Lecture 2 – System-Level Design Languages

Sources:
R. Doemer, UC Irvine
M. Radetzki, Univ. of Stuttgart

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

Lecture 2: Outline

- System design languages
  - Goals, requirements
  - Separation of computation & communication

- The SpecC language
  - Core language syntax and semantics
    - Comparison with SystemC
  - Channel library
  - Compiler and simulator

- The SystemC language
  - Syntax and semantics
Languages

- **Represent a model in machine-readable form**
  - Apply algorithms and tools

- **Syntax defines grammar**
  - Possible strings over an alphabet
  - Textual or graphical

- **Semantics defines meaning**
  - Mapping onto an abstract state machine model
    - Operational semantics
  - Mapping into a mathematical domain (e.g. functions)
    - Denotational semantics

Models vs. Languages

- **Computation models describe system behavior**
  - Conceptual notion, e.g., recipe, sequential program

- **Languages capture models**
  - Concrete form, e.g., English, C

- **Variety of languages can capture one model**
  - E.g., sequential program model → C, C++, Java

- **One language can capture variety of models**
  - E.g., C++ → sequential program model, object-oriented model, state machine model

- **Certain languages better at capturing certain models**

Text vs. Graphics

- Models versus languages not to be confused with text versus graphics
  - Text and graphics are just two types of languages
    - Text: letters, numbers
    - Graphics: circles, arrows (plus some letters, numbers)

Simulation vs. Synthesis

- Ambiguous semantics of languages
  - Simulatable but not synthesizable or verifiable
    - Impossible to automatically discern implicit meaning
    - Need explicit set of constructs


Source: D. Gajski, UC Irvine
Programming Models

- **Imperative programming models**
  - Statements that manipulate program state, control flow
    - Sequential programming languages [C, C++, …]
      - Van Neuman semantics
- **Declarative programming models**
  - Rules for data manipulation, data flow
    - Functional programming [Haskell, Lisp, Excel]
    - Logic programming [Prolog]

- Sequential behavior at processor level
  - Granularity of arithmetic/logic expressions over variables
  - Implicit or explicit operation-level parallelism
  - No coarser-grain concurrency or time

Evolution of Design Languages

- **Netlists**
  - Structure only: components and connectivity
    - Gate-level [EDIF], system-level [SPIRIT/XML]
- **Hardware description languages (HDLs)**
  - Event-driven behavior: signals/wires, clocks
  - Register-transfer level (RTL): boolean logic
    - Discrete event [VHDL, Verilog]
- **System-level design languages (SLDLs)**
  - Software behavior: sequential functionality/programs
    - C-based, event-driven [SpecC, SystemC, SystemVerilog]

- Structural descriptions at varying levels
  - Structural concurrency, time
  - Behavioral concurrency at higher levels?
System-Level Design Languages (SLDLs)

- **Goals**
  - Executability
    - Validation through simulation
  - Synthesizability
    - Implementation in HW and/or SW
    - Support for IP reuse
  - Modularity
    - Hierarchical composition
    - Separation of concepts
  - Completeness
    - Support for all concepts found in embedded systems
  - Orthogonality
    - Orthogonal constructs for orthogonal concepts
    - Minimality
  - Simplicity

Source: R. Doemer, UC Irvine

System-Level Design Languages (SLDLs)

- **Requirements**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>C++</th>
<th>Java</th>
<th>VHDL</th>
<th>Verilog</th>
<th>SystemC</th>
<th>SpecCharts</th>
<th>SpecC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavioral hierarchy</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Structural hierarchy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concurrency</td>
<td>○</td>
<td>○</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Synchronization</td>
<td>○</td>
<td>○</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Exception handling</td>
<td>○</td>
<td>○</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Timing</td>
<td>○</td>
<td>○</td>
<td></td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>State transitions</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Composite data types</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

○ not supported  ○ partially supported  ● supported

Source: R. Doemer, UC Irvine
System-Level Design Languages (SLDLs)

- **C/C++**
  - ANSI standard programming languages, software design
  - Traditionally used for system design because of practicality, availability
- **SystemC**
  - C++ API and class library
  - Initially developed at UC Irvine, standard by Open SystemC Initiative (OSCI)
- **SpecC**
  - C extension
  - Developed at UC Irvine, standard by SpecC Technology Open Consortium (STOC)
- **SystemVerilog**
  - Verilog with C extensions for testbench development
- **Matlab/Simulink**
  - Specification and simulation in engineering, algorithm design
- **Unified Modeling Language (UML)**
  - Software specification, graphical, extensible (meta-modeling)
  - Modeling and Analysis of Real-time and Embedded systems (MARTE) profile
- **IP-XACT**
  - XML schema for IP component documentation, standard by SPIRIT consortium
- **Rosetta (formerly SLDL)**
  - Formal specification of constraints, requirements
- **SDL**
  - Telecommunication area, standard by ITU
- ... 

Source: R. Doemer, UC Irvine

Separation of Concerns

- **Fundamental principle in modeling of systems**
- **Clear separation of concerns**
  - Address separate issues independently
- **System-Level Description Language (SLDL)**
  - Orthogonal concepts
  - Orthogonal constructs
- **System-level Modeling**
  - Computation
    - encapsulated in modules / behaviors
  - Communication
    - encapsulated in channels

Source: R. Doemer, UC Irvine
Computation vs. Communication

- **Traditional model**
  - Processes and signals
  - Mixture of computation and communication
  - Automatic replacement impossible

- **SpecC model**
  - Behaviors and channels
  - Separation of computation and communication
  - Plug-and-play

Source: R. Doemer, UC Irvine

---

Computation vs. Communication

- **Protocol Inlining**
  - Specification model
  - Exploration model
    - Computation in behaviors
    - Communication in channels

- **Implementation model**
  - Channel disappears
  - Communication inlined into behaviors
  - Wires exposed

Source: R. Doemer, UC Irvine
Lecture 2: Outline

- System design languages
  - Goals, requirements
  - Separation of computation & communication

- The SpecC language
  - Core language syntax and semantics
    - Comparison with SystemC
  - Channel library
  - Compiler and simulator

- The SystemC language
  - Syntax and semantics

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
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

---

SpecC vs. SystemC

- **SpecC language**
  - ANSI C
  - New keywords
  - SpecC compiler (scc)

- **SpecC simulator**
  - Discrete event C++ simulation kernel

- **SpecC standardization**
  - SpecC Technology Open Consortium (STOC)
  - Open source reference compiler and simulator

- **SystemC “language”**
  - C++
  - Class library
  - Standard C++ tools (g++)

- **SystemC simulator**
  - Discrete event C++ simulation kernel

- **SystemC standardization**
  - Open SystemC Initiative (OSCI)
  - Open-source library and simulation kernel
  - IEEE Standard
The SpecC Language

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

```c
/* 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()`

```c
#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
    - signal bit[16] address;

- **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]
The SystemC Language

- **SpecC data types**
  - C types & boolean
  - Bit vectors
  - 4-value logic vectors
  - Events
  - Signals*

- **SystemC data types**
  - C++ types
  - Bit vectors
  - 4-value logic vectors
  - Events*
  - Signal channel*
  - Fixed-point
  - Variable-length integers

*SpecC 2.0

The SpecC Language

- **Basic structure**
  - Top behavior
  - Child behaviors
  - Channels
  - Interfaces
  - Variables (wires)
  - Ports
The SpecC Language

- Basic structure

```c
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();`

- Behavioral hierarchy

Sequential execution

```c
behavior B_seq{
    B b1, b2, b3;
    void main(void) {
        b1; b2; b3;
    }
};
```

FSM execution

```c
behavior B_fsm{
    B b1, b2, b3, b4, b5, b6;
    void main(void) {
        fsm { b1:{...} b2:{...} b3:{...} b4:{...} b5:{...} b6:{...} }
    }
};
```
The SpecC Language

- **Behavioral hierarchy**
  - [Sequential execution](#)
    - \( B \_\text{seq} \)
      - \( b_1 \)
      - \( b_2 \)
      - \( b_3 \)
  - [FSM execution](#)
    - \( B \_\text{fsm} \)
      - \( b_1 \)
      - \( b_2 \)
      - \( b_3 \)
      - \( b_4 \)
      - \( b_5 \)
      - \( b_6 \)
  - [Concurrent execution](#)
    - \( B \_\text{par} \)
      - \( b_1 \)
      - \( b_2 \)
      - \( b_3 \)
  - [Pipelined execution](#)
    - \( B \_\text{pipe} \)
      - \( b_1 \)
      - \( b_2 \)
      - \( b_3 \)

```spec
definition B_seq{
  B b1, b2, b3;
  void main() {
    b1;
    b2;
    b3;
  }
};
```

```spec
definition B_fsm{
  B b1, b2, b3, b4, b5, b6;
  void main() {
    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;
    }
  }
};
```

```spec
definition B_par{
  B b1, b2, b3;
  void main() {
    par { b1;
      b2;
      b3; }
  }
};
```

```spec
definition B_pipe{
  B b1, b2, b3;
  void main() {
    pipe { b1;
      b2;
      b3; }
  }
};
```

The SpecC Language

- **Finite State Machine (FSM)**
  - Explicit state transitions
    - triple \(<\text{current\_state}, \text{condition}, \text{next\_state}>\)
    - \( \text{fsm} \{ <\text{current\_state}> : \{ \text{if } <\text{condition}> \text{ goto } <\text{next\_state}> \} \ldots \} \)
  - Moore-type FSM
  - Mealy-type FSM

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

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

(c) 2014 A. Gerstlauer
The SpecC Language

- **Pipeline**
  - Explicit execution in pipeline fashion
    - \texttt{pipe} \{ <instance\_list> \};
    - \texttt{pipe} (\texttt{init}; <cond>; <incr>) \{ ... \}

```
behavior Pipeline {
    Stage1 b1;
    Stage2 b2;
    Stage3 b3;
    void main(void) {
        pipe (b1;
             b2;
             b3);
    }
    
}
```

```
behavior Pipeline {
    Stage1 b1;
    Stage2 b2;
    Stage3 b3;
    void main(void) {
        int i;
        pipe(i=0; i<10; i++) {
            b1;
            b2;
            b3;
        }
    }
    
}
```
The SpecC Language

- **Pipeline**
  - Explicit execution in pipeline fashion
    - `pipe ( <instance_list> );`
    - `pipe (<init>; <cond>; <incr>) { ... }`
  - Support for automatic buffering

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

- **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

Shared memory

The SpecC Language

- Communication
  - via shared variable
  - via virtual channel

Shared memory

Message passing
The SpecC Language

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

![Diagram of communication modes: shared memory, message passing, and protocol stack]

**Synchronization**

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

```c
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;
        ...
    }
}
```

![Diagram of synchronization example]
The SpecC Language

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

```
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;
  }
}
```

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

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();
    ...
  }
}
```

```
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.Receive(PCI_READ, sizeof(Y), &Y);
    return Y;
  }
}
```

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

(c) 2014 A. Gerstlauer
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;
    }
  }
```

Example: SRAM read protocol

```
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);
  }
```

Specification
The SpecC Language

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

**Example: SRAM read protocol**

```specc
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: { } 
         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);
}
```

**Implementation 1**

**Implementation 2**
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

```c
// MyDesign.sc
#include <stdio.h>
#include <stdlib.h>
import "Interfaces/I1";
import "Channels/PCI_Bus";
import "Components/MPEG-2";
...
```

- **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
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

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

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
    – ...

➤ Examples under
  ➤ `$SPECC/examples/sync`
SpecC vs. SystemC

- **SpecC channels***
  - c_queue / c_typed_queue
  - c_double_handshake / c_typed_double_handshake
  - c_handshake
  - c_mutex
  - c_semaphore
  - c_token
  - c_barrier
  - c_critical_section

- **SystemC channels***
  - sc_fifo<T>
  - sc_event_queue
  - sc_mutex
  - sc_semaphore
  - sc_signal<T>
  - sc_buffer<T>
  - sc_clock

*SpecC 2.0  SystemC 2.0

The SpecC Language

- **SpecC Standard Channel Library**
  - mutex channel
  - semaphore channel

```c
interface i_semaphore
{
    void acquire(void);
    void release(void);
    bool attempt(void);
};

channel c_mutex
    implements i_semaphore;

channel c_semaphore(
    in const unsigned long c)
    implements i_semaphore;
```
The SpecC Language

- **SpecC Standard Channel Library**
  - mutex channel
  - semaphore channel
  - critical section

```
interface i_critical_section
{
    void enter(void);
    void leave(void);
};
```

```
channel c_critical_section
implements i_critical_section;
```

The SpecC Language

- **SpecC Standard Channel Library**
  - mutex channel
  - semaphore channel
  - critical section
  - barrier

```
interface i_barrier
{
    void barrier(void);
};
```

```
channel c_barrier(
    in unsigned long n)
implements i_barrier;
```
The SpecC Language

• SpecC Standard Channel Library
  • mutex channel
  • semaphore channel
  • critical section
  • barrier
  • token

interface i_consumer
{
  void consume(unsigned long n);
}

interface i_producer
{
  void produce(unsigned long n);
}

channel c_token
  implements i_consumer,
         i_producer,
         i_token;

interface i_tranceiver
{
  void receive(void *d, unsigned long l);
  void send(void *d, unsigned long l);
}

channel c_queue(
in const unsigned long s)
  implements i_receiver,
         i_sender,
         i_tranceiver;
The SpecC Language

- SpecC Standard Channel Library
  - mutex channel
  - semaphore channel
  - critical section
  - barrier
  - token
  - queue
  - double handshake

```
interface i_tranceiver
{ void receive(void *d, unsigned long l);
  void send(void *d, unsigned long l);
};

interface i_receiver
{ void receive(void *d, unsigned long l);
};

interface i_sender
{ void send(void *d, unsigned long l);
};

channel c_double_handshake
implements i_receiver, i_sender, i_tranceiver;
```

```
interface i_receive
{ void receive(void); }

interface i_send
{ void send(void); }

channel c_handshake
implements i_receive, i_send;
```
SpecC Standard Channels

- **Importing Channels (from $SPECC/import/)**
  - Synchronization channels
    - mutex channel  
      ```
      import "c_mutex";
      ```
    - semaphore channel  
      ```
      import "c_semaphore";
      ```
    - critical section  
      ```
      import "c_critical_section";
      ```
    - barrier  
      ```
      import "c_barrier";
      ```
    - handshake  
      ```
      import "c_handshake";
      ```
    - token  
      ```
      import "c_token";
      ```
  - Communication channels (typeless)
    - queue  
      ```
      import "c_queue";
      ```
    - double handshake  
      ```
      import "c_double_handshake";
      ```

- **Including Typed Channels (from $SPECC/inc/)**
  - Communication channels (typed)
    - queue  
      ```
      #include <c_typed_queue.sh>
      ```
    - double handshake  
      ```
      #include <c_typed_double_handshake.sh>
      ```
  - Example:
    ```
    #include <c_typed_double_handshake.sh>
    
    struct pack { int a, b, c; };
    DEFINE_I_TYPED_SENDER(pack, struct pack)
    DEFINE_I_TYPED_RECEIVER(pack, struct pack)
    DEFINE_C_TYPED_DOUBLE_HANDSHAKE(pack, struct pack)
    
    behavior Sender(i_pack_sender Port)
    {
      void main(void)
      {
        struct pack Data = { 1, 2, 3 }; // ...
        Port.send(Data);
        // ...
      };
    }
    ```
SpecC Tools

- **Server and accounts**
  - ECE LRC Linux servers
  - SpecC software (© by CECS, UCI)
    - `/usr/local/packages/sce-20100908`
    - `module load sce`

- **SpecC compiler package**
  - Compiler and simulator
    - `scc <design> <command> <options>`
      - Commands: `-sc2sir/-sir2out/-sc2out/...`
      - Usage/help: `scc -h/man scc`
  - Design manipulation tools
    - `sir_rename / sir_delete / sir_import`
    - `sir_list / sir_tree`
    - `sir_note`
    - ...

SpecC Compiler (scc)

≈ SystemC
System Validation using SpecC

- **SpecC Simulator**
  - Execution as regular program
  - Example:
    ```
    % ./HelloWorld
    Hello World!
    ```
- **Simulation library**
  - Access via inclusion of SpecC header files
  - Example: Print the current simulation time
    ```
    #include <sim.sh>
    ...
    sim_time t;
    sim_delta d;
    sim_time_string buffer;
    ...
    t = now();  d = delta();
    printf("Time is now %s pico seconds.\n", time2str(buffer, t));
    printf("(delta count is %s)\n", time2str(buffer, d));
    waitfor 10 NANO_SEC;
    printf("Time is now %s pico seconds.\n", time2str(buffer, t));
    ...
    ```

System Validation using SpecC

- **Simulation**
  - `scc DesignName -sc2out -vv -ww
  .DesignName`
  - Header file `sim.sh`
    - Access to simulation time
      - Macros `PICO_SEC`, `NANO_SEC`, `MICRO_SEC`, `MILLI_SEC`, `SEC`
      - Typedef: `sim_time, sim_delta, sim_time_string`
      - Functions `now(), delta()`
      - Conversion functions `time2str(), str2time()`
    - Handling of bit vectors
      - Conversion functions `bit2str(), ubit2str(), str2bit(), str2ubit()`
    - Handling of long-long values
      - Conversion functions `ll2str(), ull2str(), str2ll(), str2ull()`
System Validation using SpecC

**Debugging**
- `module load gnutools` (for ddd)
- `scc DesignName -sc2out -vv -ww -g -G
gdb ./DesignName` (interactive debugger)
ddd ./DesignName (graphical gdb frontend)
- Header file `sim.sh`
  - Access to simulation engine state
    » functions `ready_queue()`, `running_queue()`, etc.
    » functions `_print_ready_queue()`, `_print_running_queue()`, etc.
    » function `_print_process_states()`
    » function `_print_simulator_state()`
  - Access to current instance
    » functions `active_class()`, `active_instance()`
    » functions `current_class()`, `current_instance()`
    » functions `print_active_path()`, `print_current_path()`
    » ...

**Tracing**
- `module load gnutools` (for gtkwave)
- `scc DesignName -sc2out -vv -ww -Tvcds
  ./DesignName
gtkwave DesignName.vcd`
- Trace instructions in file `DesignName.do`
- Trace log in file `DesignName.vcd`
- Waveform display, e.g. `gtkwave`

- **Examples**
  - `$SPECC/examples/trace`, see README

- **Documentation**
Lecture 2: Outline

- System design languages
  - Goals, requirements
  - Separation of computation & communication

- The SpecC language
  - Core language syntax and semantics
    - Comparison with SystemC
  - Channel library
  - Compiler and simulator

- The SystemC language
  - Syntax and semantics

SpecC vs. SystemC Methodology

SpecC Abstraction Levels

- Specification model
- Computation model
- Transaction-level model
- Pin-accurate model
- Implementation model

SystemC Abstraction Levels

- Untimed functional (UTF) model
- Timed functional (TF) model
- Transaction-level model
- Pin/Cycle-accurate (CA) model
- RTL model

TLM 1.0
- Programmers view (PV)
- PV w/ timing (PVT)
- Bus-cycle accurate (BCA)

TLM 2.0
- Loosely timed (LT)
- Approximately timed (AT)
Class Library Structure

User's Model

User Libraries
Methodology Libraries
SCV  TLM  etc.

SystemC Library
Built-in Channels
SystemC Core  Data Types

Standard C++ Language

Source: M. Radetzki, Univ. of Stuttgart

The SystemC Language

- **SpecC structural hierarchy**
  - Behaviors
  - Ports and variables
  - Channels and interfaces
- **SpecC behavioral hierarchy**
  - seq, fsm
  - par, pipe
  - try-trap, -interrupt

- **SystemC structural hierarchy**
  - Modules
  - Ports and variables
  - Channels* and interfaces*
- **SystemC behavioral hierarchy**
  - Parallel leaf processes
    - SC_METHOD (combinatorial)
    - SC_THREAD (behavior)

* SystemC 2.0
**SystemC Structure**

A SystemC module consists of:

- **Module**
- **Events**
- **Channel**
- **Interface**
- **Port**

+ Bit-true data types

Source: M. Radetzki, Univ. of Stuttgart

---

**Modules and Ports**

```
// file adder.h
#include "systemc.h"

SC_MODULE(Adder)
{
  sc_in<int> x;
  sc_in<int> y;
  sc_out<int> s;
  ...
};
```

- `#include "systemc.h"`: usage of SystemC library
- `SC_MODULE(Adder)`: name of the module
- `sc_in<int> x; sc_in<int> y; sc_out<int> s;`: input and output ports, named x, y, s
- `...`: port data type
- Important (otherwise, strange error messages from C++ compiler)

Source: M. Radetzki
SC_METHOD Process Declaration

// file adder.h
#include "systemc.h"
SC_MODULE(Adder)
{
   sc_in<int> x;
   sc_in<int> y;
   sc_out<int> s;
   void add();
   SC_CTOR(Adder)
   {
      SC_METHOD(add);
      sensitive << x << y;
   }
};

function prototype
module constructor
function registered as a process
activation condition of the process:
new value (value change) on port x
or port y leads to automatic start of add()

Source: M. Radetzki

SC_METHOD Implementation

// file adder.cpp
#include "adder.h"
void Adder::add()
{
   s = x + y;
}

Alternative:
   s.write(x.read()+y.read());

Source: M. Radetzki
**SC_THREAD Process Declaration**

```c
#include "systemc.h"
SC_MODULE(Adder)
{
    sc_in<int> x;
    sc_in<int> y;
    sc_out<int> s;

    void add();
    SC_CTOR(Adder)
    {
        SC_THREAD(add);
        sensitive << x << y;
    }
}
```

- `function prototype`
- `module constructor`
- `function registered as a process`
- `activation condition defined, but no automatic start of SC_THREAD`

---

**SC_THREAD Implementation**

```c
#include "adder.h"

void Adder::add()
{
    for(;;) // infinite loop
    {
        wait();
        s = x + y;
    }
}
```

- `SC_THREAD is started only once, at the beginning of the simulation`
- `SC_THREAD specifies activation by call to wait function; here: waits for sensitive condition; in adder.h:
  ```c
  sensitive << a << b;
  ```

The above SC_THREAD implementation has the same functionality as the previous SC_METHOD implementation.
SC_METHOD vs. SC_THREAD

SC_METHOD
- Started again whenever activation condition triggers
- Must not call `wait()`
- Must not block
- Must not contain infinite loop (would block all other processes)
- May use non-blocking communications only
- Must not call functions that block or call `wait()`

SC_THREAD
- Started only once, at beginning of simulation
- May (and must) call `wait()`
- Often contains infinite loop
- May (and must) block – gives other processes chance to execute
- May use both non-blocking and blocking communications

Source: M. Radetzki

SystemC Hierarchy (SC_MODULE)

```
SC_MODULE(Testbench)
{
    sc_signal<int> ch_x, ch_y, ch_s; // channels & variables
    Adder uut; // submodule instance
    void stim(); // stimuli process
    void check(); // checking process
    ...

    sc_event_group evg;
    evg += ch_x, ch_y, ch_s;

    ch_x.driven_by(s, evg);
    ch_y.driven_by(s, evg);
    ch_s.driven_by(s, evg);
}
```

Source: M. Radetzki
**SC_MODULE Structure**

```cpp
SC_MODULE(Testbench)
{
    // top level; no ports
    sc_signal<int> ch_x, ch_y, ch_s;  // channels
    Adder uut;  // Adder instance
    void stim();  // stimuli process
    void check();  // checking process
    SC_CTOR(Testbench)
    : uut("uut"), ch_x("ch_x")  // initializer list
    {
        SC_THREAD(stim);  // without sensitivity
        SC_METHOD(check);
        sensitive << ch_s;  // sensitivity for check
        uut.x(ch_x);  // port x of uut bound to ch_x
        uut.y(ch_y);  // port y of uut bound to ch_y
        uut.s(ch_s);  // port s of uut bound to ch_s
    }
}
```

Source: M. Radetzki

---

**Invoking the Simulation from sc_main**

```cpp
// file main.cpp
int sc_main(int argc, char *argv[])  // cf. C++ main()
{
    Testbench tb("tb");
    sc_start();
    cout << "Simulation finished" << endl;
}
```

Source: M. Radetzki
SystemC Events

- Declaration: \texttt{sc\_event <name>};
- Immediate triggering: \texttt{<name>.notify();}
- Waiting for occurrence: \texttt{wait(<name>)};

```c
int x;
int y;
sc\_event new\_stimulus;

void Testbench::stim()
{
    x = 3; y = 4;
    new\_stimulus.notify();
    x = 7; y = 0;
    new\_stimulus.notify();
    // stimulus 7, 0 again
    new\_stimulus.notify();
    ...
}
```

SystemC Time

- \texttt{sc\_time} data type
- Time units:
  - SC\_FS femtosecond $10^{-15}$s
  - SC\_PS picosecond $10^{-12}$s
  - SC\_NS nanosecond $10^{-9}$s
  - SC\_US microsecond $10^{-6}$s
  - SC\_MS millisecond $10^{-3}$s
  - SC\_SEC second $10^0$s
- Time object: \texttt{sc\_time <name>\(<\text{magnitude}, \text{unit}>\);}
- e.g.: \texttt{sc\_time delay(10, SC\_NS)}
- usage, e.g.: \texttt{wait(delay)}
- alternative: \texttt{wait(10, SC\_NS)}

Source: M. Radetzki
Waiting on Events

```c
sc_event a, b, c;
sc_time t(...);
wait();
wait(a);
wait(a & b & c);
wait(a | b | c);
wait(t);
wait(t, a & b);
```

- **Static sensitivity**
  - `sensitive << ...`

- **Dynamic sensitivity**
  - ... on a single event
  - ... on a combination of events
  - all events have happened
  - at least one event has happened
  - ... for a time period
  - ... timeout (wait on event no longer than t)

Source: M. Radetzki

SystemC Channels

- Channel access via port
  - port forwards messages from the process to the channel
  - `instance.port(channel)`
- Direct access to channel
  - process passes messages directly to the channel
  - `channel.message(params)`

Source: M. Radetzki
Channels & Interfaces (sc_fifo)

```
sc_fifo

Method implementations
```

```
sc_fifo_in_if
num_available()
read()

T

my_fifo: sc_fifo<int>

sc_fifo_out_if
num_free()
write()
```

Source: M. Radetzki

Putting It All Together

```
sc_fifo_in_if sc_fifo_out_if

m

Module

Process

p

my_fifo

Process

m.p(my_fifo)

my_fifo.write(.)

sc_port<sc_fifo_in_if<.> > p;

sc_fifo_in<.> p;

p->read()
```

Source: M. Radetzki
SystemC Channels

- **Minimal Channel**
  - implements ≥ 1 interface(s)
  - is-a `sc_interface`

- **Hierarchical Channel**
  - implements ≥ 1 interface(s)
  - is-a `sc_channel (sc_module)`
  - can use module features
  - may have own process (active)

- **Primitive Channel**
  - implements ≥ 1 interface(s)
  - is-a `sc_prim_channel`
  - can interface with sim. kernel
  - has no process (is passive)

User-defined channels are often hierarchical channels

SystemC built-in channels are primitive channels

Source: M. Radetzki

SystemC Built-In Channels

- `sc_interface`
- `sc_signal<T>`
- `sc_fifo<T>`
- `sc_buffer<T>`
- `sc_mutex`, `sc_semaphore`
- `sc_clock`
- Your own IFs / channels / ports

Source: M. Radetzki
### FIFO Example: Channel

```cpp
class fifo :
    public sc_channel, public write_if, public read_if {
public:
    fifo(sc_module_name name): sc_channel(name),
    num_elements(0), first(0) {}

do_write(c) {
    if (num_elements == max)
        wait(read_event);
    data[(first + num_elements++) % max] = c;
    write_event.notify();
}

do_read(&c) {
    if (num_elements == 0)
        wait(write_event);
    c = data[first]; --num_elements;
    first = (first + 1) % max;
    read_event.notify();
}

do_reset() { num_elements = first = 0; }

private:
    enum e { max = 10 };  // Source: S. Swan, Cadence Design Systems
    char data[max];
    int num_elements, first;
    sc_event write_event, read_event;

SC_MODULE(fifo),
    SC_CTOR(fifo):
    num_elements(0), first(0) {}}
```

### FIFO Example: Behaviors

```cpp
SC_MODULE(consumer)
{
public:
    sc_port<read_if> in;
}

SC_CTOR(consumer)
{
    SC_THREAD(main);
}

void main()
{
    char c;
    while (true) {
        in->read(c);
        if (in->num_available())
            ...
        if (...) in->reset();
    }
}

SC_MODULE(producer)
{
public:
    sc_port<write_if> out;
}

SC_CTOR(producer)
{
    SC_THREAD(main);
}

void main()
{
    char c;
    while (true) {
        out->write(c);
        if (out->reset())
            ...
    }
}
```

Source: S. Swan, Cadence Design Systems
**FIFO Example: Main**

```c++
SC_MODULE(top),
{
public:
    fifo *fifo_inst;
    producer *prod_inst;
    consumer *cons_inst;

SCCTOR(top)
{
    fifo_inst = new fifo("Fifo1");
    prod_inst = new producer("Producer1");
    prod_inst->out(*fifo_inst);
    cons_inst = new consumer("Consumer1");
    cons_inst->in(*fifo_inst);
}
};

int sc_main (int argc , char *argv[])
{
    top top1("Top1");
    sc_start();
    return 0;
}
```

Source: S. Swan, Cadence Design Systems

---

**Lecture 2: Summary**

- **SpecC language**
  - True superset of ANSI-C
    - ANSI-C plus extensions for HW-design
  - Support of all concepts needed in system design
    - Orthogonal, executable, synthesizable
  - Standardization and adoption
    - SpecC Technology Open Consortium (STOC), industry & academia
- **Tools**
  - Compilation, validation, simulation

- **SystemC language**
  - C++ class library
    - Don’t invent a new language, leverage existing tools
  - De-facto industry-standard
    - Architecture & transaction-level modeling