EECS 222A: System-on-Chip Description and Modeling Lecture 6

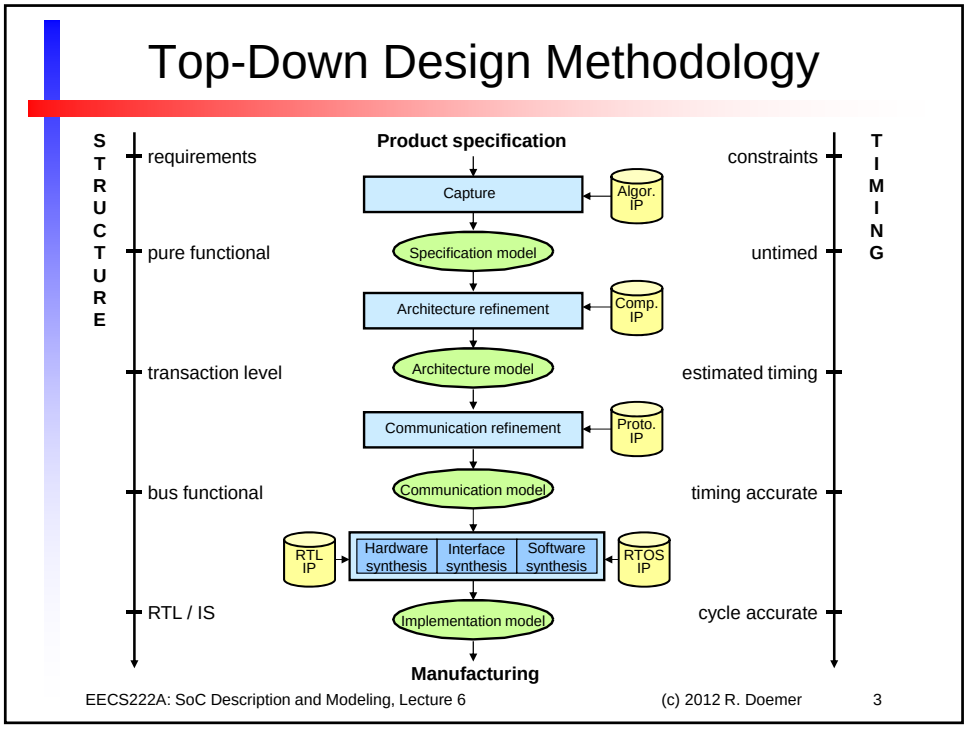
Rainer Dömer

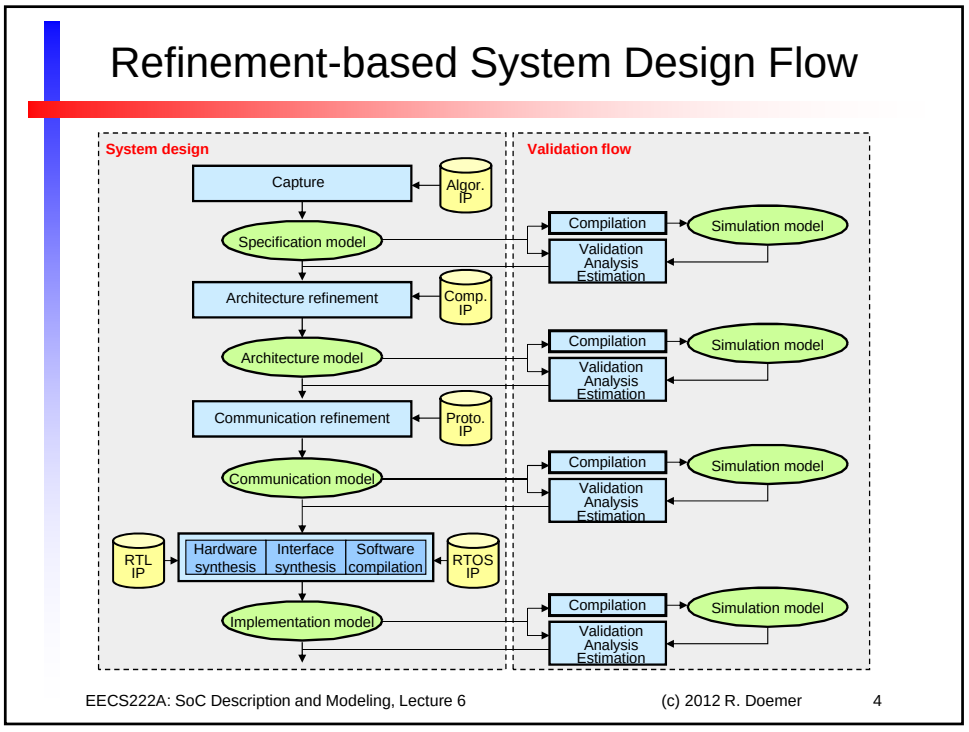
doemer@uci.edu

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

Lecture 6: Overview

- System-on-Chip Design Flow
  - Top-down design methodology
  - Refinement-based design flow
    - Specify, Explore, Refine
- System-on-Chip Environment (SCE)
  - Interactive Demonstration
  - Design Example: GSM Vocoder
- Project Discussion
  - Assignment 3
  - Intermediate Steps
  - Assignment 4


Top-Down Design Methodology
STRUCTURE
TIMING
requirements
pure functional
transaction level
bus functional
RTL / IS
constraints
untimed
estimated timing
timing accurate
cycle accurate
Product specification
Capture
Algor. IP
Specification model
Architecture refinement
Comp. IP
Architecture model
Communication refinement
Proto. IP
Communication model
RTL IP
Hardware synthesis
Interface synthesis
Software synthesis
RTOS IP
Implementation model
Manufacturing
EECS222A: SoC Description and Modeling, Lecture 6
(c) 2012 R. Doemer
3


Refinement-based System Design Flow
System design
Validation flow
Capture
Algor. IP
Specification model
Architecture refinement
Comp. IP
Architecture model
Communication refinement
Proto. IP
Communication model
RTL IP
Hardware synthesis
Interface synthesis
Software compilation
RTOS IP
Implementation model
Compilation
Validation Analysis Estimation
Simulation model
EECS222A: SoC Description and Modeling, Lecture 6
(c) 2012 R. Doemer
4

Refinement-based System Design Flow

- Refinement steps
  - Architecture refinement (Specification -> Architecture)
  - Communication refinement (Architecture -> Communication)
  - Cycle-accurate refinement (Communication -> RTL/IS)
    - HW / SW / interface synthesis
- Levels of abstraction
  - Specification model: untimed, functional
  - Architecture model: estimated, structural
  - Communication model: timed, bus-functional
  - Implementation model: cycle-accurate, RTL/IS
- Component data bases
  - Algorithms for specification
  - Components for architecture
  - Busses for communication
  - RTOS for SW
  - RTL components for HW

Refinement-based System Design Flow

- Step 1: Architecture Refinement
  - Allocation of Processing Elements (PE)
    - Type and number of processors
    - Type and number of custom hardware blocks
    - Type and number of system memories
  - Mapping to PEs
    - Map each behavior to a PE
    - Map each channel to a PE
    - Map each variable to a PE
  - Result:
    System architecture of concurrent PEs
    with abstract communication via channels

Refinement-based System Design Flow

- Step 2: Scheduling Refinement
  - For each PE, serialize the execution of behaviors to a single thread of control
  - Option (a): Static scheduling
    - For each set of concurrent behaviors, determine fixed order of execution
  - Option (b): Dynamic scheduling by RTOS
    - Choose scheduling policy, e.g. round-robin or priority-based
    - For each set of concurrent behaviors, determine scheduling priority
  - Result: System model with abstract scheduler inserted in each PE

Refinement-based System Design Flow

- Step 3: Network / Communication Refinement
  - Allocation of system busses
    - Type and number of system busses
    - Type of bus protocol for each bus (if applicable)
    - Number of transducers (if applicable)
    - System connectivity
  - Mapping of channels to busses
    - Map each communication channel to a system bus (or multiple busses, if applicable)
  - Result: Transaction-Level Model (TLM), or Bus-Functional Model (BFM)

Refinement-based System Design Flow

- Step 4: Hardware Refinement (for HW PE)
  - Allocation of Register Transfer Level (RTL) components
    - Type and number of functional units (e.g. adder, multiplier, ALU)
    - Type and number of storage units (e.g. registers, register files)
    - Type and number of interconnecting busses (drivers, multiplexers)
  - Scheduling
    - Basic blocks assigned to super-states
    - Individual operations assigned to states (clock cycles)
  - Binding
    - Bind functional operations to functional units
    - Bind variables to storage units
    - Bind assignments/transfers to busses
  - Result:
    Clock-cycle accurate model of each HW PE
  - Output: Synthesizable Verilog description

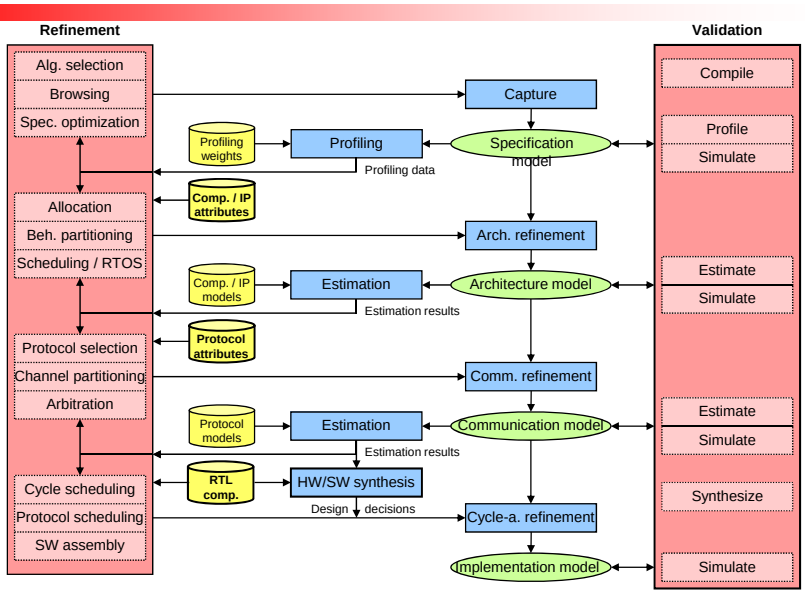
Refinement-based System Design Flow

- Step 5: Software Refinement (for SW PE)
  - C code generation
    - For selected target processor
  - RTOS targeting
    - Thin adapter layer for selected target RTOS
  - Cross-compilation to Instruction Set Architecture
    - for Instruction Set Simulation (ISS)
    - for target processor embedded in target system
  - Assembly and Linking
  - Result:
    Clock-cycle accurate, or
    instruction-accurate model of each SW PE
  - Output: binary image

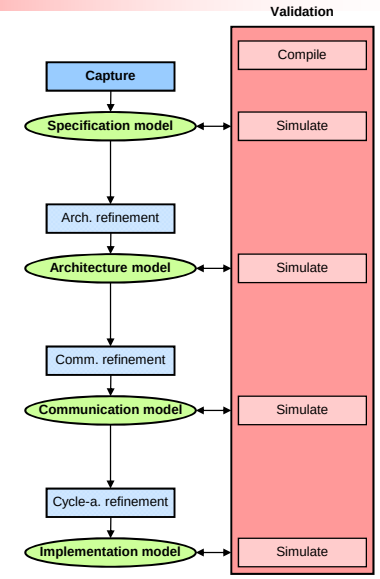
System-on-Chip Environment (SCE)

- Integrated Development Environment (IDE) with support of:
  - Graphical frontend (`sce, scchart`)
  - SLDL-aware editor (`sced`)
  - Compiler and simulator (`scc`)
  - Profiling and analysis (`scprof`)
  - Architecture refinement (`scar`)
  - RTOS refinement (`scos`)
  - Communication refinement (`sccr`)
  - RTL refinement (`scrtl`)
  - Software refinement (`sc2c`)
  - Scripting interface (`scsh`)
  - Tools and utilities ...

System-on-Chip Environment (SCE)

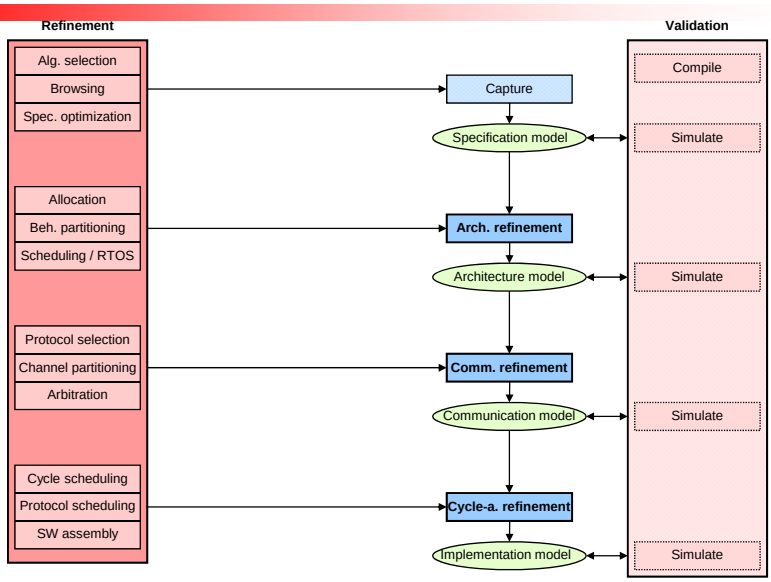


SCE Modeling Engine



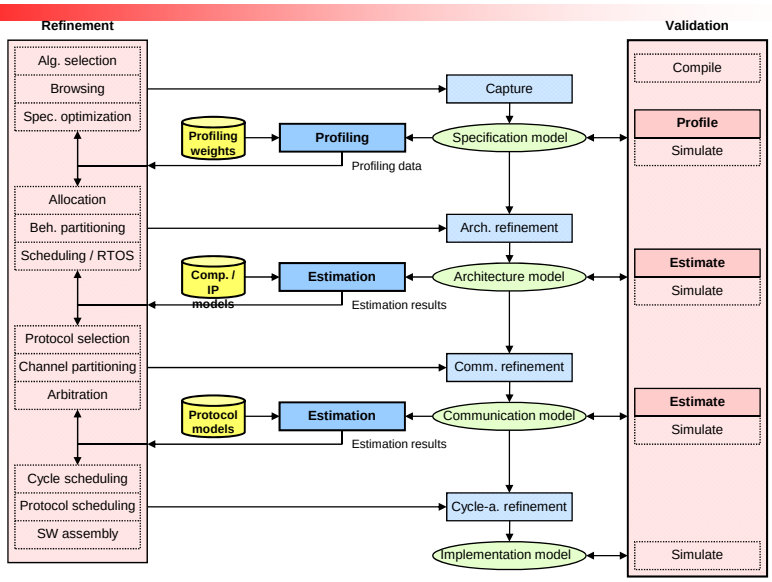
EECS222A: SoC Description and Modeling, Lecture 6 Copyright © 2003 CECS 13

SCE Refinement Engine

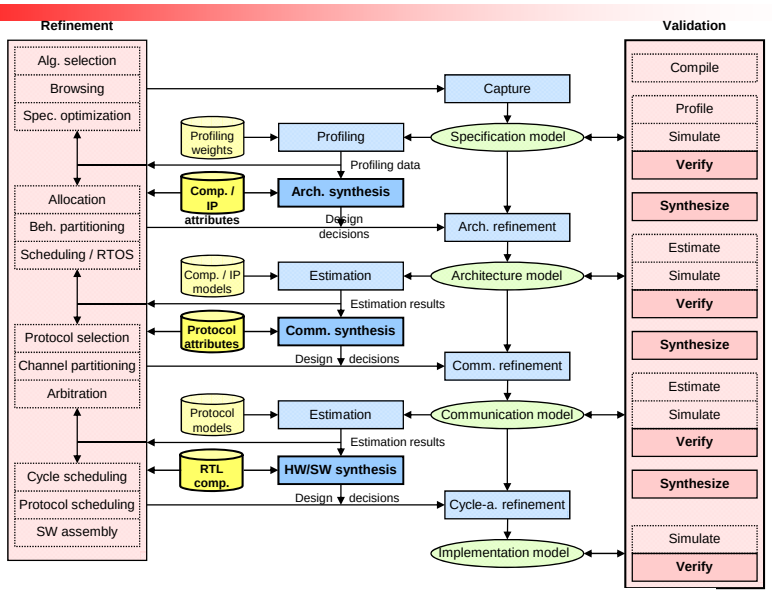


EECS222A: SoC Description and Modeling, Lecture 6 Copyright © 2003 CECS 14

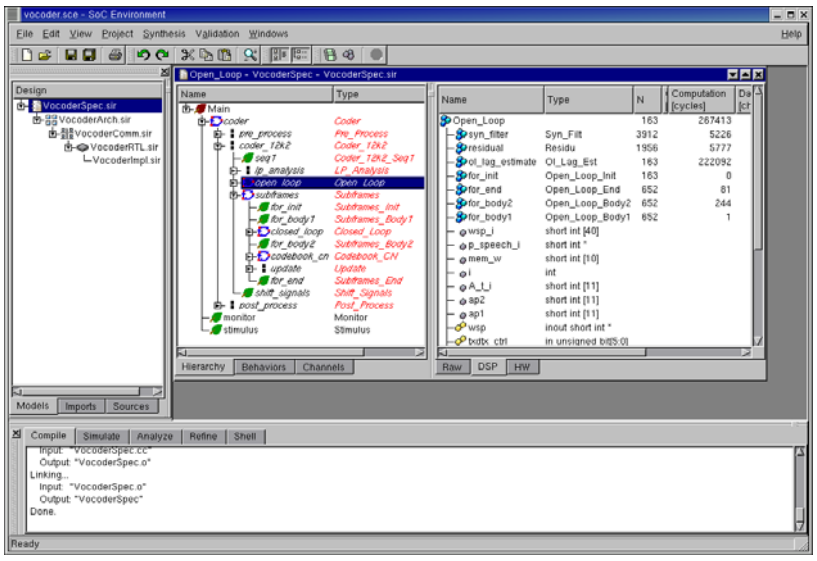
SCE Exploration Engine



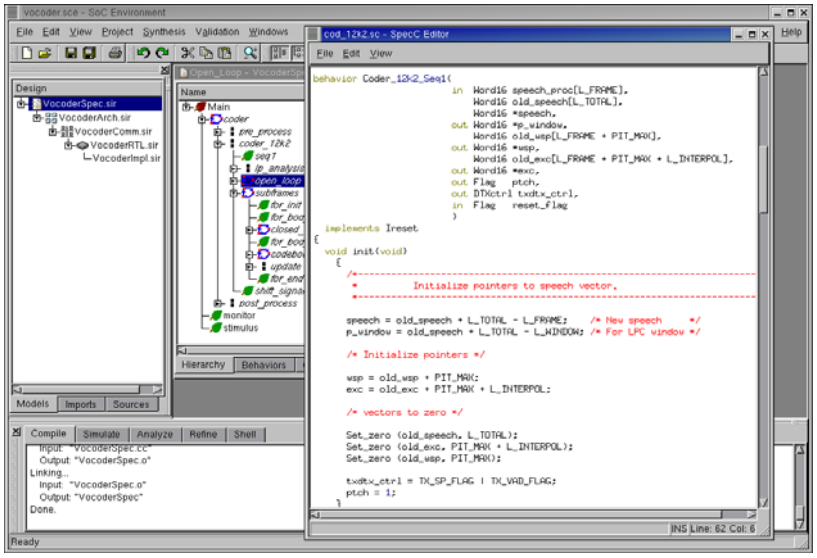
SCE Synthesis Engine



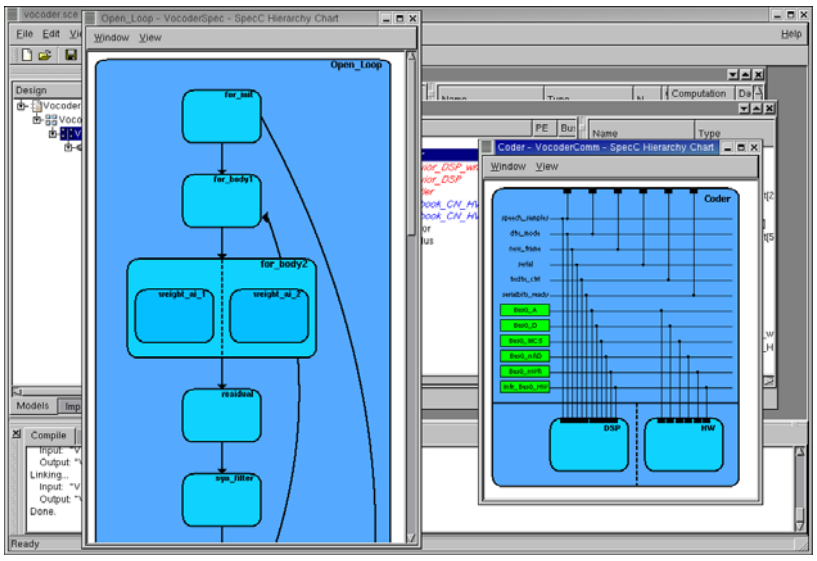
SCE Main Window



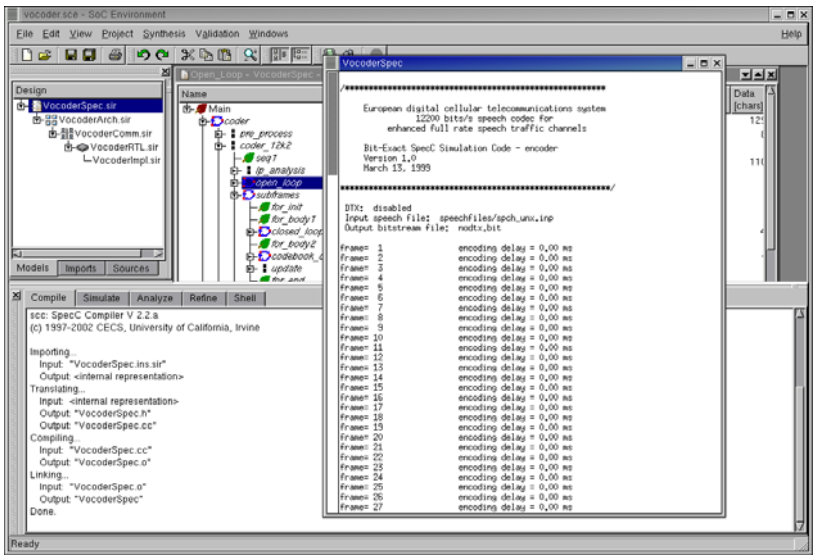
SCE Source Editor



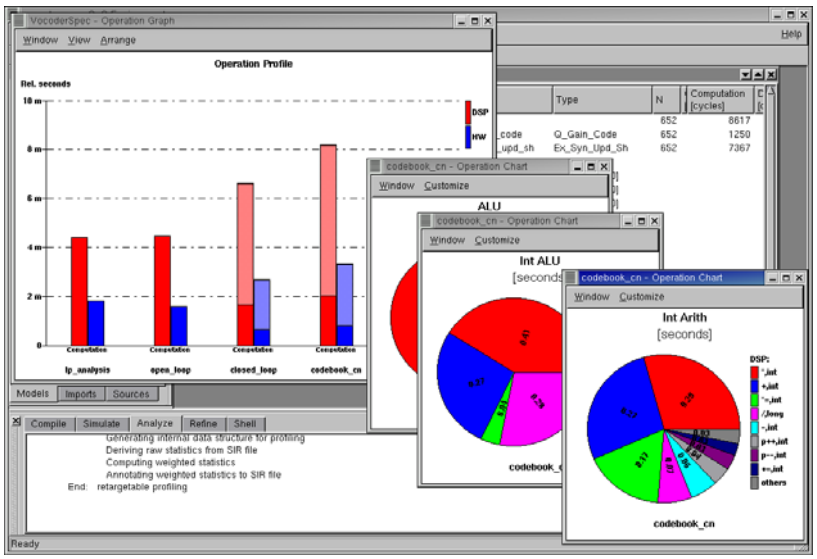
SCE Hierarchy Displays



SCE Compiler and Simulator



SCE Profiling and Analysis



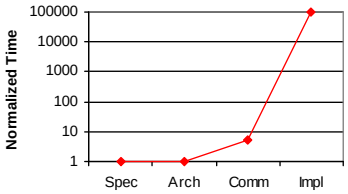
SCE Application Design Example

- GSM Vocoder
  - Enhanced full-rate voice codec
  - GSM standard for mobile telephony (GSM 06.10)
  - Lossy voice encoding/decoding
    - Incoming speech samples @ 104 kbit/s
    - Encoded bit stream @ 12.2 kbit/s
    - Frames of 4 x 40 = 160 samples (4 x 5ms = 20ms of speech)
  - Real-time constraint:
    - max. 20ms per speech frame
      (max. total of 3.26s for sample speech file)
  - SpecC specification model
    - 29 hierarchical behaviors (9 par, 10 seq, 10 fsm)
    - 73 leaf behaviors
    - 9139 formatted lines of SpecC code
      (~13000 lines of original C code, including comments)

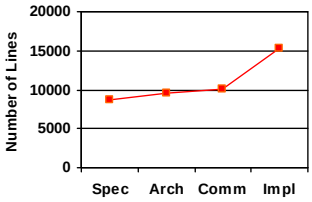
SCE Experimental Results

- Design Example 1: GSM Vocoder

Simulation Speed



Code Size



Refinement Effort

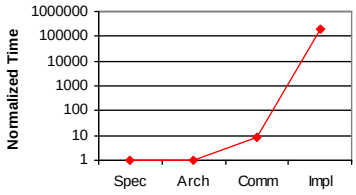
	Modified lines	Manual	Automated User / Refine
Spec → Arch	3,275	3~4 month	15 min / < 1 min
Arch → Comm	914	1~2 month	5 min / < 0.5 min
Comm → Impl	6,146	5~6 month	30 min / < 2 min
Total	10,355	9~12 month	50 min / < 4 min

- Productivity gain: 12 months vs. 1 hour = 2000x !

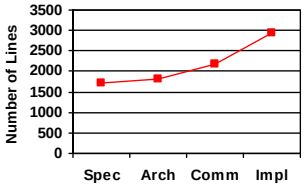
SCE Experimental Results

- Design Example 2: JPEG Encoder

Simulation Speed



Code Size



Refinement Effort

	Modified lines	Manual	Automated User / Refine
Spec → Arch	751	1~2 month	5 min / < 0.5 min
Arch → Comm	492	~1 month	3 min / < 0.5 min
Comm → Impl	1,278	3~4 month	20 min / < 1 min
Total	2,521	5~7 month	28 min / < 2 min

- Productivity gain: 6 months vs. 1/2 hour = 2000x !

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!)

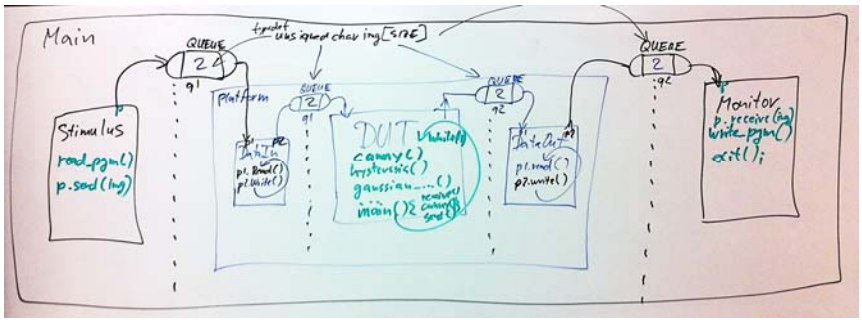
Homework Assignment 3

- 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
```

Homework Assignment 3

- Structural Hierarchy for Canny Edge Detector
  - Test bench Structure

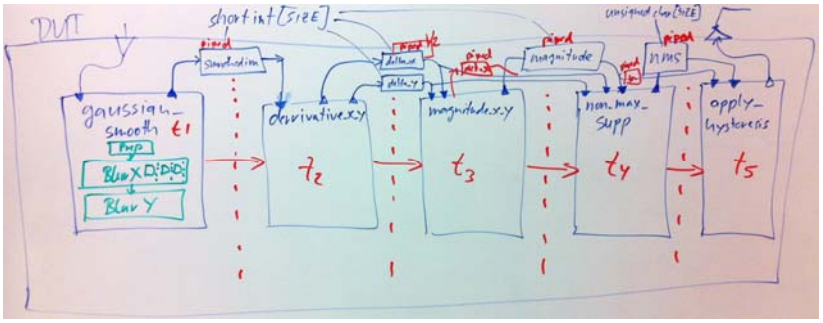


Project Discussion

- Intermediate Steps
  - Additional level of hierarchy inside DUT
    - Behavioral Composition
      - Parallel execution desirable
      - Sequential execution as needed
    - Structural Composition
      - Standard channels, or
      - Variables shared through port maps
  - Canny Edge Detector: **`canny( )`**
    - **`gaussian_smooth( )`**
      - » **`make_gaussian_kernel( )`**
    - **`derrivative_x_y( )`**
    - **`magnitude_x_y( )`**
    - **`non_max_supp( )`**
    - **`apply_hysteresis( )`**
      - » **`follow_edges( )`**

Project Discussion

- Additional Level of Hierarchy inside DUT



  - Potential for parallelism
    - 5 pipeline stages in DUT (red color)
    - Parallel decomposition of BlurX and BlurY blocks in Gaussian Smooth behavior (green color)

Homework Assignment 4

- Task: Parallelize the Gaussian Smooth Method
  - Setup: use latest **`scc`** to edit, compile, and simulate
    - **`source /opt/sce/bin/setup.csh`**
  - Provided Files
    - canny_a4_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
    - Non-local Variable View: Variable accesses and potential conflicts
- Deliverables
  - Source file: **`canny.sc`**
  - Description: **`canny.txt`**
- Due
  - By next week: May 18, 2012, 12pm (noon!)