EECS 298: Embedded Software Synthesis Lecture 5

Rainer Dömer

doemer@uci.edu

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

Lecture 5: Overview

- Project Status and Progress
- Embedded System Design with SpecC
  - The SpecC Language
  - Embedded System Design Flow
  - System-on-Chip Environment (SCE)
    - Demo

Project Status and Progress

- Option 1: Hands-on Experience with Embedded Software
  - CJC: Opt. 1 Mobile IP (embedded Linux) on wireless access point
  - CBH: Opt. 1 Port PalmOS application to WindowsCE
  - SYC+CWS: Opt. 1 Traffic light controller on Xilinx board
  - QKN + RL: Opt. 1 Temperature sensor on flash microcontroller
  - ML + HL: Opt. 1 Instant messenger application on mobile phone
  - KLN: Opt. 1 Snake game (Java) on mobile phone
  - SI: Opt. 1 Real-time UML/Java appl. wallet PDA, cash register PC
- Option 2: Literature Research
  - GK: Opt. 2 Power management for embedded applications
  - EKS: Opt. 2 Code generation for embedded processors
  - KDS: Opt. 2 Target processor survey
  - HEC: Opt. 2 RTOS survey
- Option 3: Embedded Software Synthesis using SpecC
  - JHB: Opt. 3 Reed-Solomon decoder
  - AG: Opt. 3 Digital camera
  - TWH: Opt. 3 Tic-tac-toe game
  - GS: Opt. 3 Wireless sensor node measuring motion
  - ISG: Opt. 3 Controller (traffic light, elevator, ...)
  - HCL: Opt. 3 Algorithm evaluation for fair packet scheduling

Presentation Scheduling

- Planned schedule: (to be filled in discussion)
  - Week 6: Literature research on RTOS
  - Week 7: Literature on target processors
  - Week 8: Literature / Project
  - Week 9: Project presentations
  - Week 10: Project presentations
  - Final week: Project presentations

Embedded System Design with SpecC

- The SpecC Language
- Embedded System Design Flow
- System-on-Chip Environment (SCE)
  - Demo

The SpecC Language

- Overview
  - Foundation
  - Types
  - Structural and behavioral hierarchy
  - Concurrency
  - State transitions
  - Exception handling
  - Communication
  - Synchronization
  - Timing
  - RTL

The SpecC Language

- Foundation: ANSI-C
  - Software requirements are fully covered
  - SpecC is a true superset of ANSI-C
  - Every C program is a SpecC program
  - Leverage of large set of existing programs
  - Well-known
  - Well-established
- SpecC has extensions needed for hardware
  - Minimal, orthogonal set of concepts
  - Minimal, orthogonal set of constructs
- SpecC is a real language
  - Not just a class library

The SpecC Language

- ANSI-C
  - Program is set of functions
  - Execution starts from function `main()`

```
/* HelloWorld.c */

#include <stdio.h>

void main(void)
{
  printf("Hello World!\n");
}
```

- SpecC
  - Program is set of behaviors, channels, and interfaces
  - Execution starts from behavior `Main.main()`

```
// HelloWorld.sc

#include <stdio.h>

behavior Main
{
  void main(void)
  {
    printf("Hello World!\n");
  }
};
```

The SpecC Language

- SpecC types
  - Support for all ANSI-C types
    - predefined types (**`int`**, **`float`**, **`double`**, …)
    - composite types (arrays, pointers)
    - user-defined types (**`struct`**, **`union`**, **`enum`**)
  - Boolean type: Explicit support of truth values
    - **`bool`** `b1 =` **`true;`**
    - **`bool`** `b2 =` **`false;`**
  - Bit vector type: Explicit support of bit vectors of arbitrary length
    - **`bit`**`[15:0] bv = 1111000011110000b;`
  - Event type: Support of synchronization
    - **`event`** `e;`
  - Buffered and signal types: Explicit support of RTL concepts
    - **`buffered`**`[clk]` **`bit`**`[32] reg;`
    - **`signal bit`**`[16] address;`

The SpecC Language

- Bit vector type
  - signed or unsigned
  - arbitrary length
  - standard operators
    - logical operations
    - arithmetic operations
    - comparison operations
    - type conversion
    - type promotion
  - concatenation operator
    - `a @ b`
  - slice operator
    - `a[l:r]`

```
typedef bit[7:0] byte;  // type definition
byte             a;
unsigned bit[16] b;

bit[31:0] BitMagic(bit[4] c, bit[32] d)
{
 bit[31:0] r;

 a = 11001100b;          // constant
 b = 1111000011110000ub; // assignment

 b[7:0] = a;             // sliced access
 b = d[31:16];

 if (b[15])              // single bit
    b[15] = 0b;          // access

 r = a @ d[11:0] @ c     // concatenation
     @ 11110000b;

 a = ~(a & 11110000);    // logical op.
 r += 42 + 3*a;          // arithmetic op.

 return r;
}
```

The SpecC Language

- **`buffered`** type modifier
  - Representation of storage in RTL models
    - registers
    - register files
    - memories
    - etc.
  - Update at notification of specified events
    - synchronized with explicit clock

```
event Clk1, Clk2;                // system clock
buffered[Clk1] bit[32] R1;       // register
buffered[Clk1] bit[32] R2;

buffered[CLK2] bit[16] RF[64];   // register file
buffered[CLK2] bit[ 8] M1[1024]; // memory

R1 = R2;      // swap contents of R1 and R2
R2 = R1;
wait CLK1;

RF[2] = RF[0] + RF[1];
...
wait CLK2;
```


The SpecC Language

- **`signal`** type modifier
  - Representation of wires and busses in RTL models
  - Semantics as in VHDL, Verilog

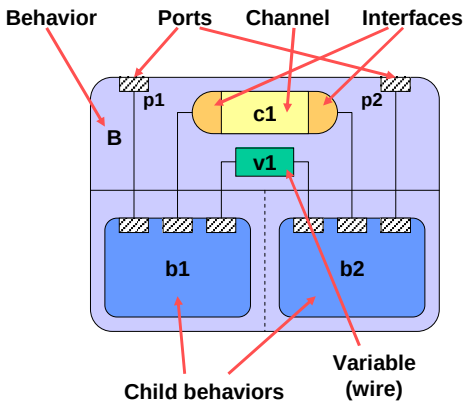
```
signal bit[31:0] addr;   // address bus
signal bit[31:0] data;   // data bus
buffered[CLK] M[1024];

wait addr;               // memory read access
data = M[addr];
...

wait addr && data;
M[addr] = data;          // memory write access
...
```

  - Implemented as buffered variables with associated event

```
signal int x;    ⇔  buffered int x_v; event x_e;
x = 55;          ⇔  x_v = 55; notify x_e;
y = x + 2;       ⇔  y = x_v + 2;
wait x;          ⇔  wait x_e;
notify x;        ⇔  notify x_e;
wait (x == 5);   ⇔  while(x_v != 5) { wait x_e; }
```

The SpecC Language

- Basic structure
  - Top behavior
  - Child behaviors
  - Channels
  - Interfaces
  - Variables (wires)
  - Ports

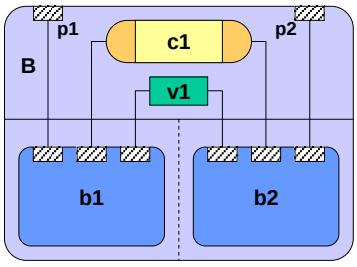


EECS298: Embedded Software Synthesis, Lecture 5 (c) 2004 R. Doemer 13

The SpecC Language

- Basic structure

```
interface I1
{
  bit[63:0] Read(void);
  void Write(bit[63:0]);
};

channel C1 implements I1;

behavior B1(in int, I1, out int);

behavior B(in int p1, out int p2)
{
  int v1;
  C1  c1;
  B1  b1(p1, c1, v1),
      b2(v1, c1, p2);

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

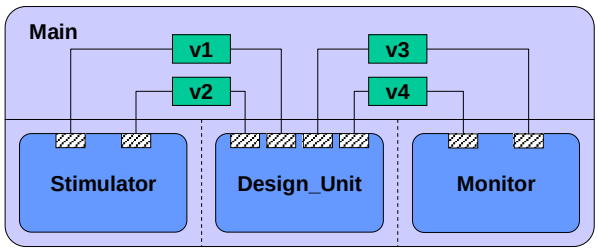


SpecC 2.0:
if b is a behavior instance,
b; is equivalent to b.main();

EECS298: Embedded Software Synthesis, Lecture 5 (c) 2004 R. Doemer 14

The SpecC Language

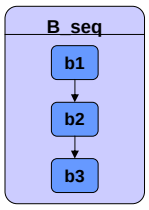
- Typical test bench
  - Top-level behavior: `Main`
  - Stimulator provides test vectors
  - Design unit under test
  - Monitor observes and checks outputs



The SpecC Language

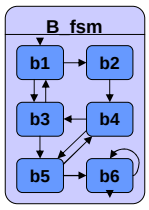
- Behavioral hierarchy

Sequential execution



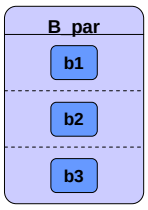

```
behavior B_seq
{
 B b1, b2, b3;

 void main(void)
 {  b1;
    b2;
    b3;
  }
};
```

FSM execution



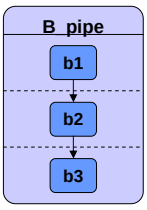

```
behavior B_fsm
{
 B b1, b2, b3,
   b4, b5, b6;
 void main(void)
 { fsm { b1:{…}
         b2:{…}
         …}
  }
};
```

Concurrent execution




```
behavior B_par
{
 B b1, b2, b3;

 void main(void)
 { par{ b1;
        b2;
        b3; }
  }
};
```

Pipelined execution

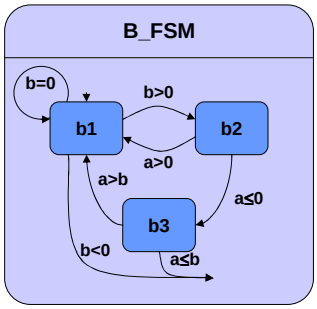



```
behavior B_pipe
{
 B b1, b2, b3;

 void main(void)
 { pipe{ b1;
         b2;
         b3; }
  }
};
```

The SpecC Language

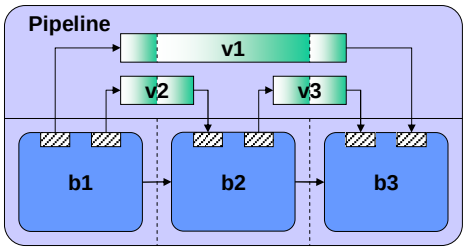
- Finite State Machine (FSM)
  - Explicit state transitions
    - triple < *current_state, condition, next_state* >
    - **fsm** { <*current_state*> : { **if** <*condition*> **goto** <*next_state*> } … }
  - Moore-type FSM
  - Mealy-type FSM

```
behavior B_FSM(in int a, in int b)
{
 B b1, b2, b3;

 void main(void)
 { fsm { b1:{ if (b<0) break;
              if (b==0) goto b1;
              if (b>0) goto b2; }
         b2:{ if (a>0) goto b1; }
         b3:{ if (a>b) goto b1; }
       }
 }
};
```



The SpecC Language

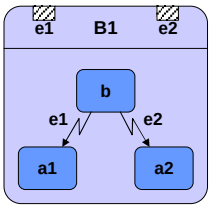
- Pipeline
  - Explicit execution in pipeline fashion
    - **pipe** { <*instance_list*> };
    - **pipe** (<*init*>; <*cond*>; <*incr*>) { … }
  - Support for automatic buffering
    - **piped** […] <*type*> <*variable_list*>;

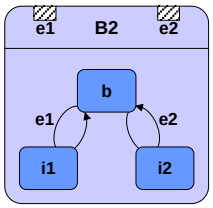



```
behavior Pipeline
{
 piped piped int v1;
 piped int v2;
 piped int v3;

 Stage1 b1(v1, v2);
 Stage2 b2(v2, v3);
 Stage3 b3(v3, v1);

 void main(void)
 {
   int i;
   pipe(i=0; i<10; i++)
       { b1;
         b2;
         b3;
        }
  }
};
```

The SpecC Language

- Exception handling
  - Abortion
  - Interrupt



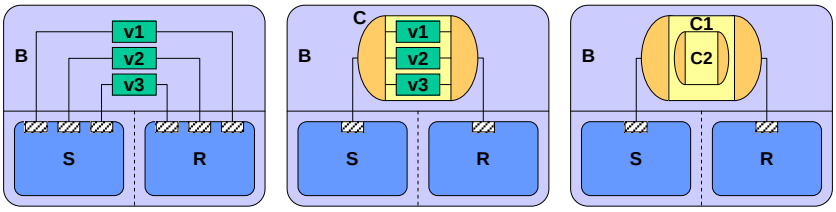



```
behavior B1(in event e1, in event e2)
{
 B b, a1, a2;

 void main(void)
 { try { b; }
   trap (e1) { a1; }
   trap (e2) { a2; }
  }
};
```

```
behavior B2(in event e1, in event e2)
{
 B b, i1, i2;

 void main(void)
 { try { b; }
   interrupt (e1) { i1; }
   interrupt (e2) { i2; }
  }
};
```


The SpecC Language

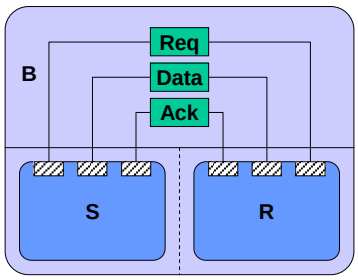
- Communication
  - via shared variable
  - via virtual channel
  - via hierarchical channel



Shared memory **Message passing** **Protocol stack**

The SpecC Language

- Synchronization
  - Event type
    - **event** <*event_List*>;
  - Synchronization primitives
    - **wait** <*event_list*>;
    - **notify** <*event_list*>;
    - **notifyone** <*event_list*>;

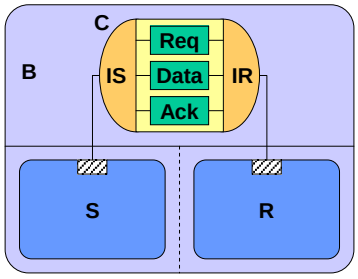



```
behavior S(out event Req,
           out float Data,
           in  event Ack)
{
 float X;
 void main(void)
 { ...
   Data = X;
   notify Req;
   wait Ack;
   ...
 }
};
```

```
behavior R(in  event Req,
           in  float Data,
           out event Ack)
{
 float Y;
 void main(void)
 { ...
   wait Req;
   Y = Data;
   notify Ack;
   ...
 }
};
```


The SpecC Language

- Communication
  - Interface class
    - **interface** <*name*> { <*declarations*> };
  - Channel class
    - **channel** <*name*> **implements** <*interfaces*> { <*implementations*> };




```
interface IS
{
 void Send(float);
};
interface IR
{
 float Receive(void);
};
```

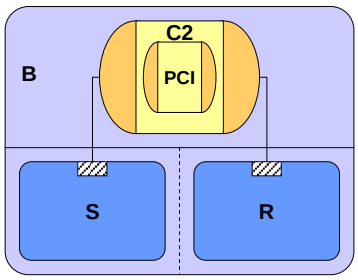
```
behavior S(IS Port)
{
 float X;
 void main(void)
 { ...
   Port.Send(X);
   ...
 }
};
```

```
behavior R(IR Port)
{
 float Y;
 void main(void)
 {...
  Y=Port.Receive();
  ...
 }
};
```

```
channel C
    implements IS, IR
{
 event Req;
 float Data;
 event Ack;

 void Send(float X)
 { Data = X;
   notify Req;
   wait Ack;
 }
 float Receive(void)
 { float Y;
   wait Req;
   Y = Data;
   notify Ack;
   return Y;
 }
};
```

The SpecC Language

- Hierarchical channel
  - Virtual channel implemented by standard bus protocol
    - example: PCI bus




```
interface PCI_IF
{
 void Transfer(
      enum Mode,
      int NumBytes,
      int Address);
};
```

```
behavior S(IS Port)
{
 float X;
 void main(void)
 { ...
   Port.Send(X);
   ...
 }
};
```

```
behavior R(IR Port)
{
 float Y;
 void main(void)
 {...
  Y=Port.Receive();
  ...
 }
};
```

```
interface IS
{
 void Send(float);
};
interface IR
{
 float Receive(void);
};
```

```
channel PCI
   implements PCI_IF;

channel C2
   implements IS, IR
{
 PCI Bus;
 void Send(float X)
 { Bus.Transfer(
       PCI_WRITE,
       sizeof(X),&X);
 }
 float Receive(void)
 { float Y;
   Bus.Transfer(
       PCI_READ,
       sizeof(Y),&Y);
   return Y;
 }
};
```

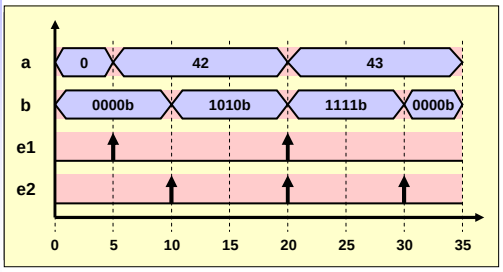

The SpecC Language

- SpecC Standard Channel Library
  - introduced with SpecC Language Version 2.0
  - includes support for
    - mutex
    - semaphore
    - critical section
    - barrier
    - token
    - queue
    - handshake
    - double handshake
    - …

The SpecC Language

- Timing
  - Exact timing
    - **waitfor** <*delay*>;

Example: stimulator for a test bench

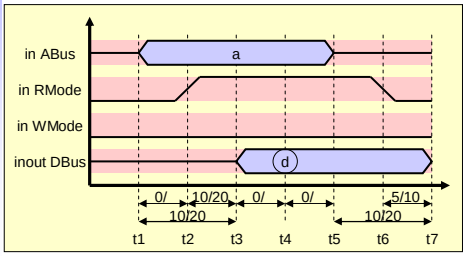



```
behavior Testbench_Driver
         (inout int a,
          inout int b,
          out event e1,
          out event e2)
{
  void main(void)
  {
   waitfor 5;
   a = 42;
   notify e1;

   waitfor 5;
   b = 1010b;
   notify e2;

   waitfor 10;
   a++;
   b |= 0101b;
   notify e1, e2;

   waitfor 10;
   b = 0;
   notify e2;
  }
};
```


The SpecC Language

- Timing
  - Exact timing
    - **waitfor** <*delay*>;
  - Timing constraints
    - **do** { <*actions*> }
      timing {<*constraints*>}

Example: SRAM read protocol



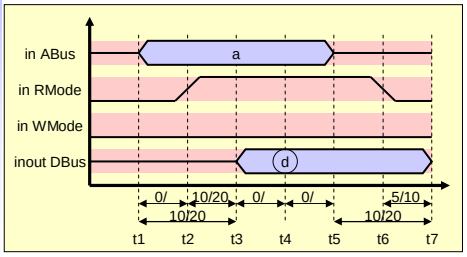
Specification

```
bit[7:0] Read_SRAM(bit[15:0] a)
{
 bit[7:0] d;

 do { t1: {ABus = a;  }
      t2: {RMode = 1;
           WMode = 0; }
      t3: {           }
      t4: {d = Dbus;  }
      t5: {ABus = 0;  }
      t6: {RMode = 0;
           WMode = 0; }
      t7: { }
     }
  timing { range(t1; t2;  0;   );
           range(t1; t3; 10; 20);
           range(t2; t3; 10; 20);
           range(t3; t4;  0;   );
           range(t4; t5;  0;   );
           range(t5; t7; 10; 20);
           range(t6; t7;  5; 10);
          }
  return(d);
}
```

The SpecC Language

- Timing
  - Exact timing
    - **waitfor** <*delay*>;
  - Timing constraints
    - **do** { <*actions*> }
      timing {<*constraints*>}

Example: SRAM read protocol



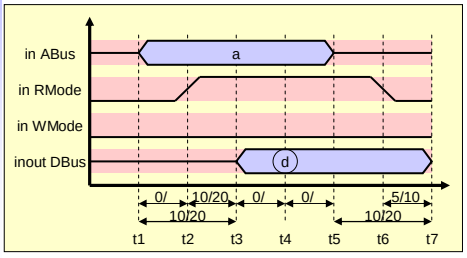
Implementation 1

```
bit[7:0] Read_SRAM(bit[15:0] a)
{
 bit[7:0] d;

 do { t1: {ABus = a;  waitfor( 2);}
      t2: {RMode = 1;
           WMode = 0; waitfor(12);}
      t3: {           waitfor( 5);}
      t4: {d = Dbus;  waitfor( 5);}
      t5: {ABus = 0;  waitfor( 2);}
      t6: {RMode = 0;
           WMode = 0; waitfor(10);}
      t7: { }
     }
  timing { range(t1; t2;  0;   );
           range(t1; t3; 10; 20);
           range(t2; t3; 10; 20);
           range(t3; t4;  0;   );
           range(t4; t5;  0;   );
           range(t5; t7; 10; 20);
           range(t6; t7;  5; 10);
          }
  return(d);
}
```


The SpecC Language

- Timing
  - Exact timing
    - **waitfor** <*delay*>;
  - Timing constraints
    - **do** { <*actions*> }
      timing {<*constraints*>}

Example: SRAM read protocol



Implementation 2

```
bit[7:0] Read_SRAM(bit[15:0] a)
{
 bit[7:0] d;       // ASAP Schedule

 do { t1: {ABus = a;  }
      t2: {RMode = 1;
           WMode = 0; waitfor(10);}
      t3: {           }
      t4: {d = Dbus;  }
      t5: {ABus = 0;  }
      t6: {RMode = 0;
           WMode = 0; waitfor(10);}
      t7: { }
     }
  timing { range(t1; t2;  0;   );
           range(t1; t3; 10; 20);
           range(t2; t3; 10; 20);
           range(t3; t4;  0;   );
           range(t4; t5;  0;   );
           range(t5; t7; 10; 20);
           range(t6; t7;  5; 10);
          }
  return(d);
}
```

The SpecC Language

- RTL Modeling
  - Accellera RTL Semantics Standard
    - Style 1: *unmapped*
      - `a = b * c`
    - Style 2: *storage mapped*
      - `R1 = R1 * RF2[4]`
    - Style 3: *function mapped*
      - `R1 = ALU1(MULT, R1, RF2[4])`
    - Style 4: *connection mapped*
      - `Bus1=R1; Bus2=RF2[4]; Bus3=ALU1(MULT, Bus1, Bus2)`
    - Style 5: *exposed control*
      - `ALU_CTRL = 011001; RF2_CTRL = 010; …`

    Source: http://www.eda.org/alc-cwg/cwg-open.pdf
  - Types specific for RTL:
    - **`buffered`** type modifier: Storage
    - **`signal`** type modifier: Communication
  - Control flow specific for RTL:
    - **`fsmd`** construct: Explicit finite state machine with datapath

The SpecC Language

- RTL Modeling
  - **`fsmd`** construct
    - Similar to **`fsm`** construct, but specifically for RTL
    - Explicit states and state transitions
    - State actions represent well-defined register transfers
      - limited to conditional/unconditional assignments and function calls
      - general loops, exceptions, synchronization, timing are not allowed
    - Explicit clock specifier
      - event list (external clock)
      - time delay (internal clock)
    - Explicit sensitivity list
      - needed for Mealy machine support
    - Explicit reset state
      - synchronous reset
      - asynchronous reset

The SpecC Language

RTL Modeling Example

```
behavior FSMD_Example(
        signal in  bool       CLK,         // system clock
        signal in  bool       RST,         // system reset
        signal in  bit[31:0]  Inport,      // input ports
        signal in  bit[1]     Start,
        signal out bit[31:0]  Outport,     // output ports
        signal out bit[1]     Done)
{  void main(void)
   {
     fsmd(CLK)                             // clock + sensitivity
        {
          bit[32] a, b, c, d, e;           // local variables

                      { Outport = 0;       // default
                        Done = 0b;         // assignments
                       }
          if (RST)    { goto S0;           // reset actions
                       }
          S0 :        { if (Start) goto S1;
                        else       goto S0;
                       }
          S1 :        { a = b + c;         // state actions
                        d = Inport * e;    // (register transfers)
                        Outport = a;
                        goto S2;
                       }
          ... }
     }
};
```


The SpecC Language

- Library support
  - Import of precompiled SpecC code
    - **import** <*component_name*>;
  - Automatic handling of multiple inclusion
    - no need to use **`#ifdef`** - **`#endif`** around included files
  - Visible to the compiler/synthesizer
    - not inline-expanded by preprocessor
    - simplifies reuse of IP components

```
// MyDesign.sc

#include <stdio.h>
#include <stdlib.h>

import "Interfaces/I1";
import "Channels/PCI_Bus";
import "Components/MPEG-2";

...
```

The SpecC Language

- Persistent annotation
  - Attachment of a key-value pair
    - globally to the design, i.e. **note** <*key*> = <*value*>;
    - locally to any symbol, i.e. **note** <*symbol*>.<*key*> = <*value*>;
  - Visible to the compiler/synthesizer
    - eliminates need for pragmas
    - allows easy data exchange among tools

SpecC 2.0: <*value*> can be a composite constant (just like complex variable initializers)

```
/* comment, not persistent */

// global annotations
note Author = "Rainer Doemer";
note Date   = "Fri Feb 23 23:59:59 PST 2001";

behavior CPU(in event CLK, in event RST, ...)
{
  // local annotations
  note MinMaxClockFreq = {750*1e6, 800*1e6 };
  note CLK.IsSystemClock = true;
  note RST.IsSystemReset = true;
  ...
};
```

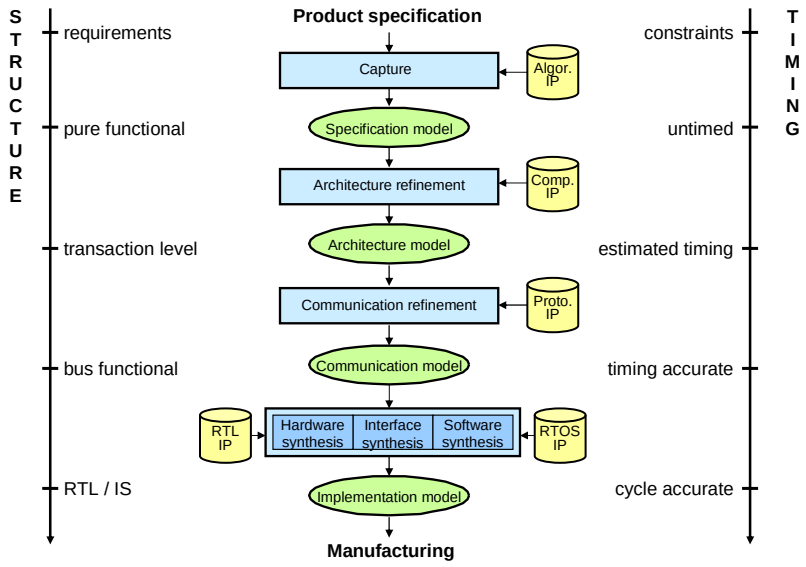

SpecC Summary

- SpecC model
  - Hierarchical network of behaviors and channels
  - Separation of communication and computation
- SpecC language
  - True superset of ANSI-C
    - ANSI-C plus extensions for HW-design
  - Support of all concepts needed in system design
    - Structural and behavioral hierarchy
    - Concurrency
    - State transitions
    - Communication
    - Synchronization
    - Exception handling
    - Timing
    - RTL

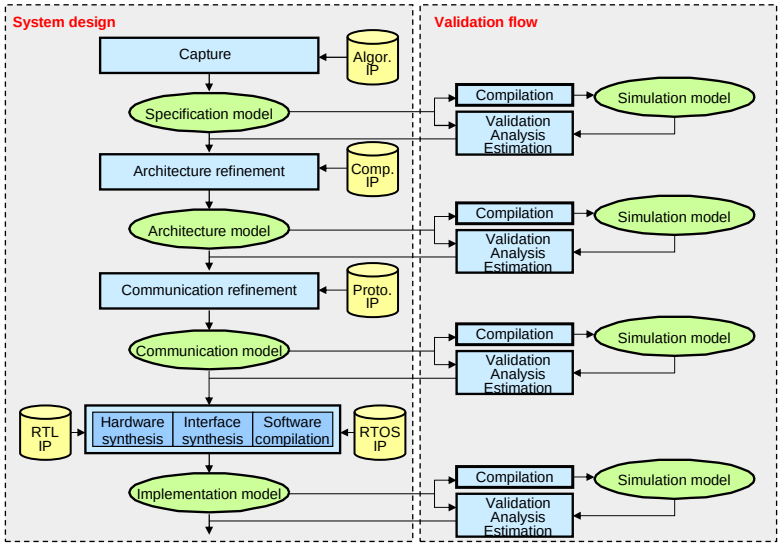
Embedded System Design with SpecC

- The SpecC Language
- Embedded System Design Flow
- System-on-Chip Environment (SCE)
  - Demo

Embedded System Design Flow



Design Methodology



Refinement Decisions

- 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 in channels

Refinement Decisions

- 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, i.e. Round-robin or priority-based
    - For each set of concurrent behaviors, determine scheduling priority
  - Result:
    System model with abstract RTOS scheduler inserted in each PE

Refinement Decisions

- Step 3: 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:
    Bus-functional model of the system

Refinement Decisions

- Step 4: Hardware Refinement (for HW PE)
  - Allocation of RTL components
    - Type and number of functional units
    - Type and number of storage units
    - Type and number of interconnecting busses
  - Scheduling
    - Basic blocks assigned to super-states
    - Operations assigned to clock cycles
  - Binding
    - Bind functional operations to functional units
    - Bind variables to storage units
    - Bind transfers to busses
  - Result:
    Clock-cycle accurate model of each HW PE
  - Output: Synthesizable Verilog description

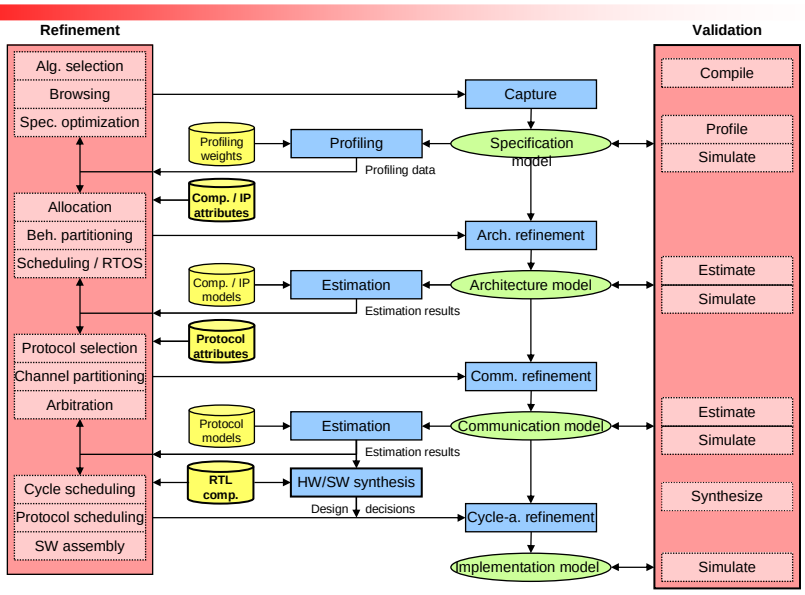
Refinement Decisions

- Step 5: Software Refinement (for SW PE)
  - C code generation
    - For selected target processor
    - For selected target RTOS
  - Compilation to Instruction Set
    - for Instruction Set Simulation (ISS)
  - Assembly
  - Result:
    Clock-cycle accurate model of each SW PE
  - Output: downloadable object code

Embedded System Design with SpecC

- The SpecC Language
- Embedded System Design Flow
- System-on-Chip Environment (SCE)
  - Demo

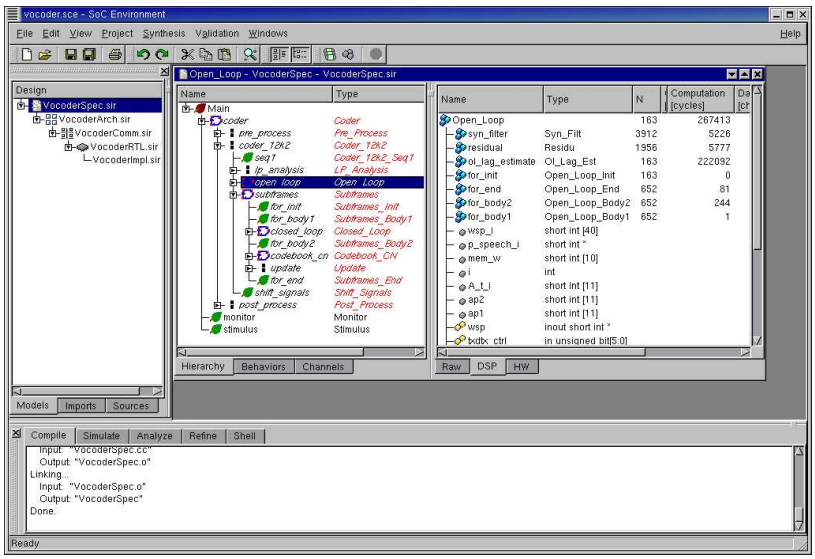
System-on-Chip Environment (SCE)



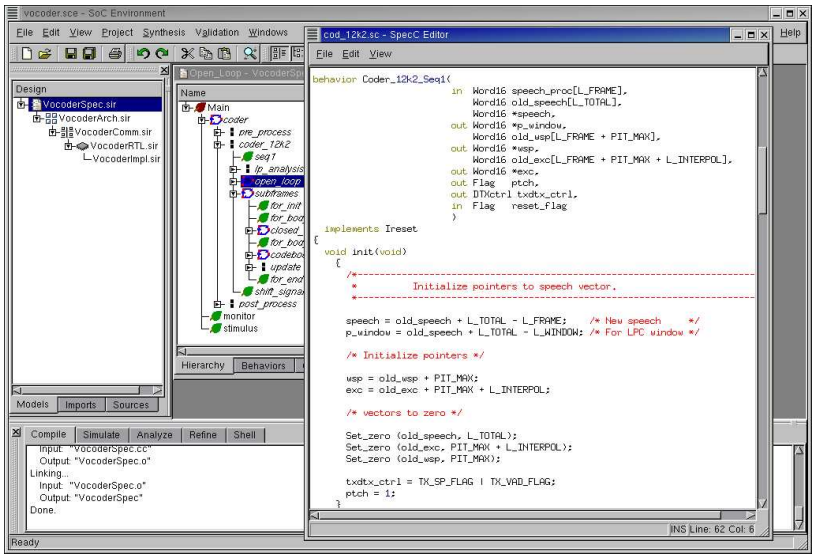
System-on-Chip Environment (SCE)

- SCE Components:
  - Graphical frontend (`sce, scchart`)
  - 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 ...

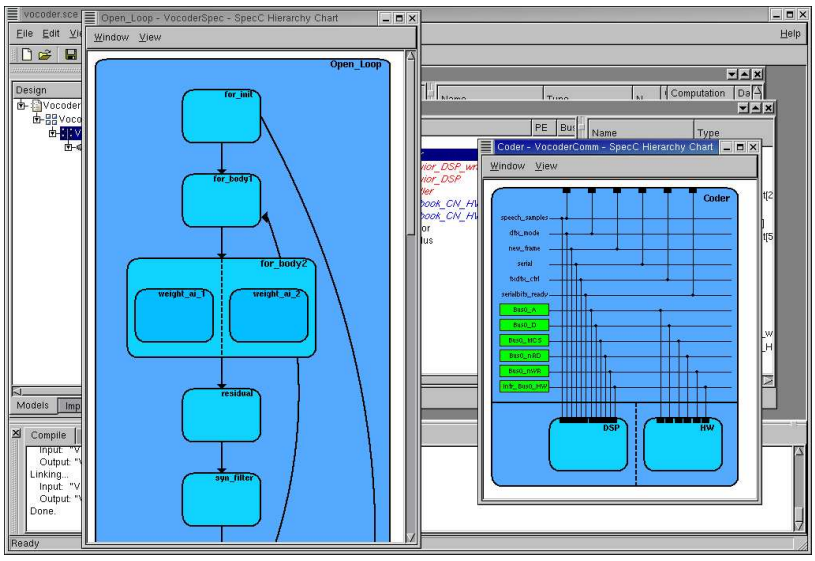
SCE Main Window



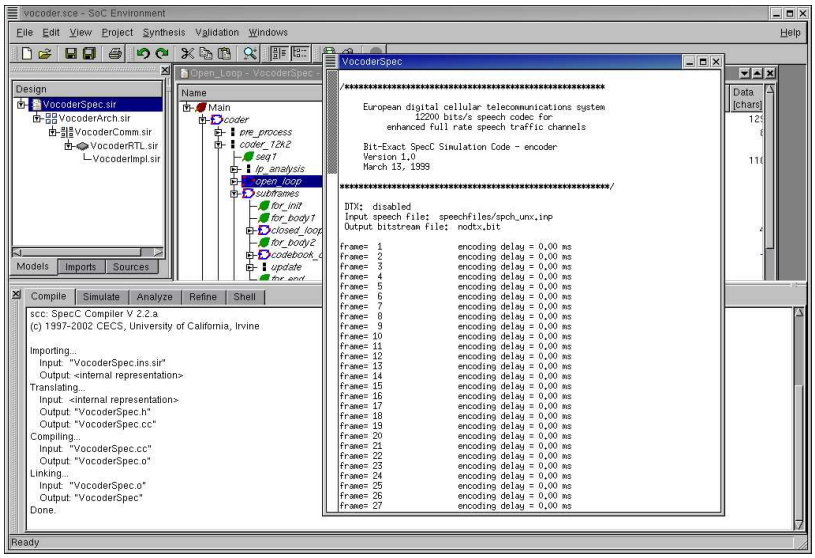
SCE Source Editor



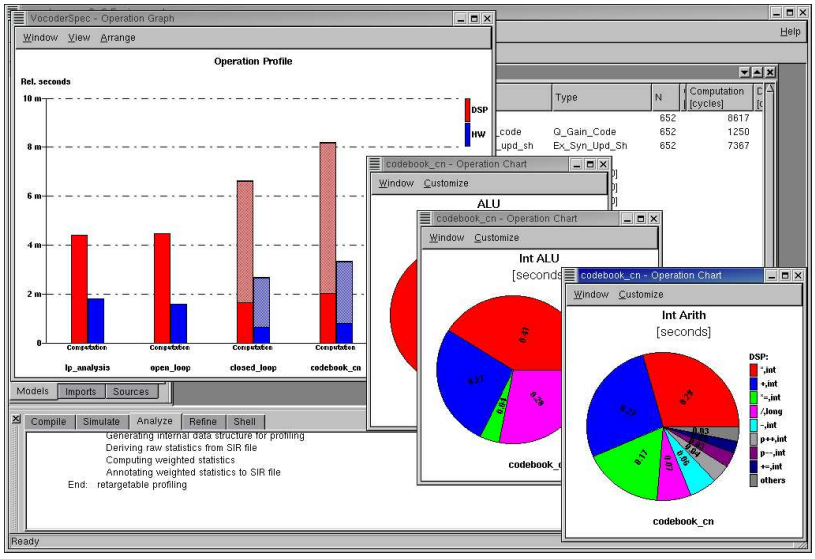
SCE Hierarchy Displays



SCE Compiler and Simulator



SCE Profiling and Analysis



Specification Example

- 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)