2: no BEHAVIOR is running



ASM: SpecC Behavior Semantics

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



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)

ASM: SpecC Semantics 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

Homework Assignment 2

- Task: Introduction to Canny Edge Detector
  - Setup: use latest **`scc`** to edit, compile, and simulate
    - **`source /opt/sce/bin/setup.csh`**
  - Application Example
    - Canny Edge Detector: SoC for edge detection in digital camera
  - Tools
    - Image tools: command-line tools (pbm, pgm, pnm), eog, gimp
    - Editor tools: SpecC-enhanced Eclipse
  - Steps
    - Download and compile with gcc
    - Study the application
    - Convert and compile with scc (timed!)
- Deliverables
  - Source file: **`Canny.sc`**
  - Description: **`Canny.txt`**
- Due
  - By next week: April 27, 2012, 12pm (noon!)

Homework Assignment 3

- Task: Structural Hierarchy for Canny Edge Detector
  - Setup: use latest **`scc`** to edit, compile, and simulate
    - **`source /opt/sce/bin/setup.csh`**
  - Provided Files
    - canny_a3_start.sc, Makefile
    - golfcart.pgm, ref_golfcart.pgm_s_0.60_l_0.30_h_0.80.pgm
  - Eclipse Support
    - Outline View: Source code structure
    - Behavior Hierarchy: SpecC structural hierarchy
- Deliverables
  - Source file: **`canny.sc`**
  - Description: **`canny.txt`**
- Due
  - By next week: May 4, 2012, 12pm (noon!)

Homework Assignment 3

- Structural Hierarchy for Canny Edge Detector
  - Test bench Structure
    - `B i o behavior Main`
    - `B i l |------ Monitor monitor`
    - `B i c |------ Platform platform`
    - `B i l | |------ DUT canny`
    - `B i l | |------ DataIn din`
    - `B i l | |------ DataOut dout`
    - `C i l | |------ c_img_queue q1`
    - `C i l | \------ c_img_queue q2`
    - `B i l |------ Stimulus stimulus`
    - `C i l |------ c_img_queue q1`
    - `C i l \------ c_img_queue q2`