System-Level Design (and Modeling for Embedded Systems)

Lecture 4 – The SpecC Language

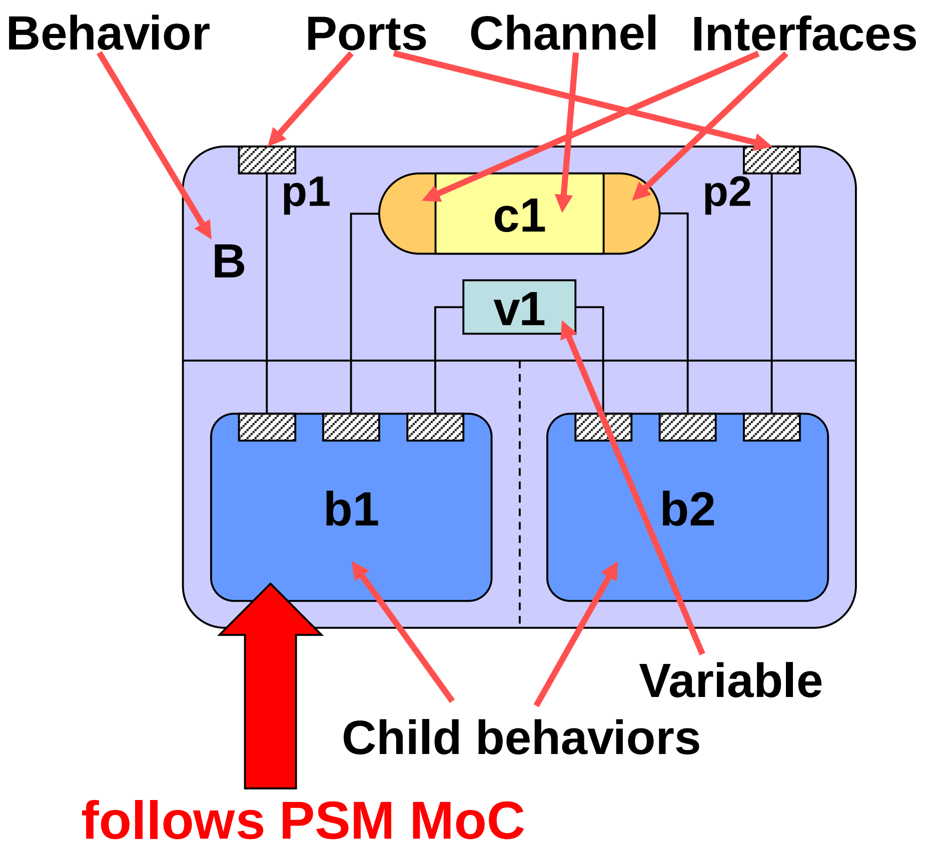
Kim Grüttner kim.gruettner@dlr.de
Henning Schlender henning.schlender@dlr.de
Jörg Walter joerg.walter@offis.de

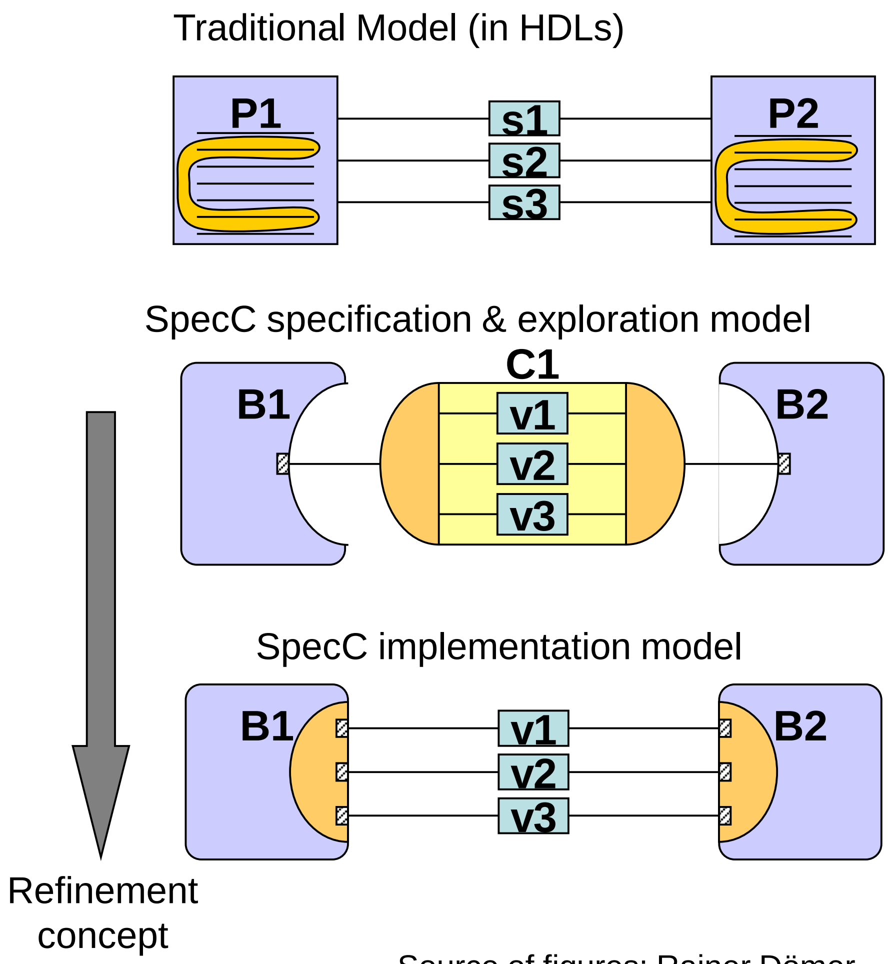
System Evolution and Operation
German Aerospace Center (DLR)
&
Distributed Computation and Communication
OFFIS

Properties of SLDLs

- **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**

SpecC Overview



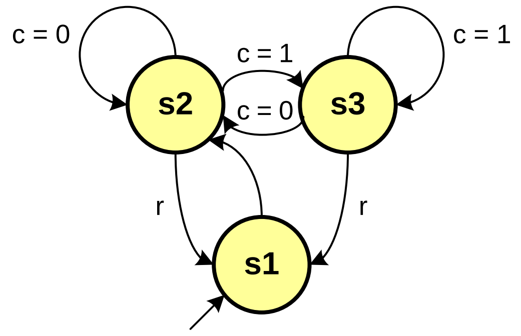


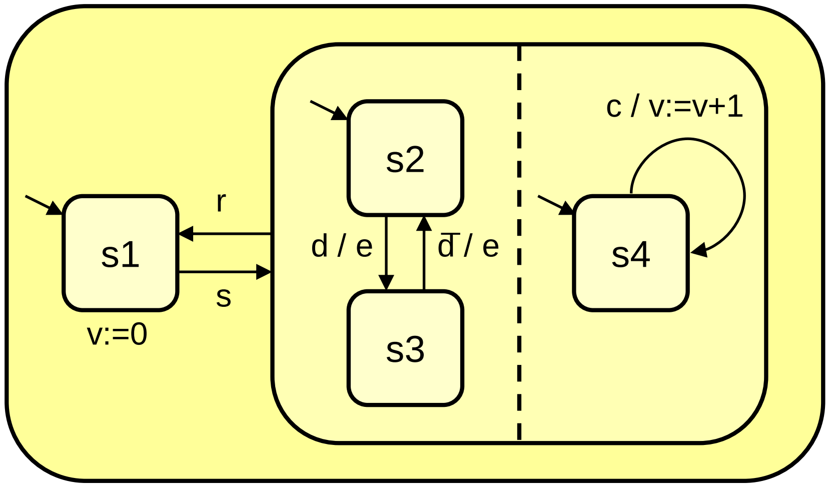
Traditional Model (in HDLs)
P1
s1
s2
s3
P2
SpecC specification & exploration model
C1
B1
v1
v2
v3
B2
SpecC implementation model
B1
v1
v2
v3
B2
Refinement concept

Source of figures: Rainer Dömer
University of California, Irvine

PSM Model of Computation

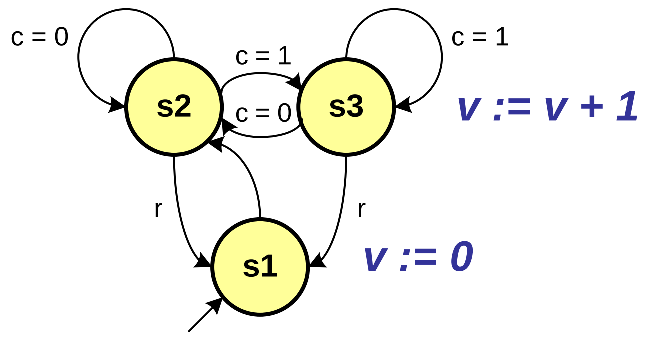
- **Finite-State Machine (FSM)**
  - Description of control systems
  - Moore Type
    - State based
    - <S, I, O, f : S x I -> S, **h : S -> O**>
  - Mealy Type
    - Transition based
    - <S, I, O, f : S x I -> S, **h : S x I -> O**>
  - Potential problem: state & arc explosion
    - Can be handled with **Hierarchical Concurrent Finite-State Machines (HCFSM)**
    - e.g. StateCharts [D. Harel]





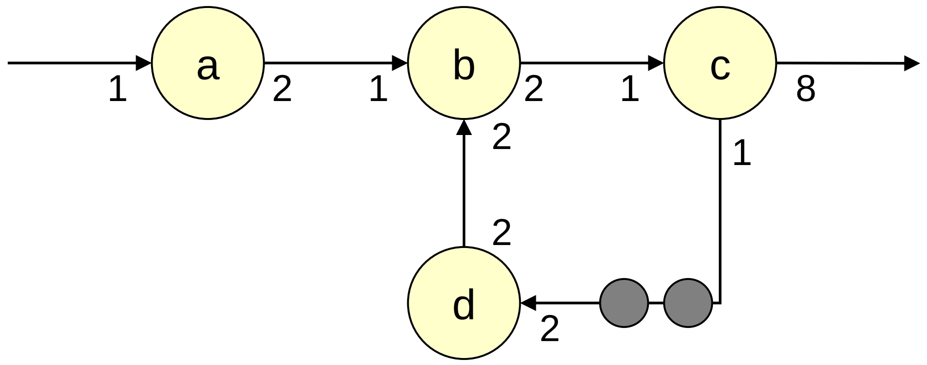
PSM Model of Computation

- **Finite-State Machine with Datapath (FSMD)**
  - Description of control and computation-oriented systems
  - FSM plus variables V
  - FSMD: <S, I , O, V, f , h>
    - Next state function *f*: $S \times V \times I \rightarrow S \times V$
    - Output function *h*: $S \times V \times I \rightarrow O$
  - For modelling Hardware
    - Each state transition at a single clock cycle
    - State is set of RT operations



PSM Model of Computation

- **Dataflow Graph (DFG)**
  - Description of computationally intensive systems
  - Set of nodes
    - Input (or source) and Output (or destination)
    - Activity (or process)
      - Transformation or manipulation of data
      - Can be program, function or instruction
    - Data Store
  - Set of directed edges
    - Define execution order
    - Labelled with transmitted data

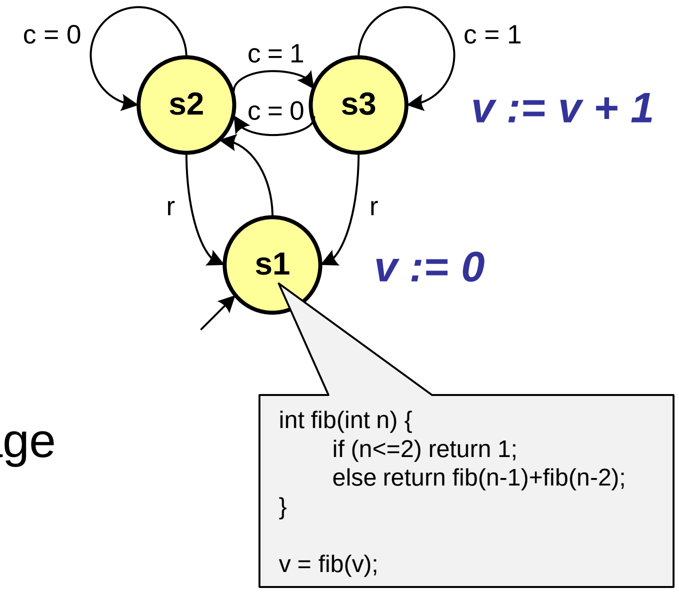


PSM Model of Computation

- **Programming Language**
  - Heterogeneous model
    - Data
    - Activity
    - Control
  - Basic paradigms
    - Imperative
    - Declarative
  - Focus on imperative (and object-oriented) languages
    - Data: basic types, composite types, or object types
    - Activities: Statements, Functions or Procedures, Methods
    - Control: sequential composition (;), branching (if, switch), looping (while, do-while, for)

PSM Model of Computation

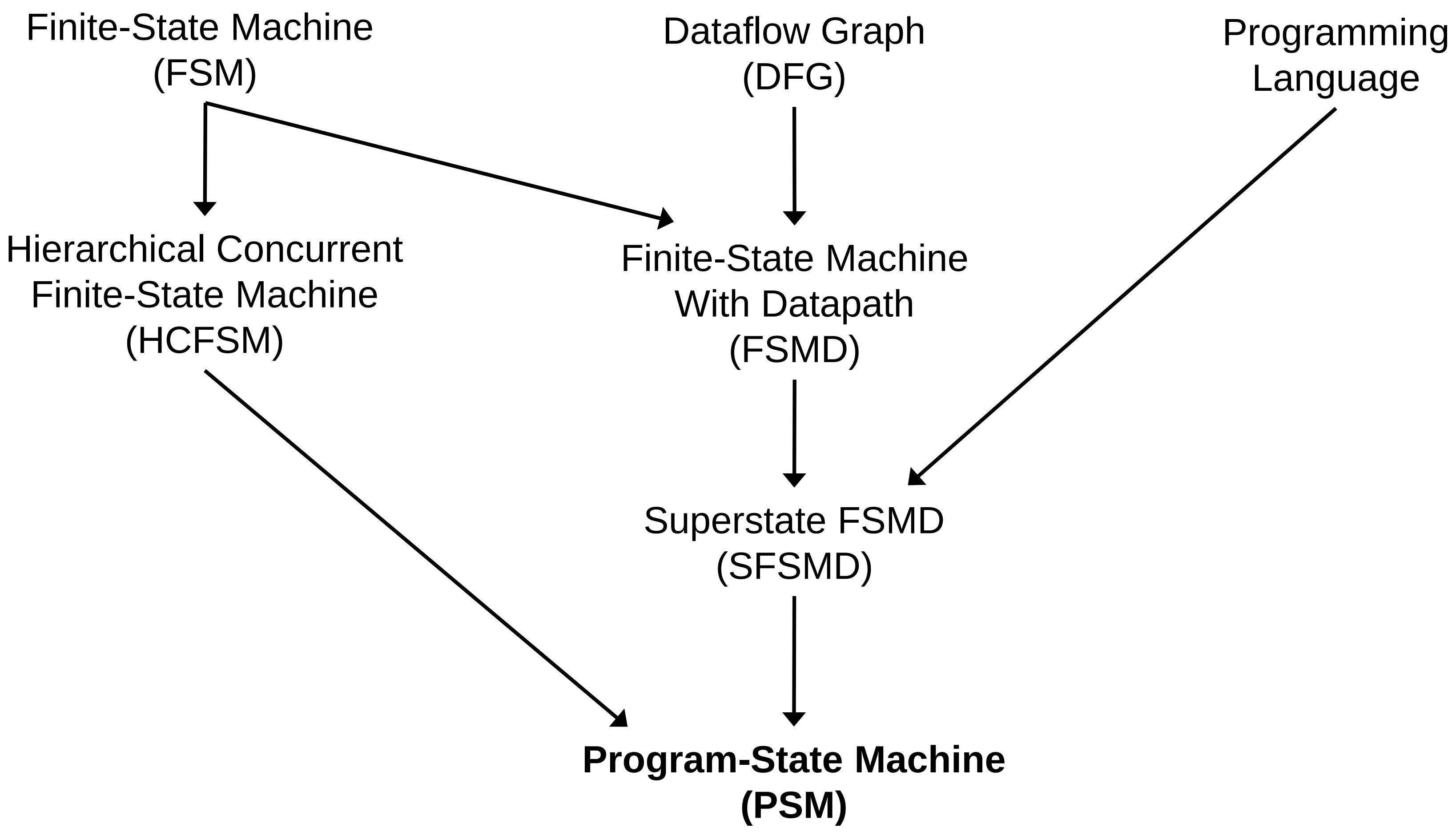
- **Superstate FSMD (SFSMD)**
  - FSMD and programming language
  - Superstate
    - is not required to be single clock cycle
    - can take arbitrary amount of time (depends on implementation)
    - is specified by constructs of programming language



```
int fib(int n) {
        if (n<=2) return 1;
        else return fib(n-1)+fib(n-2);
}

v = fib(v);
```

- **Program-State Machine (PSM)**
  - Replace FSM in HCFSM by SFSMD -> PSM
  - Hierarchy of program-states (behaviour),
  - each specifying single mode of computation
  - Only subset of program-states active at each time
  - Can be decomposed into program sub-states:
    - Sequential
    - Concurrent

The SpecC Model of Computation


Finite-State Machine (FSM)
Dataflow Graph (DFG)
Programming Language
Hierarchical Concurrent Finite-State Machine (HCFSM)
Finite-State Machine With Datapath (FSMD)
Superstate FSMD (SFSMD)
Program-State Machine (PSM)

Lecture 4: Outline

- **The SpecC Language**
  - Foundation
  - Types
  - Structural and behavioral hierarchy
  - Concurrency
  - State transitions
  - Exception handling
  - Communication
  - Synchronization
  - Timing
- **SpecC Tools**
  - Compiler and simulator

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

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");
}
```

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;`

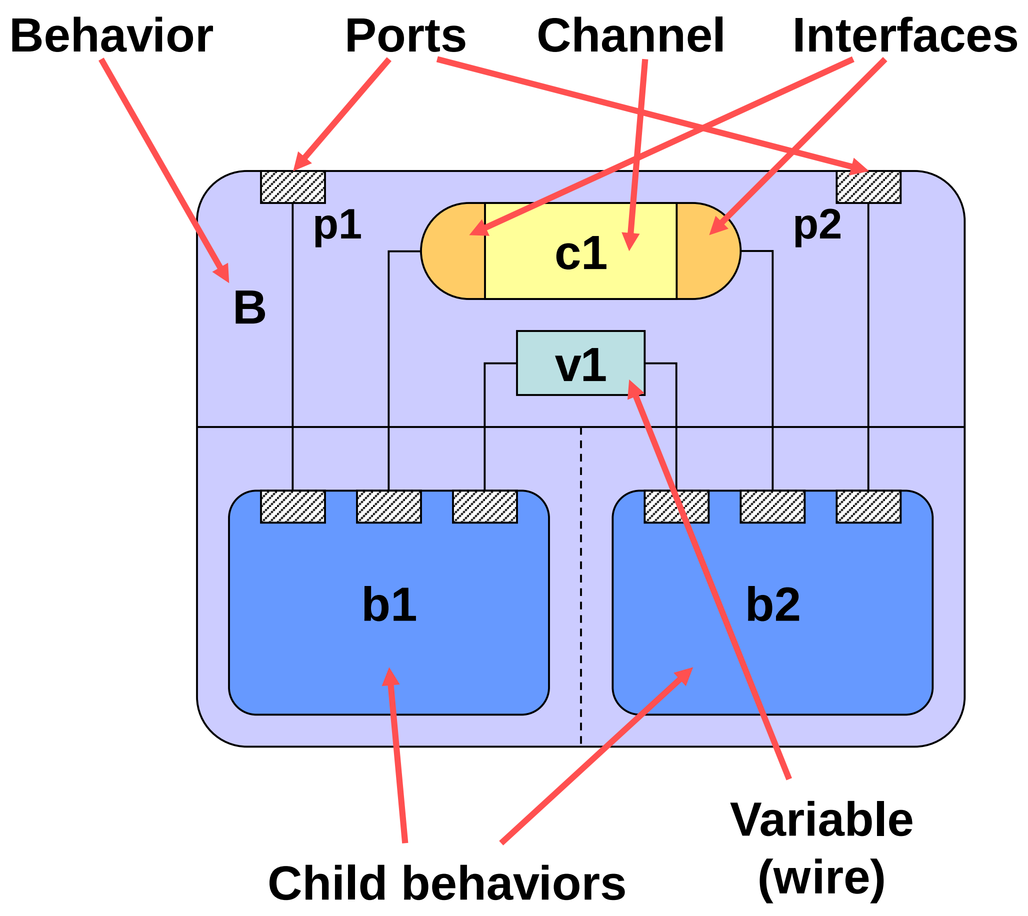
The SpecC Language

- **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

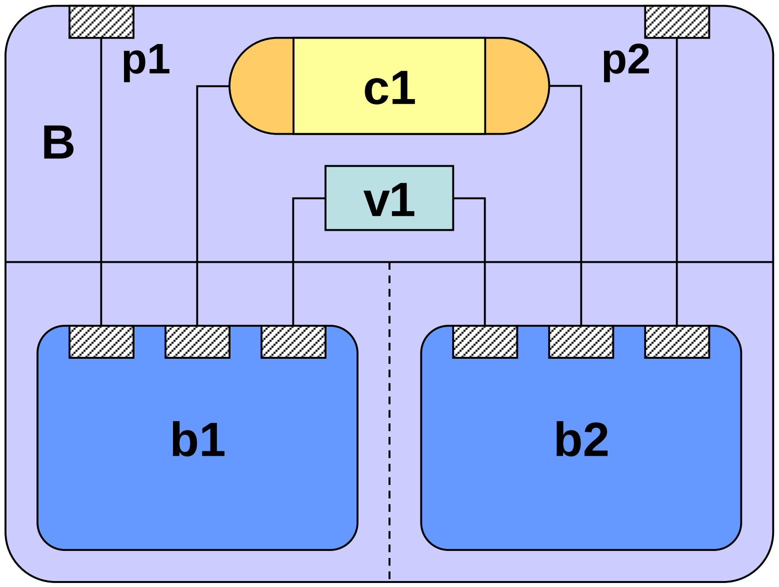
- **Basic structure**
  - Top behavior
  - Child behaviors
  - Channels
  - Interfaces
  - Variables (wires)
  - Ports



The SpecC Language

- **Basic structure**

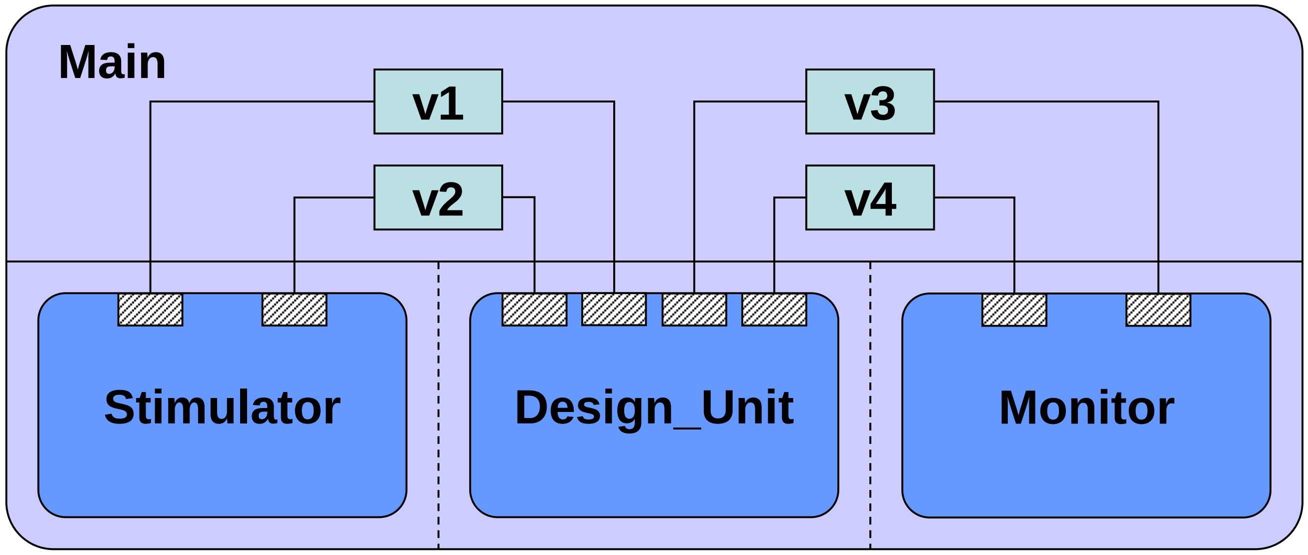
```
interface I1
{
  bit[63:0] Read(void);
  void Write(bit[63:0]);
};

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();`

The SpecC Language

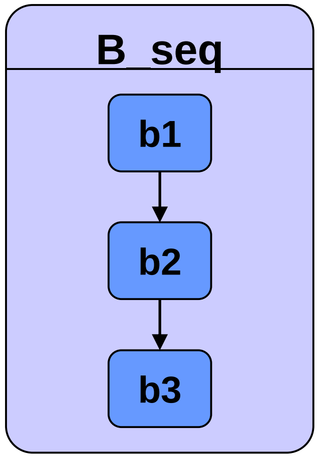
- **Typical test bench**
  - Top-level behavior: `Main`
  - Stimulator provides test vectors
  - Design unit under test
  - Monitor observes and checks outputs

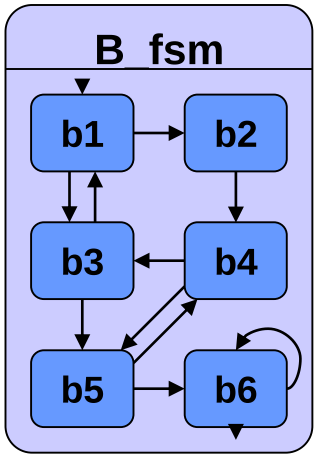


The SpecC Language

- **Behavioral hierarchy**

Sequential execution	FSM execution	Concurrent execution	Pipelined execution





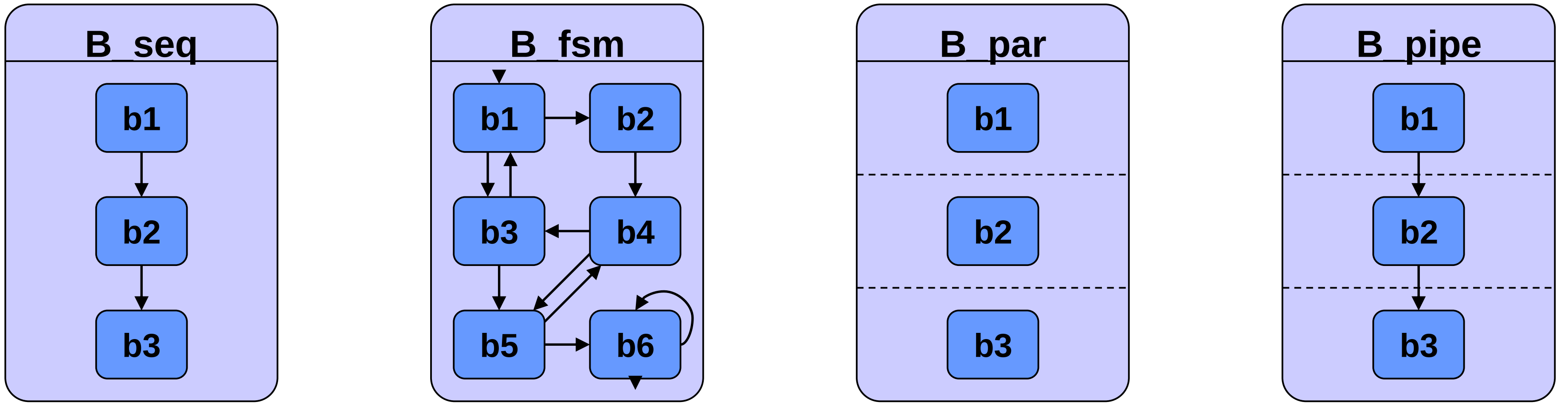
```
behavior B_seq
{
 B b1, b2, b3;

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

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

The SpecC Language

- **Behavioral hierarchy**

Sequential execution	FSM execution	Concurrent execution	Pipelined execution



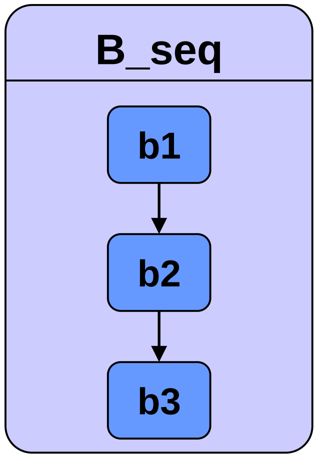
```
behavior B_seq
{
 B b1, b2, b3;

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

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

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

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

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

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

The SpecC Language



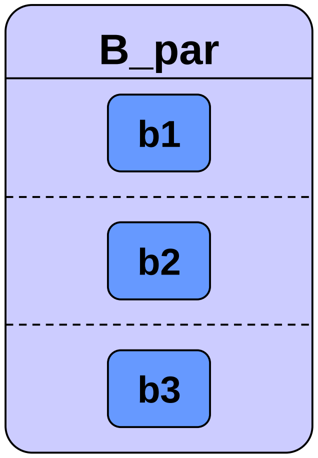
```
behavior B1() {
  void main(void) {
    // your code here!
  }
};

behavior B_seq() {
  B1 b1;
  B2 b2;
  B3 b3;

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

In this example the execution of each behavior B1, B2 and B3 takes 1 ns (Nanosecond)

Execution trace

```
[Reset]
...
@110 ns : void B1::main()
@111 ns : void B2::main()
@112 ns : void B3::main()
[End]
```

The SpecC Language

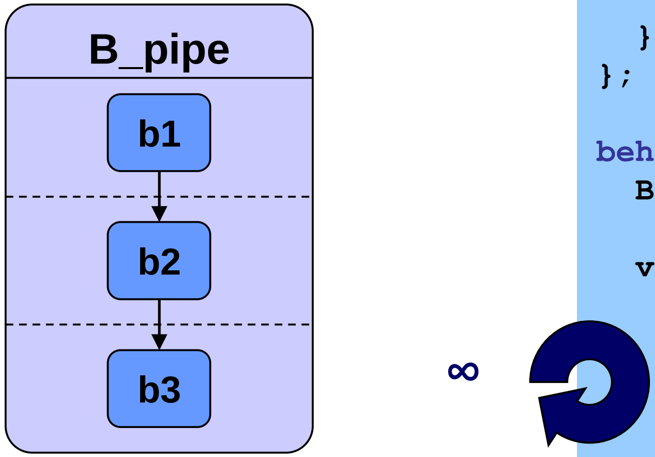


```
behavior B1() {
  void main(void) {
    // your code here!
  }
};

behavior B_par() {
  B1 b1;
  B2 b2;
  B3 b3;

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

Execution trace

```
[Reset]
...
@110 ns : void B1::main()
@110 ns : void B2::main()
@110 ns : void B3::main()
[End]
```

The SpecC Language



```
behavior B1() {
  void main(void) {
    // your code here!
  }
};

behavior B_pipe() {
  B1 b1; B2 b2; B3 b3;

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

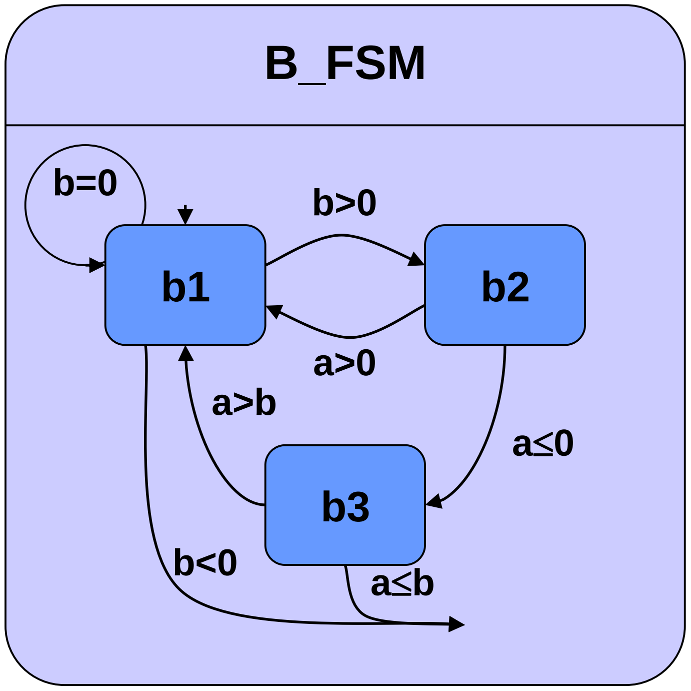
Execution trace

```
[Reset]
...
@110 ns : void B1::main()
@111 ns : void B1::main()
@111 ns : void B2::main()
@112 ns : void B2::main()
@112 ns : void B1::main()
@112 ns : void B3::main()
...
```

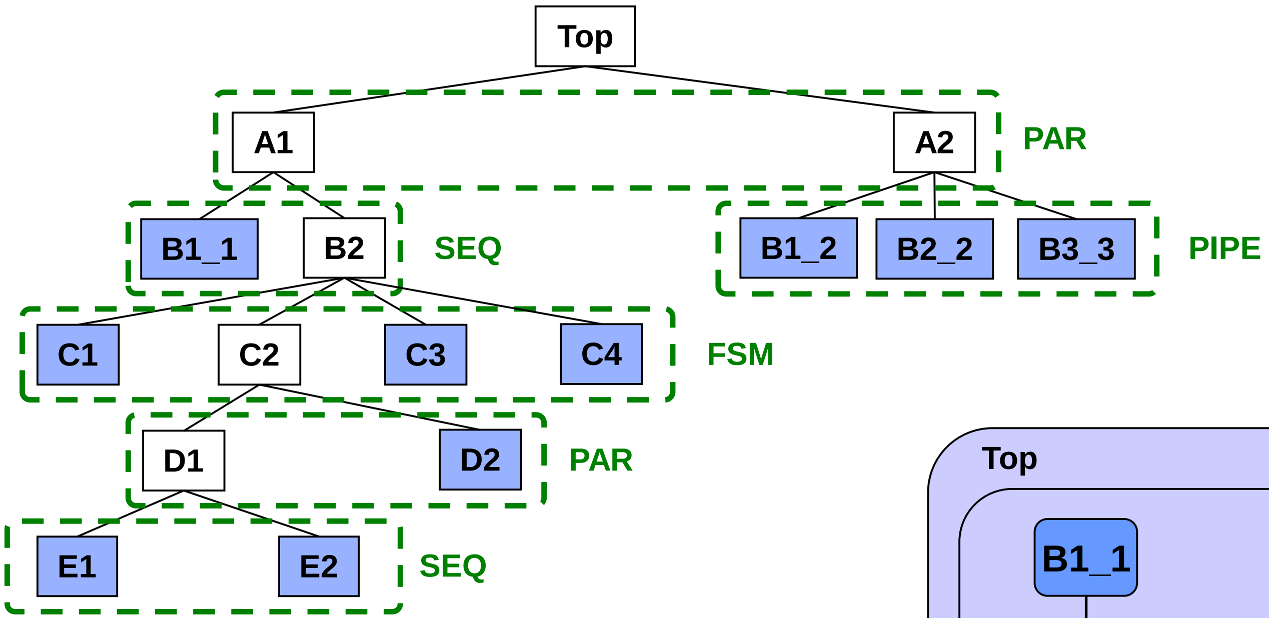
The SpecC Language

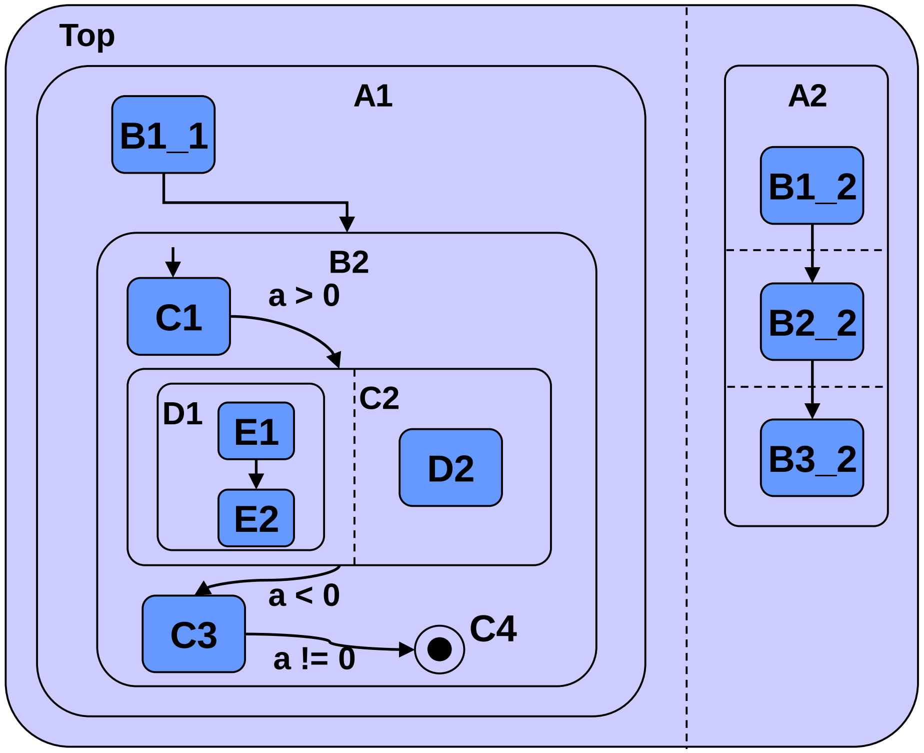
- **Finite State Machine (FSM)**
  - Explicit state transitions
    - triple < *current_state, condition, next_state* >
    - **fsm** { <*current_state*> : { **if** <*condition*> **goto** <*next_state*> } … }
  - Moore-type FSM
  - Mealy-type FSM

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



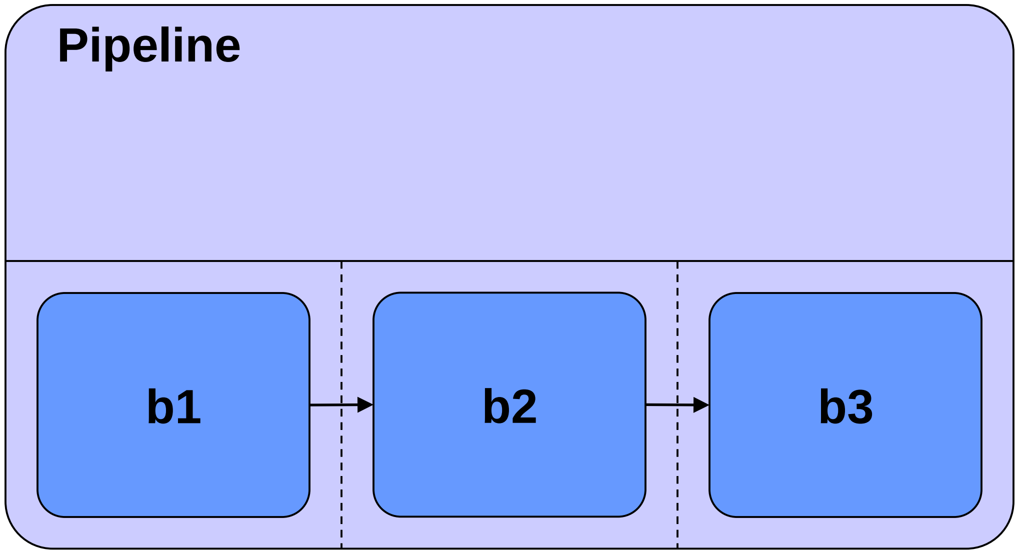
Hierarchical Composition





The SpecC Language

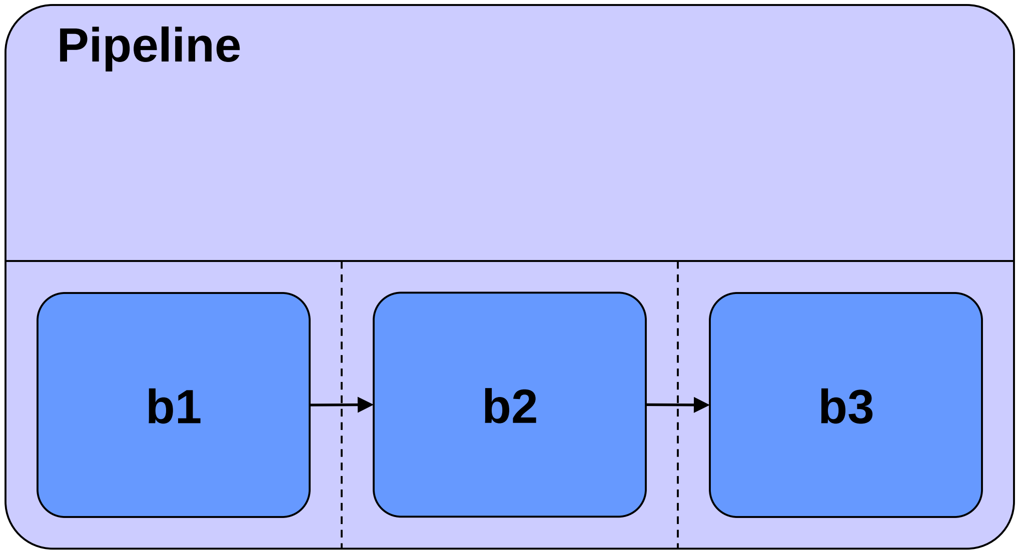
- **Pipeline**
  - Explicit execution in pipeline fashion
    - **pipe** { <*instance_list*> };



```
behavior Pipeline
{




 Stage1 b1;
 Stage2 b2;
 Stage3 b3;

 void main(void)
 {

   pipe
       { b1;
         b2;
         b3;
        }
   }
};
```

The SpecC Language

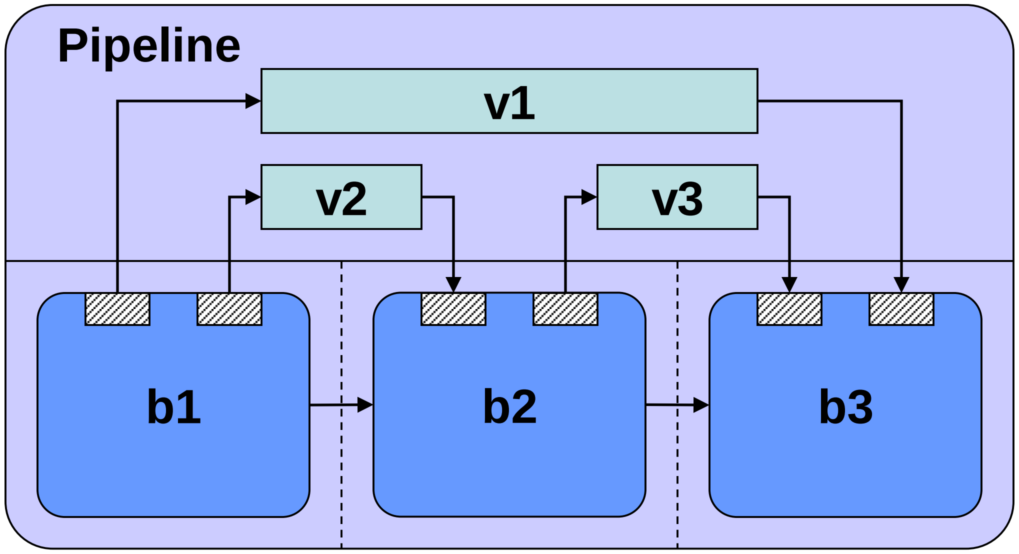
- **Pipeline**
  - Explicit execution in pipeline fashion
    - **pipe** { <*instance_list*> };
    - **pipe** (<*init*>; <*cond*>; <*incr*>) { … }



```
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

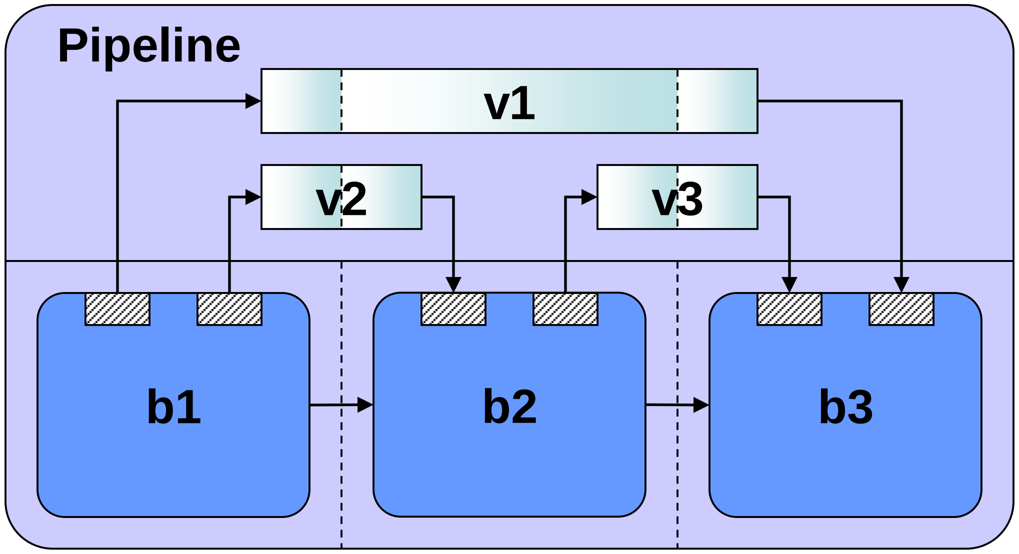
- **Pipeline**
  - Explicit execution in pipeline fashion
    - **pipe** { *<instance_list>* };
    - **pipe** (*<init>*; *<cond>*; *<incr>*) { … }
  - Support for automatic buffering



```
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

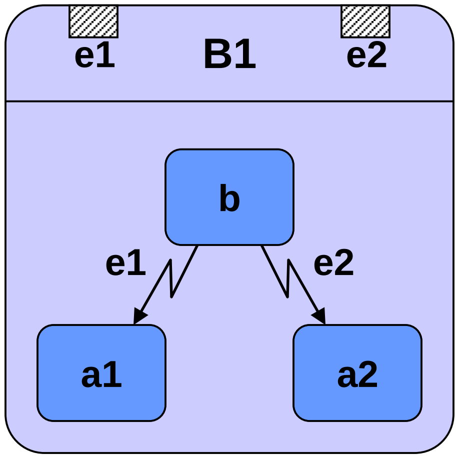
- **Pipeline**
  - Explicit execution in pipeline fashion
    - **pipe** { <*instance_list*> };
    - **pipe** (<*init*>; <*cond*>; <*incr*>) { … }
  - Support for automatic buffering
    - **piped** […] <*type*> <*variable_list*>;



```
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

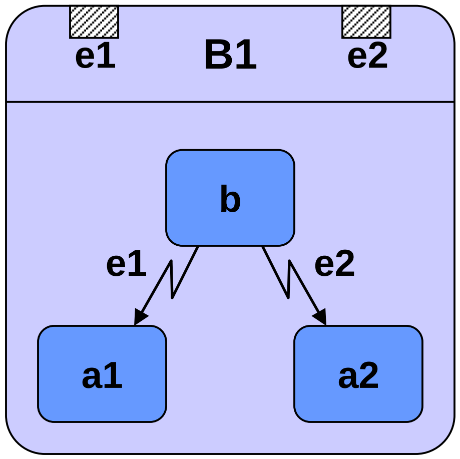
- **Exception handling**
  - Abortion
  - Interrupt

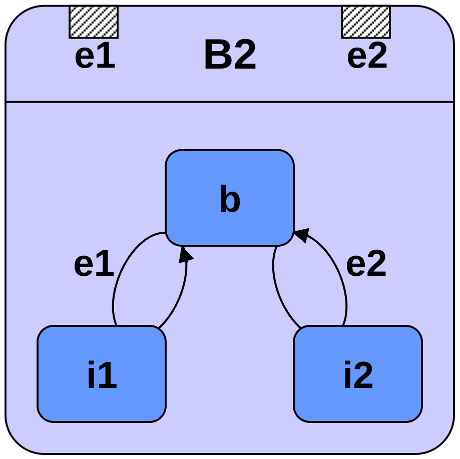


```
behavior B1(in event e1, in event e2)
{
 B b, a1, a2;

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

The SpecC Language

- **Exception handling**
  - Abortion
  - Interrupt



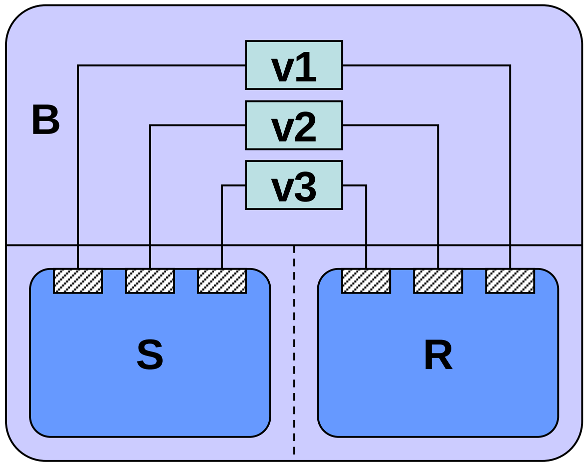


```
behavior B1(in event e1, in event e2)
{
 B b, a1, a2;

 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

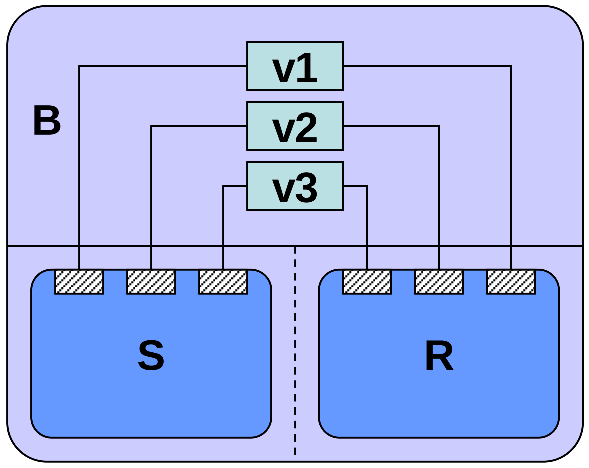
- **Communication**
  - via shared variable



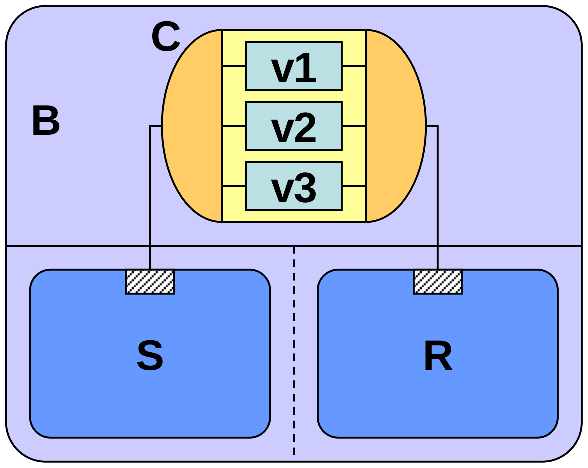
Shared memory

The SpecC Language

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



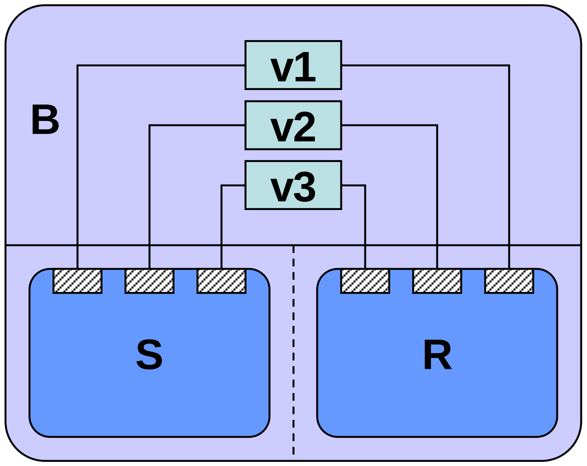
Shared memory



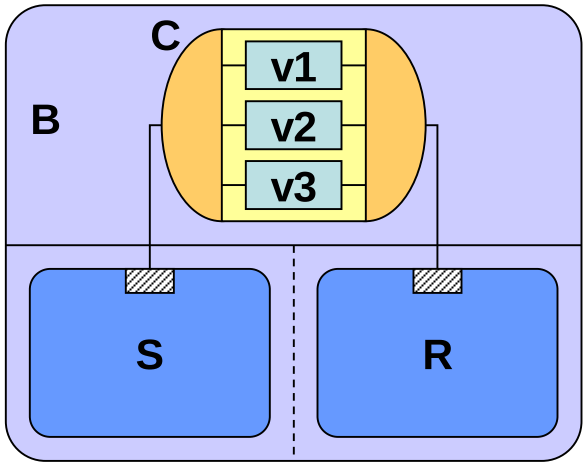
Message passing

The SpecC Language

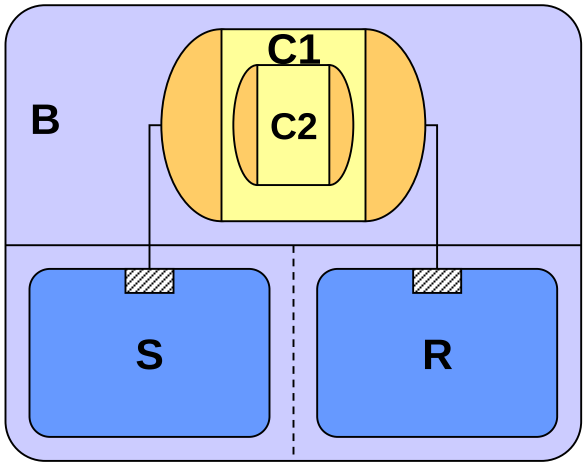
- **Communication**
  - via shared variable
  - via virtual channel
  - via hierarchical channel



Shared memory



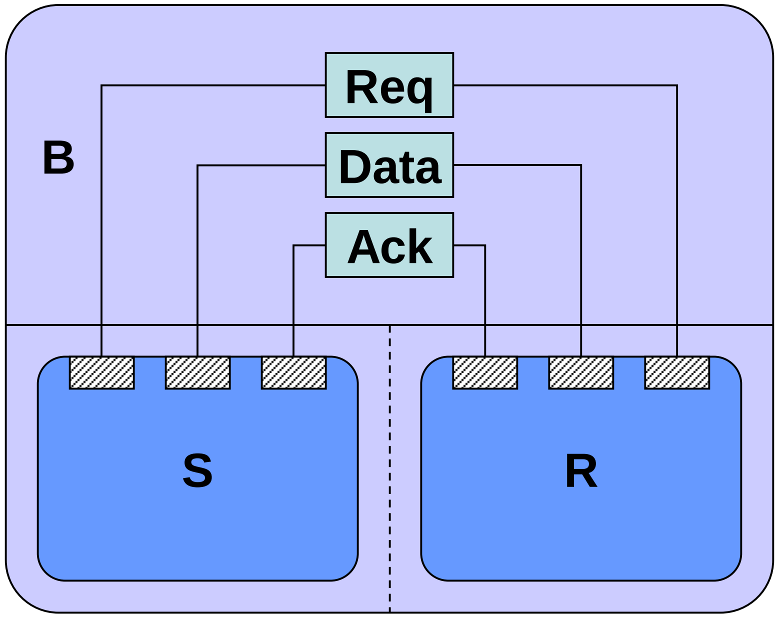
Message passing



Protocol stack

The SpecC Language

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

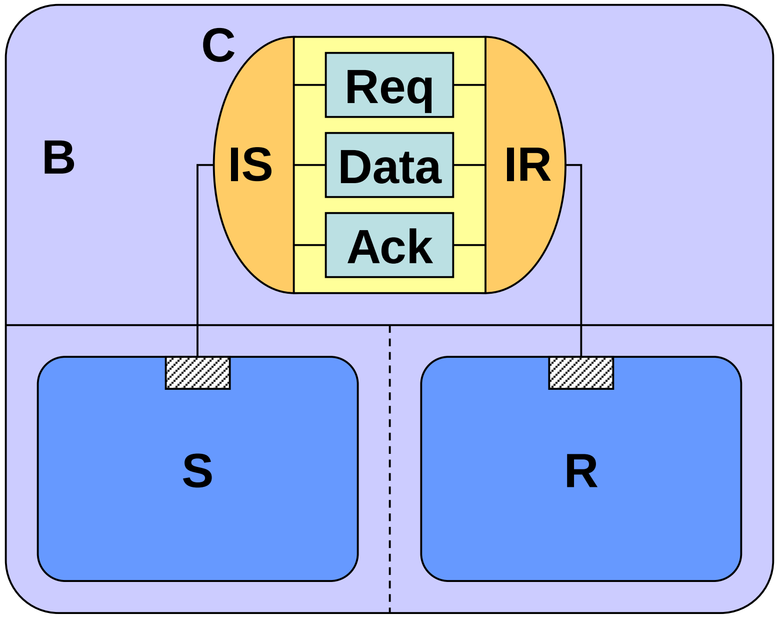


```
behavior S(out event Req,
           out float Data,
           in  event Ack)
{
 float X;
 void main(void)
 { ...
   Data = X;
   notify Req;
   wait Ack;
   ...
 }
};
```

```
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*> };



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

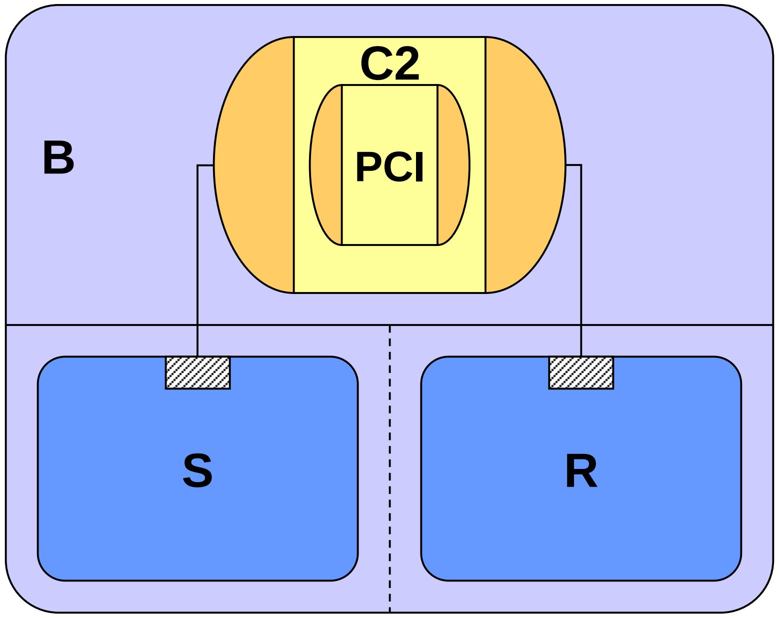
```
behavior R(IR Port)
{
 float Y;
 void main(void)
 {...
  Y=Port.Receive();
  ...
 }
};
```

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

```
channel 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

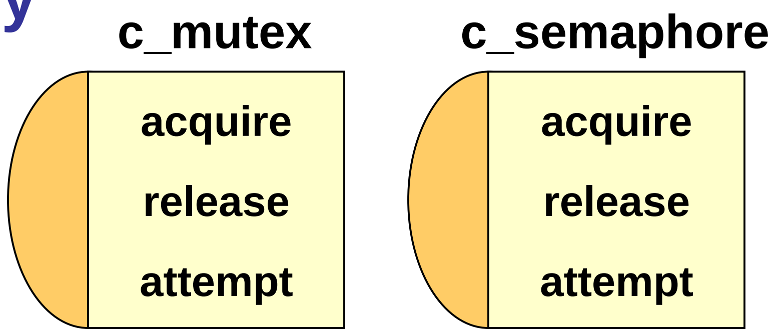
- **SpecC Standard Channel Library**
  - introduced with SpecC Language Version 2.0
  - includes support for
    - mutex
    - semaphore
    - critical section
    - barrier
    - token
    - queue
    - handshake
    - double handshake
    - …

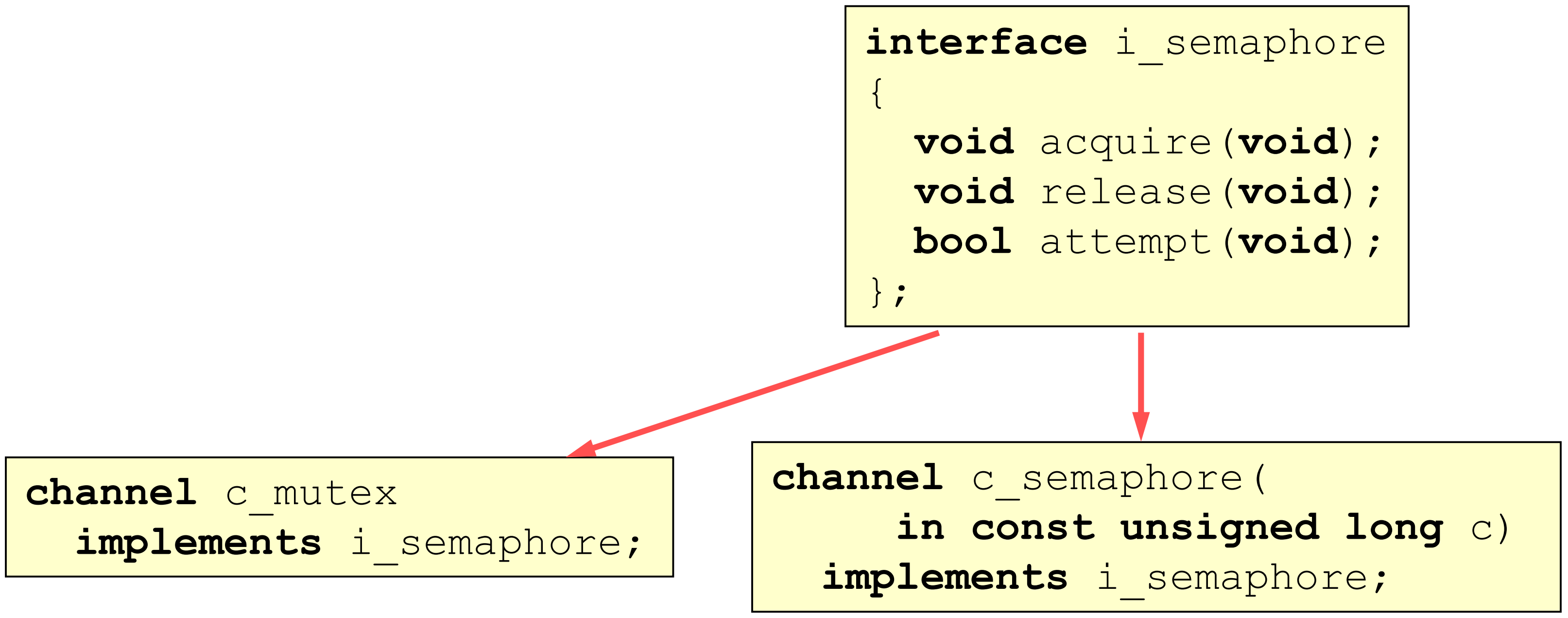
➢ Examples under

  ➢ **`$SPECC/examples/sync`**

The SpecC Language

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





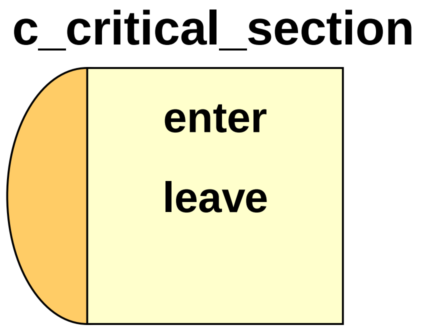
```
interface i_semaphore
{
  void acquire(void);
  void release(void);
  bool attempt(void);
};
```

```
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

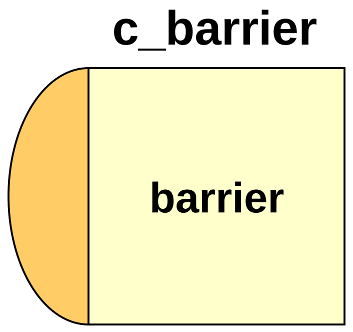


```
interface i_critical_section
{
  void enter(void);
  void leave(void);
};
```

```
channel c_critical_section
  implements i_critical_section;
```

The SpecC Language

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

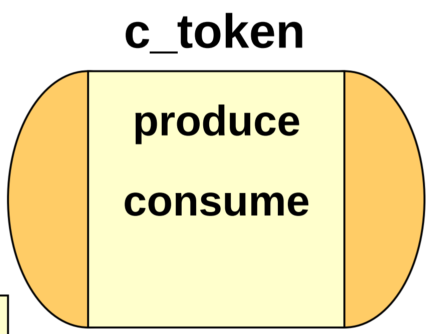


```
interface i_barrier
{
  void barrier(void);
};
```

```
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_token
{
  void consume(unsigned long n);
  void produce(unsigned long n);
};
```

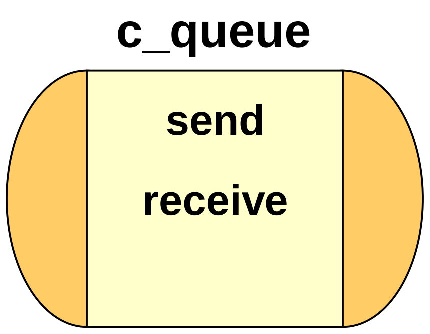
```
interface i_consumer
{
  void consume(unsigned long n);
};
```

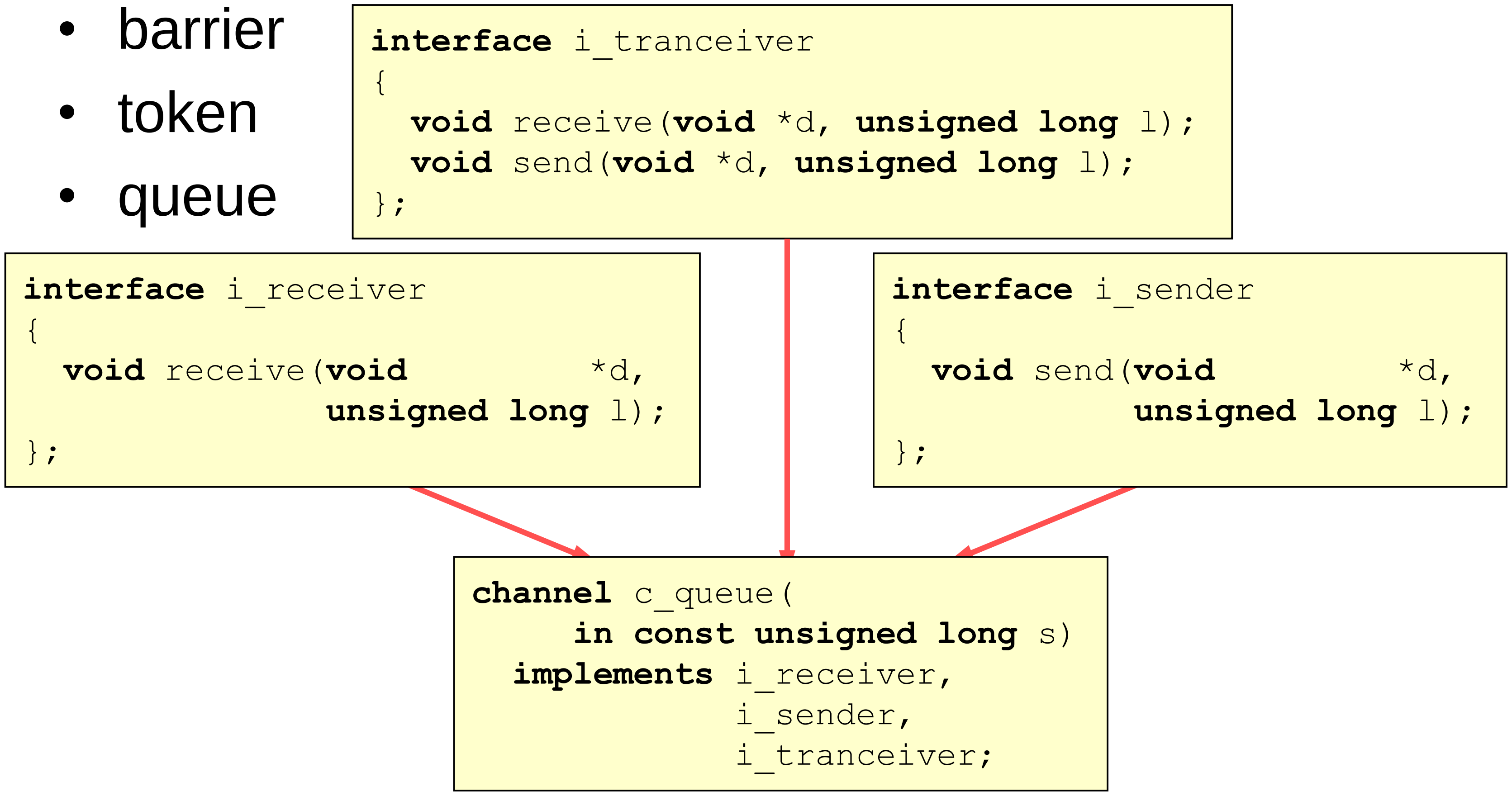
```
interface i_producer
{
  void produce(unsigned long n);
};
```

```
channel c_token
  implements i_consumer,
             i_producer,
             i_token;
```

The SpecC Language

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





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

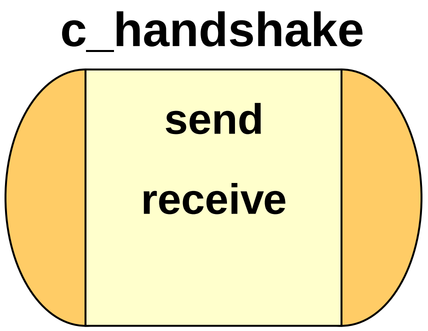
```
interface i_receiver
{
  void receive(void          *d,
               unsigned long l);
};
```

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

```
channel c_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
  - handshake



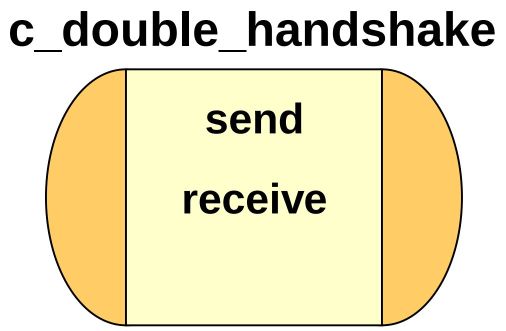
```
interface i_receive
{
  void receive(void);
};
```

```
interface i_send
{
  void send(void);
};
```

```
channel c_handshake
  implements i_receive,
             i_send;
```

The SpecC Language

- **SpecC Standard Channel Library**
  - mutex channel
  - semaphore channel
  - critical section
  - barrier
  - token
  - queue
  - handshake
  - 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);
};
```

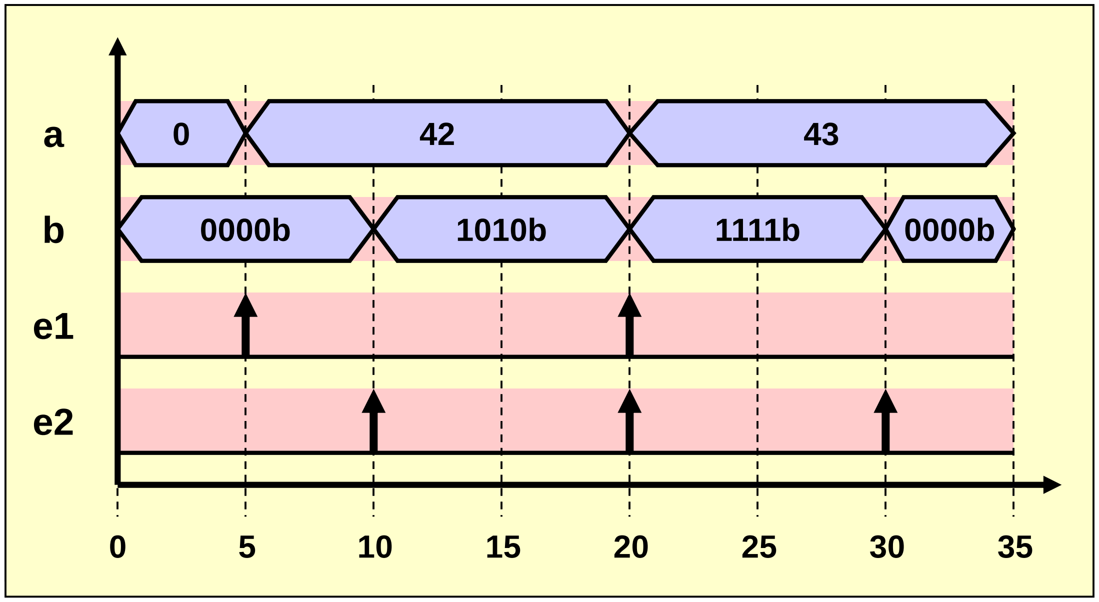
```
interface i_sender
{
  void send(void          *d,
            unsigned long l);
};
```

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

The SpecC Language

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

Example: stimulator for a test bench

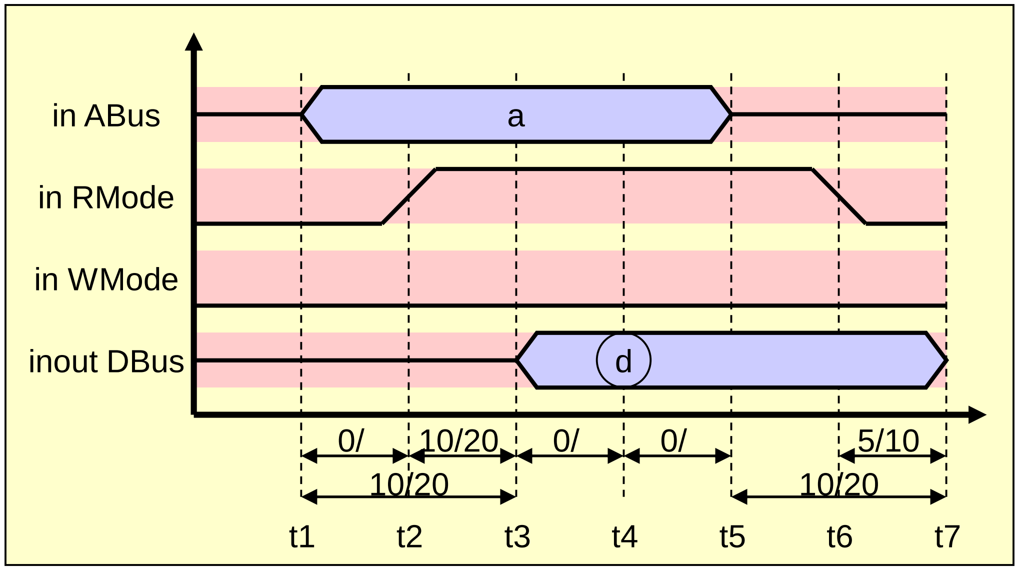


```
behavior Testbench_Driver
         (inout int a,
          inout int b,
          out event e1,
          out event e2)
{
  void main(void)
  {
   waitfor 5;
   a = 42;
   notify e1;

   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



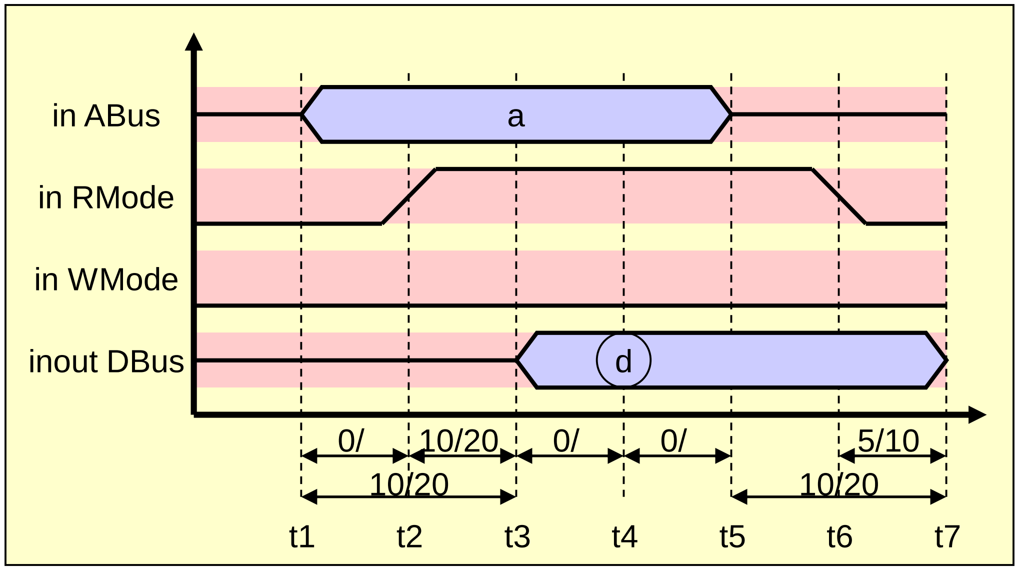
Specification

```
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



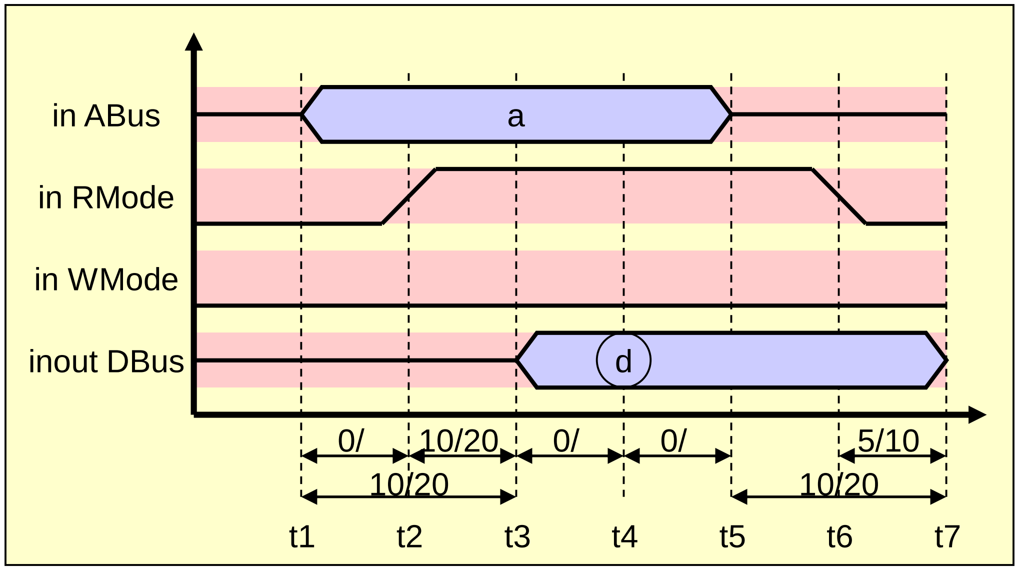
Implementation 1

```
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

- **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

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;
  ...
};
```

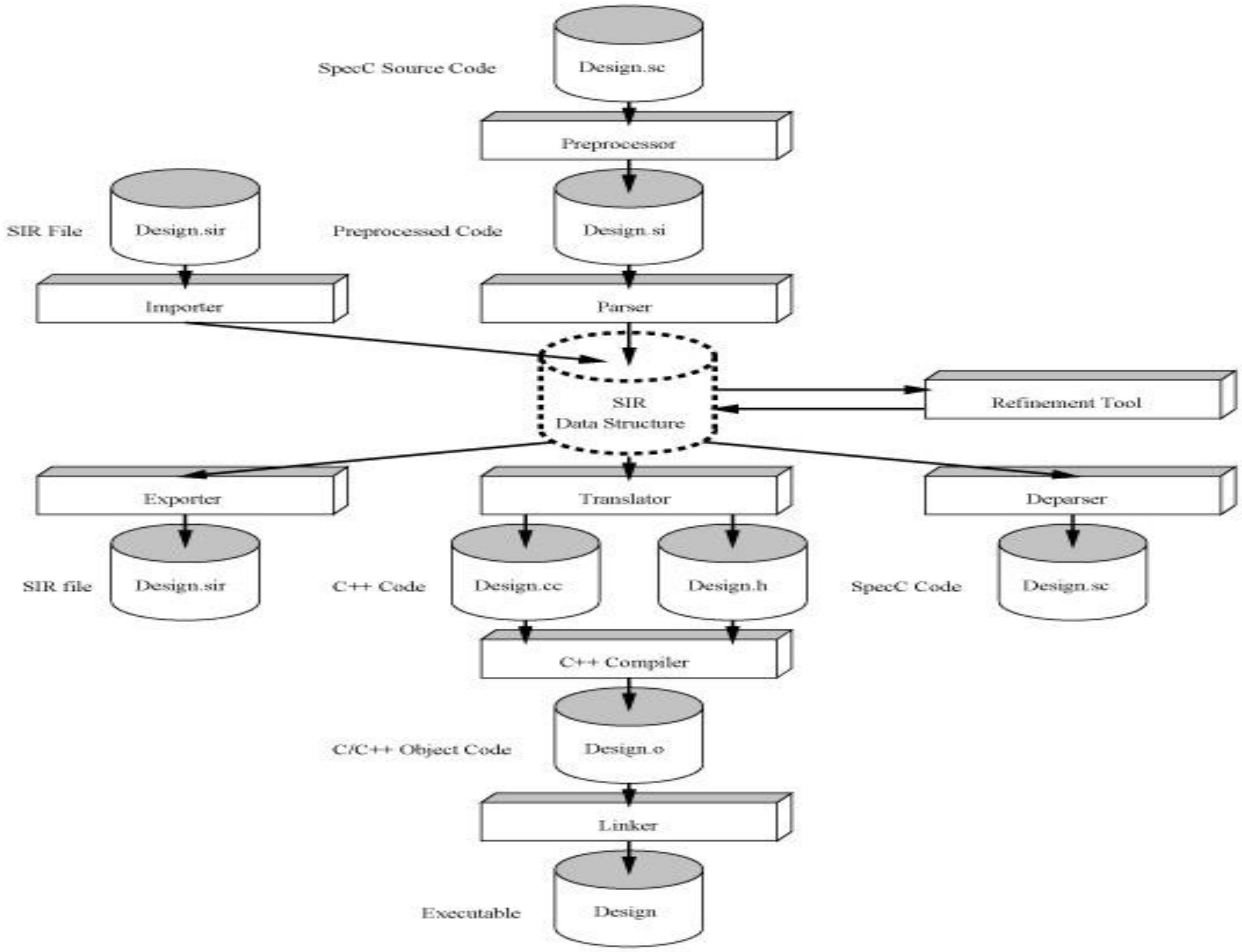
Lecture 4: Outline

- ✓ **The SpecC Language**
  - ✓ Foundation
  - ✓ Types
  - ✓ Structural and behavioral hierarchy
  - ✓ Concurrency
  - ✓ State transitions
  - ✓ Exception handling
  - ✓ Communication
  - ✓ Synchronization
  - ✓ Timing
- **SpecC Tools**
  - Compiler and simulator

SpecC Tools

- **SpecC compiler package**
  - SpecC Reference Compiler available under BSD license at http://www.cecs.uci.edu/~specc/reference/index.html
  - The SCRC distribution consists of:
    - SpecC compiler (parser, internal representation, code generator)
    - SpecC simulator (run-time libraries for SpecC execution)
    - SpecC standard channel library (channels for synchronization, etc.)
    - test suite (SpecC LRM compliance test cases)
  - Design examples avaliable at http://www.cecs.uci.edu/~specc/download.html#examples
    - MP3 Player
    - Parity Checker
    - JPEG Encoder
    - GSM Vocoder

SpecC Compiler (scc)



Lecture 4: Summary

- **SpecC model**
  - PSM model of computation
    - Separation of computation and communication, plug-and-play
    - Behaviors, channels, variables and interfaces
- **SpecC language**
  - True superset of ANSI-C
    - ANSI-C plus extensions for HW-design
  - Support of all concepts needed in system design
    - Hierarchy, concurrency, state, time, communication/synchronization
    - Orthogonal, executable, synthesizable
- **SpecC Tools**
  - Compilation, validation, simulation