ECE382N.23: Embedded System Design and Modeling

Lecture 5 – System Architecture Modeling

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

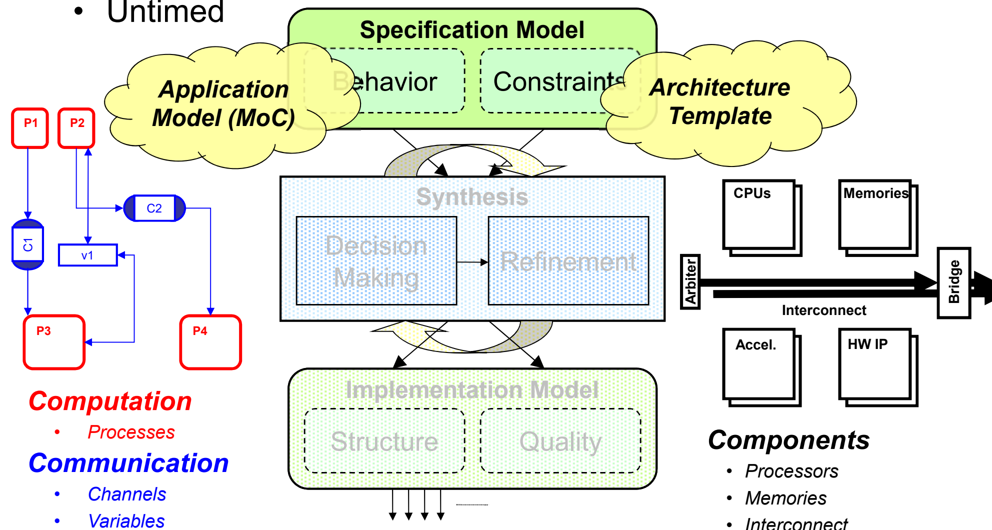
The University of Texas at Austin
Electrical and Computer Engineering
Cockrell School of Engineering

Lecture 5: Outline

- **System architecture templates and components**
  - SoC computation and communication
- **System-level architecture models**
  - Modeling & abstraction levels
  - Virtual protoyping & virtual platform models
- **System-level design languages (SLDLs)**
  - Goals, requirements, principles
  - SLDL examples, SystemC language
  - Discrete-event model of computation & simulation

System Specification

- **Desired system behavior**
  - Pure functionality
  - Untimed
- **System constraints**
  - Non-functional requirements

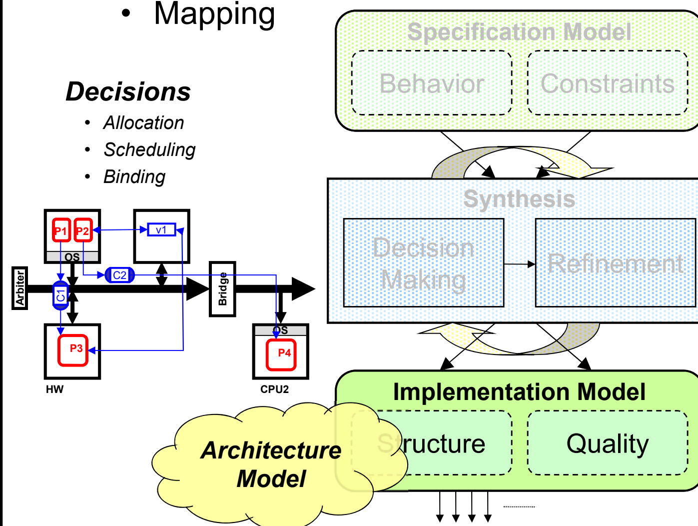


SoC Components

- **Computation**
  - CPUs
    - Operating systems
  - GPUs
  - Accelerators
  - FPGAs/CGRAs
  - Custom hardware
  - …
- **Storage**
  - Caches
  - Scratchpads
  - …
- **Communication**
  - Busses & bridges
  - Crossbars
  - Network-on-Chip (NoC)
    - Topology
    - Routing
  - …
- **I/O**
  - Bus & memory interfaces
  - Peripherals
    - Audio & video
    - …
  - …

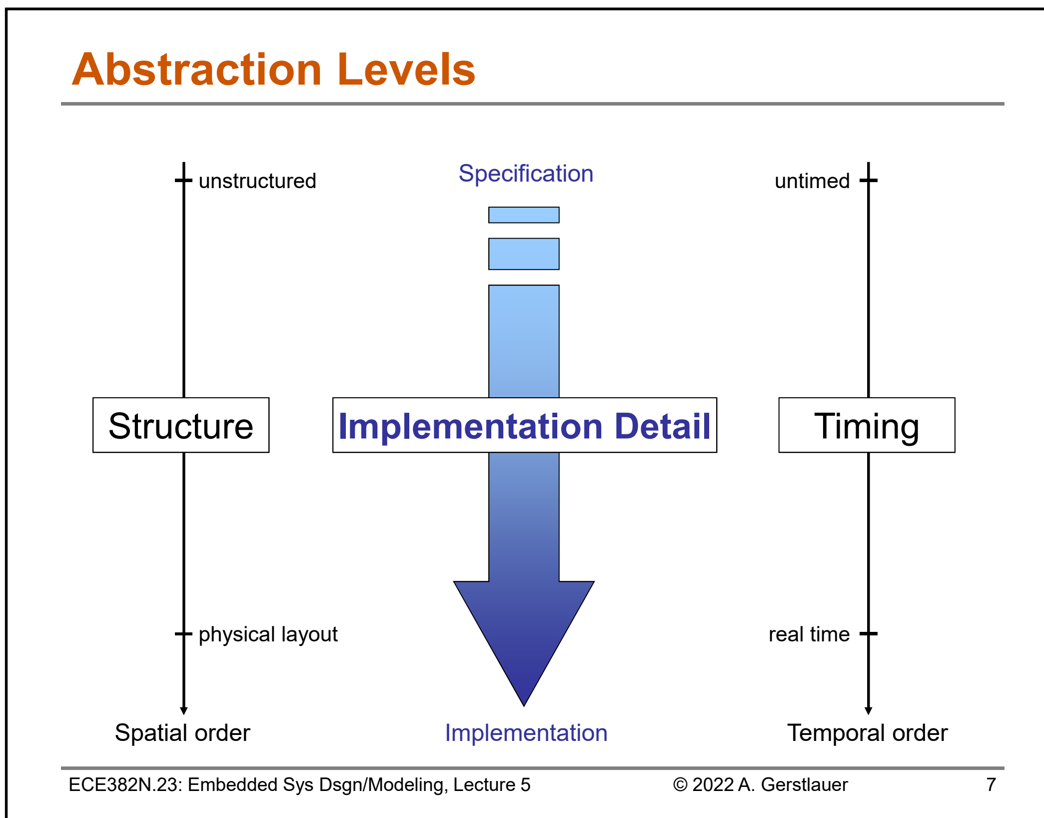
System Model

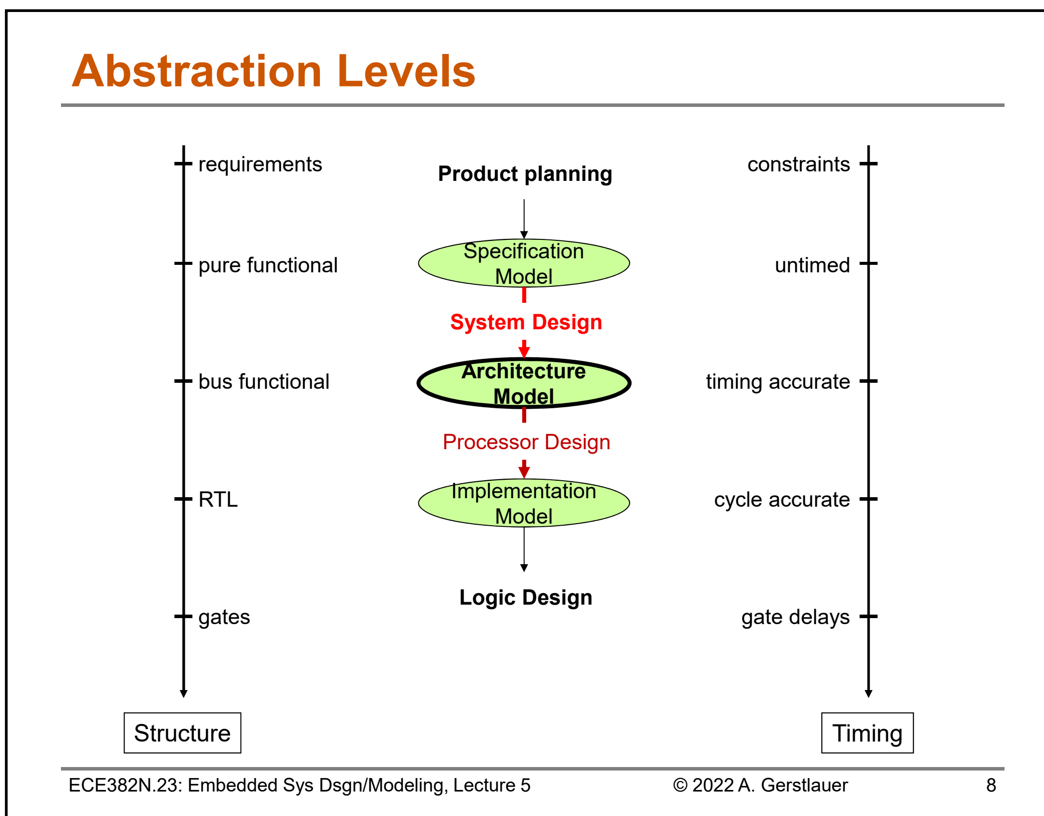
- **System definition**
  - Platform
  - Mapping
- **System quality**
  - Performance, power, …



System Modeling

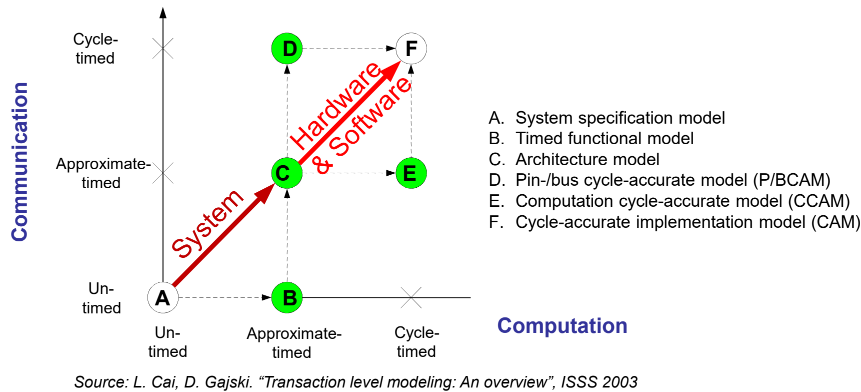
- ➢ **Basis of design flow and design automation**
  - ➢ Inputs and outputs of design steps
    - ➢ Capability to capture complex systems
    - ➢ Precise, complete and unambiguous
  - ➢ Models at varying levels of abstraction
    - ➢ Level and granularity of implementation detail
    - ➢ Speed vs. accuracy
- **Design models as an abstraction of a design instance**
  - Representation of some aspect of reality
    - – Virtual prototyping for validation through simulation or formal analysis
  - Specification for further implementation
    - – Describe desired functionality
  - ➢ Documentation & Specification (Simulation & Synthesis)
    - ➢ Abstraction to hide details that are not relevant or not yet known
    - ➢ Different parts of the model or different use cases for the same model


Abstraction Levels
unstructured
Specification
untimed
Structure
Implementation Detail
Timing
physical layout
real time
Spatial order
Implementation
Temporal order
ECE382N.23: Embedded Sys Dsgn/Modeling, Lecture 5
© 2022 A. Gerstlauer
7


Abstraction Levels
requirements
Product planning
constraints
pure functional
Specification Model
untimed
System Design
bus functional
Architecture Model
timing accurate
Processor Design
RTL
Implementation Model
cycle accurate
Logic Design
gates
gate delays
Structure
Timing
ECE382N.23: Embedded Sys Dsgn/Modeling, Lecture 5
© 2022 A. Gerstlauer
8

Computation vs. Communication

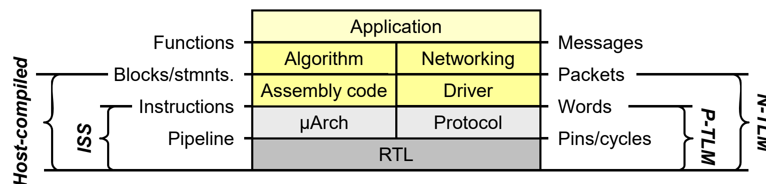
- **System design flow**
  - Path from model A to model F



Source: L. Cai, D. Gajski. "Transaction level modeling: An overview", ISSS 2003

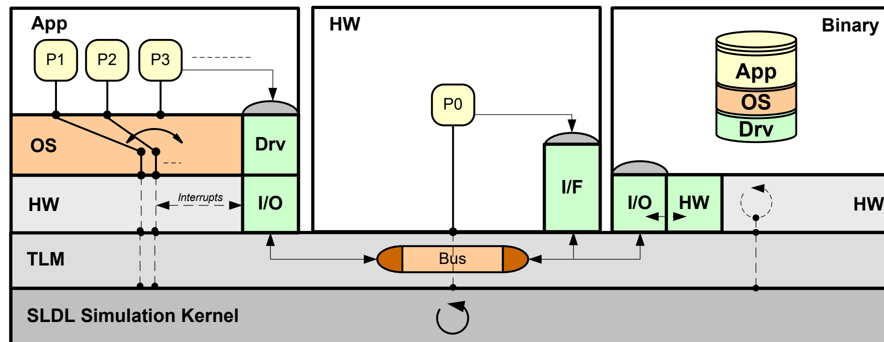
- **Design methodology and modeling flow**
  - Set of models and transformations between models

Modeling Levels



- **Host-compiled modeling of computation**
  - Abstract execution above instructions
  - Native execution of functionality
  - Back-annotation of timing, energy, …
  - Models of execution environment (OS & processor)
- **Transaction-level modeling (TLM) of communication**
  - Abstract transactions above pins and wires
  - Function calls for data transfer functionality
  - Back-annotation of timing, energy, …
  - Models of topology and glue logic

Virtual Platform Models



- **CPU model**
  - Source-level timing, energy, .. back-annotation
  - OS & processor models
- **Hardware/IP model**
  - Functional model
  - Timing, energy, … back-annotation
- **ISS model**
  - Cycle-accurate [GEM5]
  - Functional [QEMU] + timing, energy, … back-annotation
- **System-level design language (SLDL) & TLM backplane [SystemC]**

Lecture 5: Outline

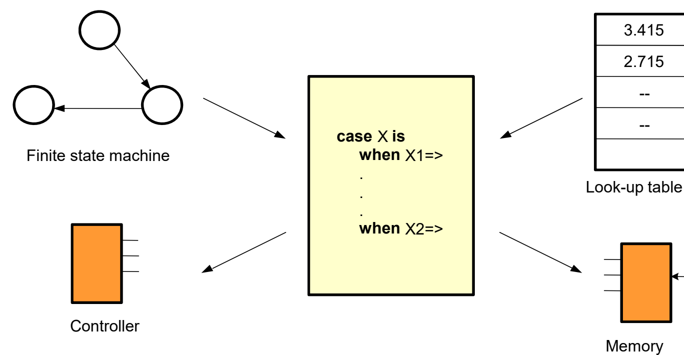
- ✓ System architecture templates and components
  - ✓ SoC computation and communication
- ✓ System-level architecture models
  - ✓ Modeling & abstraction levels
  - ✓ Virtual protoyping & virtual platform models
- **System-level design languages (SLDLs)**
  - Goals, requirements, principles
  - SLDL examples, SystemC language
  - Discrete-event model of computation & simulation

Languages

- **Capture models in machine-readable form**
  - At different levels of abstraction
  - Simulate & execute
  - Automatic refinement, analysis, synthesis
- **Syntax defines grammar**
  - Possible strings over an alphabet
  - Textual or graphical
- **Semantics defines meaning**
  - Execution/simulation model
  - Operational or denotational

Simulation vs. Synthesis Languages

- **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

Software Programming Languages

- **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]

➢ **No concurrency, structure or time**
  - ➢ Sequential behavior at task/block level
  - ➢ Implicit or explicit operation-level parallelism

Hardware 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 (block-level) concurrency and time**
  - ➢ Purely behavioral (task-level) concurrency?

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

System-Level Design Languages (SLDLs)

- **Requirements**

	C	C++	Java	VHDL	Verilog	SystemC	Statecharts	SpecCharts	SpecC
Behavioral hierarchy	○	○	○	○	○	○	●	●	●
Structural hierarchy	○	○	○	●	●	●	○	○	●
Concurrency	○	○	◐	●	●	●	●	●	●
Synchronization	○	○	◐	●	●	●	●	●	●
Exception handling	◐	●	●	○	●	○	◐	●	●
Timing	○	○	○	●	●	●	◐	◐	●
State transitions	○	○	○	○	○	○	●	●	●
Composite data types	●	●	●	●	◐	●	○	●	●

○ not supported ◐ partially supported ● supported

System-Level Design Languages (SLDLs)

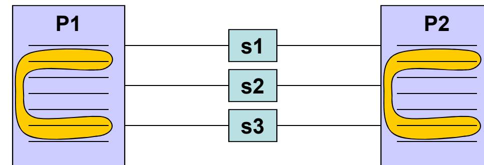
- **C/C++**
  - ANSI standard programming languages, software design
  - Traditionally used for system design because of practicality, availability
- **SpecC**
  - C extension
  - Developed at UC Irvine, standard by SpecC Technology Open Consortium (STOC)
- **SystemC**
  - C++ API and class library
  - Initially developed at UC Irvine, standard by Open SystemC Initiative (OSCI)
- **SystemVerilog**
  - Verilog with C extensions for testbench development
- **Matlab/Simulink**
  - Specification and simulation in engineering, algorithm design
- **Unified Modeling Language (UML)**
  - Requirements specification, no execution semantics, graphical
  - Extensible (meta-modeling), MARTE & SysML profiles for real-time/embedded/arch
- **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
- **…**

Separation of Concerns

- **Address separate issues independently**
  - Fundamental principle in modeling of systems
  - Fundamental principle in language design
- **System-level design**
  - Computation
    - Specified in processes or states
    - Implemented in processing & storage elements
  - Communication
    - Specified in channels
    - Implemented in interconnect
- ➢ **System-level design languages**
  - Orthogonal concepts & constructs

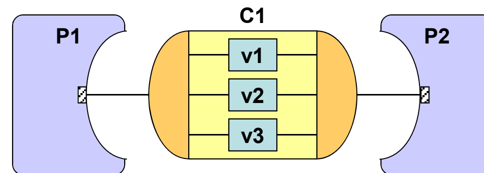
Computation vs. Communication

- **Traditional model**



  - Processes and signals
  - Mixture of computation and communication
  - Automatic replacement impossible

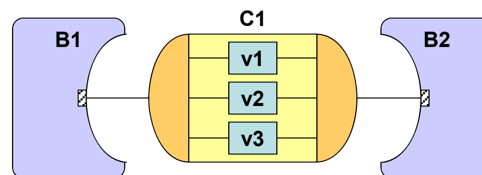
- **SLDL model**



  - Processes and channels
  - Separation of computation and communication
  - Plug-and-play

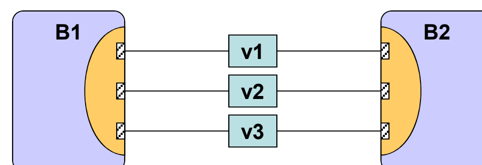
Computation vs. Communication

- **Protocol Inlining**
  - Specification model
  - Exploration model



    – Computation in behaviors
    – Communication in channels

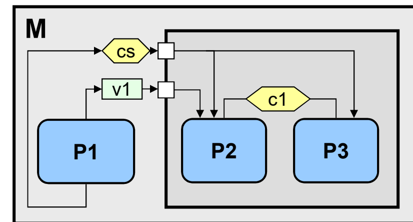
  - Implementation model



    – Channel disappears
    – Communication inlined into behaviors
    – Wires exposed

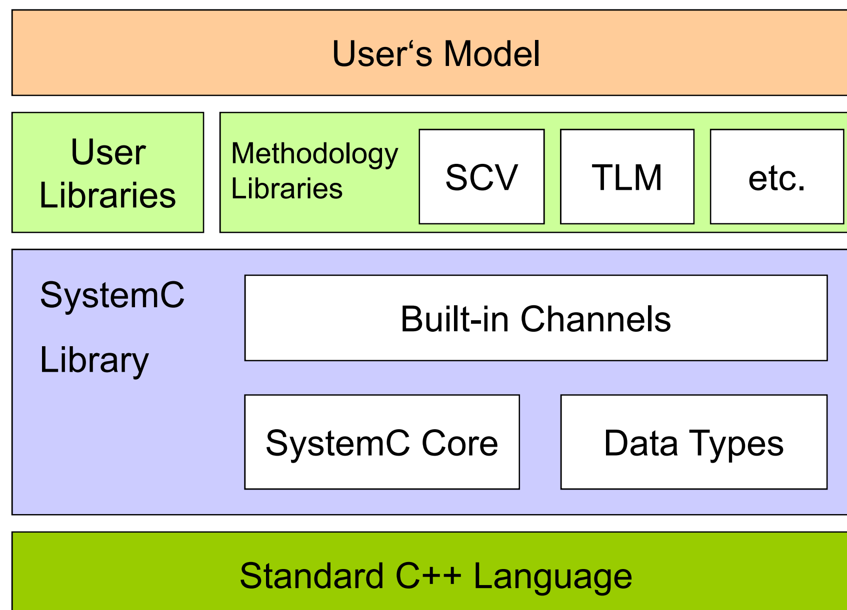
The SystemC Language

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



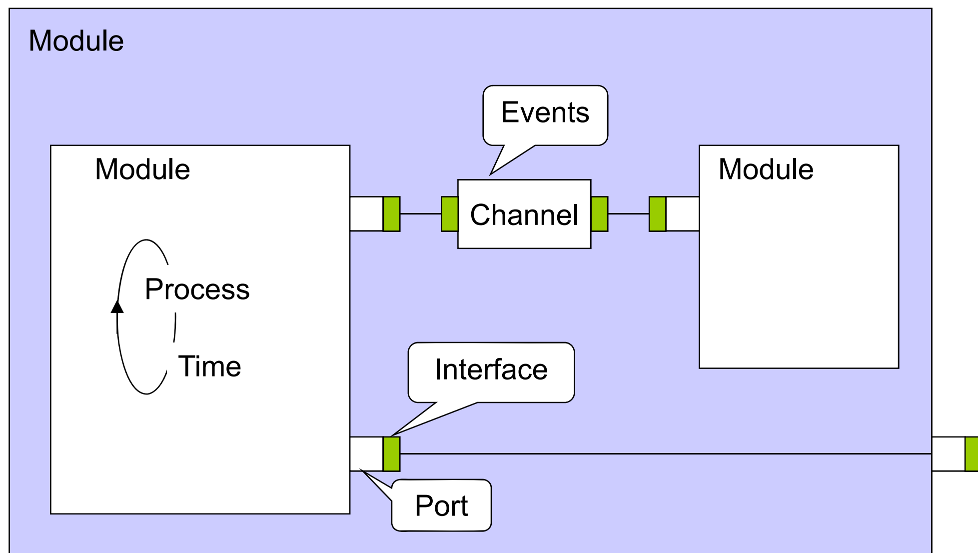
** SystemC 2.0*

SystemC Class Library Structure



Source: M. Radetzki, Univ. of Stuttgart

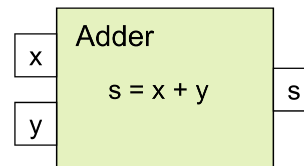
SystemC Structure



+ Bit-true data types

Source: M. Radetzki, Univ. of Stuttgart

Modules and Ports



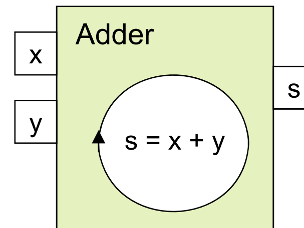

```
// file adder.h

#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
- `int` — port data type
- `};` — important (otherwise, strange error messages from C++ compiler)

Source: M. Radetzki

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_THREAD(add);
  }
};
```



function prototype

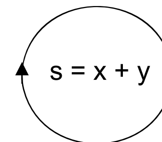
module constructor

function registered as a process

Source: M. Radetzki

Process Implementation

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

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

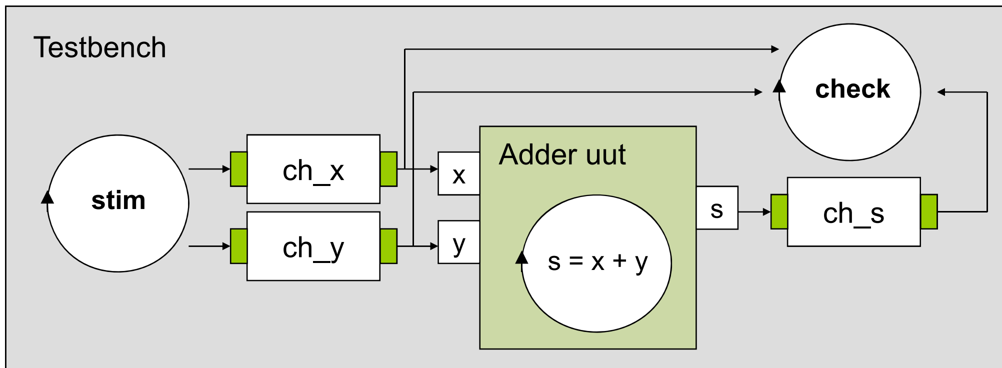


SC_THREAD is started only once, at the beginning of the simulation

SC_THREAD specifies activation by call to wait function

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
  ...
```

Source: M. Radetzki

SC_MODULE Structure

```
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)                  // constructor
  : 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
  }
};
```

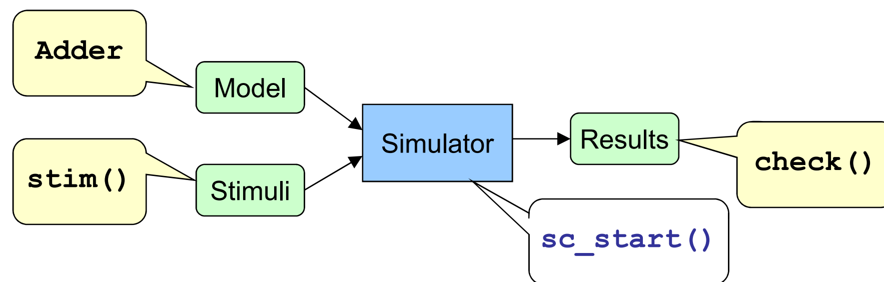
Source: M. Radetzki

Implementation of Module Processes

```
// file testbench.cpp
#include "testbench.h"

void Testbench::stim()  // SC_THREAD
{
  ch_x = 3; ch_y = 4;   // first stimulus
  wait(10, SC_NS);      // wait for 10 ns
  ch_x = 7; ch_y = 0;   // second stimulus
  wait(10, SC_NS);      // wait (no sensitivity!)
  ...                   // further stimuli
}

void Testbench::check() // SC_METHOD
{
  cout << ch_x << ch_y << ch_s << endl;  // debug output
  if( ch_s != ch_x+ch_y ) sc_stop();     // stop simulation
  else cout << "-> OK" << endl;
}
```

Source: M. Radetzki

Invoking the Simulation from `sc_main`




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

- Debugging: C++ debuggers (e.g. gdb/ddd)
- Tracing: `sc_trace` (VCD waveforms, view e.g. with `gtkwave`)

Source: M. Radetzki

SystemC Data Types

- **Bit & bit vector data types**
  - bool, sc_logic
  - sc_bv<N>, sc_lv<N>
- **Integer data types**
  - C/C++ types
  - sc_int<N>, sc_uint<N>
  - sc_bigint<N>, sc_biguint<N>
- **Fixed-point data types**
  - sc_fixed<…>, sc_ufixed<…>

Events

- **Declaration:** `sc_event <name>;`
- **Immediate triggering:** `<name>.notify();`
- **Waiting for occurrence:** `wait( <name> );`

```
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();
  ...
}
```

```
void Testbench::check()
{
  for(;;)
  {
    wait(new_stimulus);
    if( s == x+y )
      cout << "OK" << ...;
    else
      cout << "ERROR" << ...;
  }
}
```

Source: M. Radetzki

© 2022 A. Gerstlauer

Time

- `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:** `sc_time <name>(<magnitude>, <unit>);`
- **e.g.:** `sc_time delay(10, SC_NS);`
- **usage, e.g.:** `wait(delay);`
- **alternative:** `wait(10, SC_NS);`

Source: M. Radetzki

SystemC Built-In Channels

Channel	Matching Ports (Shortcuts)	Events
sc_signal<T>	sc_in<T> sc_out<T> sc_inout<T>	value changed
sc_buffer<T>	sc_in<T> sc_out<T> sc_inout<T>	value written (also if same as previous value)
sc_fifo<T>	sc_fifo_in<T> sc_fifo_out<T>	fifo contents changed
sc_semaphore	--	--
sc_mutex	--	--
sc_clock	sc_in<bool>	value changed

© 2022 A. Gerstlauer

System-Level Language Semantics

- **Language concepts (syntax)**
  - Behavioral and structural hierarchy
  - Concurrency and time
  - Synchronization and communication
  - Exception handling
  - State transitions
- **Language semantics needed to define the *meaning***
  - Semantics of execution for modeling, simulation, synthesis
  - Model of concurrency, time, synchronization, communication
    - Determinism vs. non-determinism
    - Atomicity
    - …

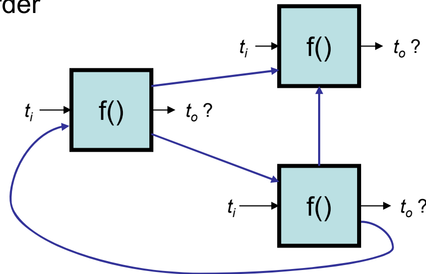
Source: R. Doemer, UC Irvine

Models of Time (Order)

- **Untimed**
  - Partial order based on causality only
    - No ordering in time, explicit dependencies only
    - ➢ Free of implementation (purely behavioral)
    - ➢ Specification & programming, Models of computation (Lecture 2-4)
- **Logical**
  - Discrete time, partial order
    - Discrete instants of time (time tags $t_0 < t_1 < \cdots t_k < \cdots$), nothing in between
    - Unspecified interleaving of events with same time tag
    - ➢ Freedom of implementation
    - ➢ Simulation & execution, design languages
- **Physical**
  - Continuous time, total order
    - Physical components naturally interleaved in (very fine) time
    - ➢ Differential equations, Hybrid models

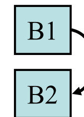
Recall: (Logical) Concurrency

- **Events/actions happening "at the same time"**
  - Undefined, unspecified or unknown order
    - Implementation/simulation determines actual interleaving
  - Communication/synchronization establishes causal order
  - Logical time establishes additional order
    - Partial order



➢ **Fundamental issues**
  ➢ Non-determinism
  ➢ Causality loops

Language Semantics



- **Motivating example 1**
  - Given:

```
void B::B1(void)
{
  x = 5;
};
```

```
void B::B2(void)
{
  x = 6;
};
```

```
SC_MODULE(B)
{
  int x;
  void B1(void);
  void B2(void);

  SC_TOR(B)
  {
    B1(); B2();
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: x = 6

Source: R. Doemer, UC Irvine

© 2022 A. Gerstlauer

Language Semantics

- **Motivating example 2**
  - Given:

```
void B::B1(void)
{
  x = 5;
};
```

```
void B::B2(void)
{
  x = 6;
};
```

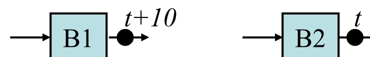
```
SC_MODULE(B)
{
  int x;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: The model is non-deterministic! (x may be 5, or 6)

Source: R. Doemer, UC Irvine

Language Semantics

- **Motivating example 3**
  - Given:




```
void B::B1(void)
{
  wait(10, SC_NS);
  x = 5;
};
```

```
void B::B2(void)
{
  x = 6;
};
```

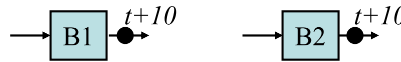
```
SC_MODULE(B)
{
  int x;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: x = 5

Source: R. Doemer, UC Irvine

Language Semantics

- **Motivating example 4**
  - Given:




```
void B::B1(void)
{
  wait(10, SC_NS);
  x = 5;
};
```

```
void B::B2(void)
{
  wait(10, SC_NS);
  x = 6;
};
```

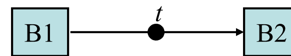
```
SC_MODULE(B)
{
  int x;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: The model is non-deterministic!
    (x may be 5, or 6)

Source: R. Doemer, UC Irvine

Language Semantics

- **Motivating example 5**
  - Given:




```
void B::B1(void)
{
  x = 5;
  e.notify();
};
```

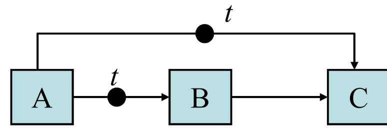
```
void B::B2(void)
{
  wait(e);
  x = 6;
};
```

```
SC_MODULE(B)
{
  int x;
  sc_event e;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: The model is non-deterministic!
    (x may be 6 or deadlock with x being 5)

Source: R. Doemer, UC Irvine

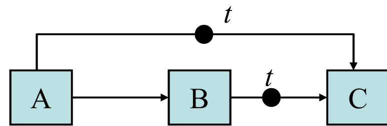
Simultaneous Events




```
void Top::B(void)
{
  void main(void){
    while (true) {
      wait(a);
      …
      b.notify();
    }
  }
};
```

```
void Top::C(void)
{
  void main(void) {
    while(true) {
      // a or b
      wait(a | b);
      …
    }
  }
};
```

Suppose B is invoked first

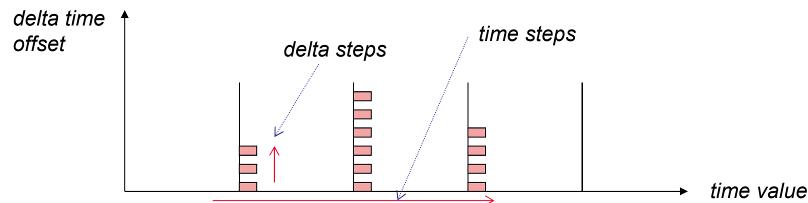


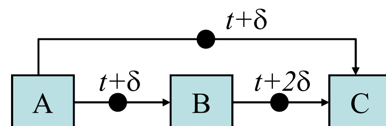
- **Depending on simulator**
  - Process C might be invoked once, observing both inputs in one invocation
  - Process C might be invoked twice, processing events one at a time
  - ➢ Non-deterministic order of event processing

Source: M. Jacome, UT Austin.

Delta (Superdense) Time

- **Two-level model of time**
  - Break each time instant into multiple delta steps
  - Each “zero” delay event results in a delta step
  - Delta time has zero delay but imposes semantic order



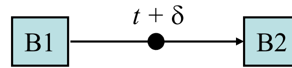


Answer: C is invoked twice

Source: M. Jacome, UT Austin.

Delta Semantics

- **Motivating example 5**
  - Given:




```
void B::B1(void)
{
  x = 5;
  e.notify(
   SC_ZERO_TIME);
};
```

```
void B::B2(void)
{
  wait(e);
  x = 6;
};
```

```
SC_MODULE(B)
{
  int x;
  sc_event e;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

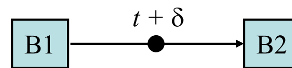
  - What is the value of x after the execution of B?
  - Answer: x = 6
    Delta notification is safe!

Source: R. Doemer, UC Irvine

ECE382N.23: Embedded Sys Dsgn/Modeling, Lecture 5 © 2022 A. Gerstlauer 47

Delta Semantics

- **Motivating example 6**
  - Given:




```
void B::B1(void)
{
  e.notify(
   SC_ZERO_TIME);
  x = 5;
};
```

```
void B::B2(void)
{
  wait(e);
  x = 6;
};
```

```
SC_MODULE(B)
{
  int x;
  sc_event e;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

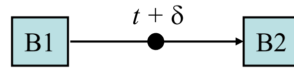
  - What is the value of x after the execution of B?
  - Answer: x = 6

Source: R. Doemer, UC Irvine

ECE382N.23: Embedded Sys Dsgn/Modeling, Lecture 5 © 2022 A. Gerstlauer 48

Delta Semantics

- **Motivating example 7**
  - Given:




```
void B::B1(void)
{
  e.notify(
   SC_ZERO_TIME);
  x = 4;
  e.notify(
   SC_ZERO_TIME);
  x = 5;
};
```

```
void B::B2(void)
{
  wait(e);
  x = 6;
  wait(e);
  x = 7;
};
```

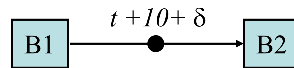
```
SC_MODULE(B)
{
  int x;
  sc_event e;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: B2 never terminates (x = 6)

Source: R. Doemer, UC Irvine

Delta Semantics

- **Motivating example 8**
  - Given:




```
void B::B1(void)
{
  wait(10, SC_NS);
  x = 5;
  e.notify();
};
```

```
void B::B2(void)
{
  wait(e);
  x = 6;
};
```

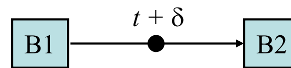
```
SC_MODULE(B)
{
  int x;
  sc_event e;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: x = 6

Source: R. Doemer, UC Irvine

Delta Semantics

- **Motivating example 9**
  - Given:




```
void B::B1(void)
{
  x = 5;
  e.notify(
    SC_ZERO_TIME);
};
```

```
void B::B2(void)
{
  wait(10, SC_NS);
  wait(e);
  x = 6;
};
```

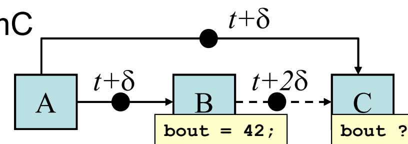
```
SC_MODULE(B)
{
  int x;
  sc_event e;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

  - What is the value of x after the execution of B?
  - Answer: B2 never terminates!
    (the event is lost)

Source: R. Doemer, UC Irvine

Delta Semantics

- **Resolve some non-determinism**
  - As long as each output has “unique” delta delay
  - Often not the case, e.g. in SystemC
    - ➢ Example 2, Example 4



➢ **Ambiguity still exists**
  - Shared resource/variable accesses in same delta cycle
    - ➢ With B->C sharing: B or C first? Undefined order, non-deterministic
    - ➢ Example on slide 56 with variables: value of `bout` on first invocation of C?

➢ **Alternative semantics based on precedence analysis**
  - Graph is executed in topologically sorted order [Ptolemy]
    - ➢ C only invoked once, with B->C dependency: B invoked first
  - Potentially still ambiguous
    - ➢ If there is no B->C dependency: B or C first?
    - ➢ Only matters if there are other side effects (`printf()`)

Source: M. Jacome, UT Austin.

© 2022 A. Gerstlauer

Deterministic Communication

- **Given**

```
void B::B1(void)
{
  x = 5;
  e.notify(
   SC_ZERO_TIME);
};
```

```
void B::B2(void)
{
  int y, z;

  y = x;
  wait(e);
  z = x;
};
```

```
SC_MODULE(B)
{
  int x = 0;
  sc_event e;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

- What is the value of (y,z) at the end of execution?
- Answer: z = 5, y is non-deterministic (0 or 5)

Deterministic Communication

- **Delta-semantics for variables [VDHL, Verilog, SystemC]**
  - Signals combine event with current/new value updates

```
void B::B1(void)
{
  x = 5;
  ~~x.default_event().~~
  ~~notify();~~ // implicit
};
```

```
void B::B2(void)
{
  int y, z;

  y = x;
  wait(x.default_event());
  z = x;
};
```

```
SC_MODULE(B)
{
  sc_signal<int> x = 0;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

  - What is the value of (y,z) at the end of execution?
  - Answer: z = 5, y = 0
- **Resolves concurrent read-write**
  - Concurrent writes still a problem
    - Non-deterministic [SpecC], resolution functions [VHDL, Verilog]

© 2022 A. Gerstlauer

Deterministic Communication

- **Avoid event loss by combining event with flag**

```
void B::B1(void)
{
 …
  wait(D2, SC_NS);
  …
  flag = true;
  x = 5;
  e.notify();
  …
};
```

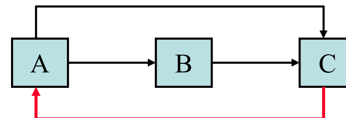
```
void B::B2(void)
{
  …
  wait(D2, SC_NS);
  …
  if(!flag) wait(e);
  flag = false;
  x = 6;
  …
};
```

```
SC_MODULE(B)
{
  int x = 0;
  bool f = false
  sc_event e;
  void B1(void);
  void B2(void);

  SC_CTOR(B) {
    SC_THREAD(B1);
    SC_THREAD(B2);
  }
};
```

  - What is the value of x at the end of execution?
  - Answer: actually, may end up being x = 5, why?
    - Race conditions, interleaving of B1 and B2 if D1 == D2
- **Enforce atomicity**
  - SystemC dictates that all code is atomic!

Zero-Delay Feedback Loops

- **Causality loop**
  - Where to start & stop?
  - Progress?

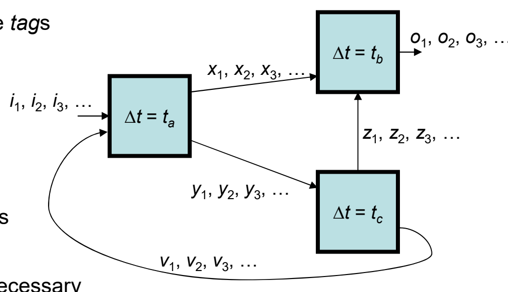


- **Reject zero-delay cycles**
  - Forbid all zero delay (every Δt must be strictly > 0) [DEVS]
  - Detect (compile error on zero cycle)
- **Delta cycle semantics**
  - Oscillate with no time progress [Zeno machine/model]
- **Approaches based on topological sorting**
  - Annotate feedback arcs to “break” for ordering purposes
    - Insert explicit δ delay block [Ptolemy]
    - Potentially same oscillation without progress

Source: M. Jacome, UT Austin.

Discrete Event (DE) Model

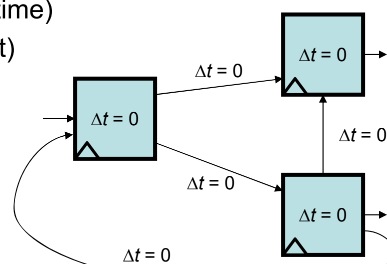
- **General, universal model for system simulation**
  - Hardware [VHDL, Verilog], system [SpecC, SystemC], network [OMNet]
- **Formal, generic discrete time model**
  - Signals = globally ordered streams of events
    - Event e_i = (*value*, *tag*), discrete time *tag*s
  - Asynchronous event processing
    - Execute block on input event
    - Process input and generate output events with *tag*+Δt
  - Flexible
    - Multi-scale, arbitrary dynamic delays
  - Efficient
    - Event-driven, only evaluate when necessary



- **Synthesis challenges**
  - Semantics: simultaneous events (non-determinism), zero-delay cycles
  - Implementation: global order (maintain global notion of time) [PTIDES]

Synchronous Reactive (SR) Model

- **Synchronous hypothesis**
  - Sequence of discrete lock steps (ticks of logical clock)
  - Reactions are instantaneous (zero time)
  - Events are simultaneous (broadcast)
  - Deterministic, static verifiable (independent of actual delays)
    - Precedence analysis, topological sort
    - No arbitrary shared variables



- **Synthesis challenges:**
  - Semantics: causality loops, conflicts?
  - Implementation: broadcast events, global clock
- **Synchronous languages**
  - Imperative (control) [Esterel] or declarative (data) [Lustre] style
  - Reject cycles [Lustre] or require unique fixed-point [Esterel]
  - Hardware (FSMs) or software (safety critical) compilers

DE vs. SR

- **Discrete-event (DE) w/ delta cycles**
  - Can accurately model sub-cycle timing & glitching effects
  - Non-determinism with shared variables/resources
  - Issues with oscillation in case of cycles
- **Discrete-event (DE) w/ topological sort**
  - Deterministic if there are no other side effects
  - Requires explicit modifications to break cycles
- **Synchronous-reactive (SR)**
  - Consistent & fully deterministic
  - Combines toplogical sorting with fixed-point
  - But restricted model of time (global ticks only)

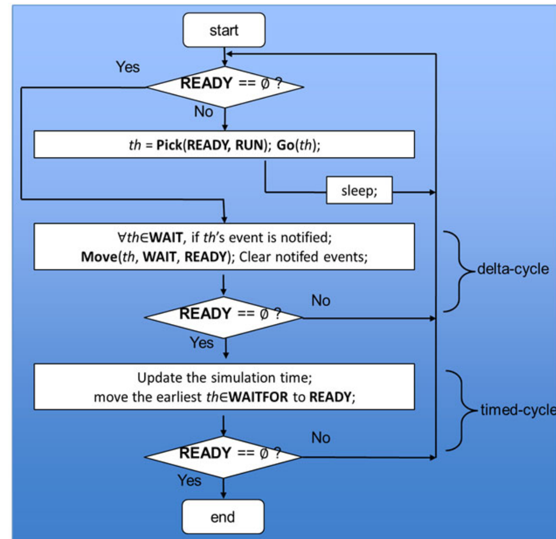
Simulation Semantics

- **Abstract discrete-event simulation algorithm**
  - ⇒ set of valid implementations
  - ⇒ not general (possibly incomplete)
- **Definitions:**
  - At any time, each thread *t* is in one of the following sets:
    - **READY**: set of threads ready to execute (initially root thread)
    - **WAIT**: set of threads suspended by `wait` (initially Ø)
    - **WAITFOR**: set of threads suspended by `waitfor` (initially Ø)
  - Notified events are stored in a set **N**
    - `e1.notify()` adds event `e1` to **N**
    - `wait(e1)` will wakeup when `e1` is in **N**
    - Consumption of event e means event e is taken out of **N**
    - Expiration of notified events means **N** is set to Ø

Source: R. Doemer, UC Irvine

Discrete Event Simulation (DES)

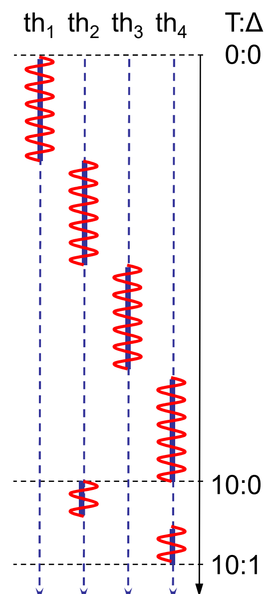
- **Traditional DE simulation algorithm**
  - Threads managed in READY queue
  - Scheduler picks a *single* thread and executes it
  - Time advances
    - In delta-cycle
    - In timed-cycle



Source: R. Doemer, UC Irvine

Discrete Event Simulation (DES)

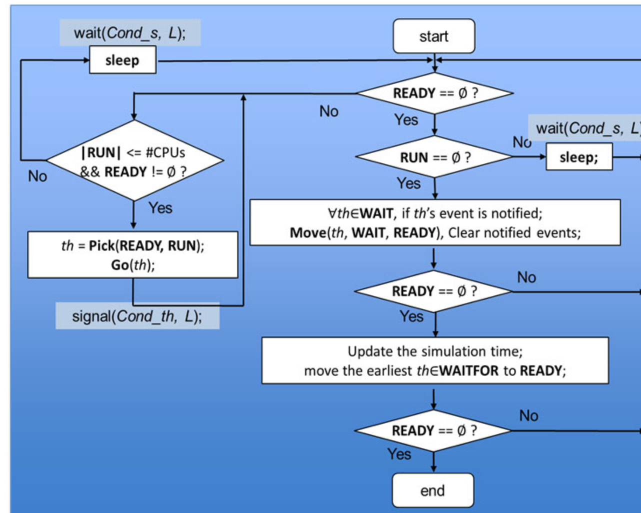
- **Traditional DES**
  - Concurrent threads of execution
  - Managed by a central scheduler
  - Driven by events and time advances
    - Delta-cycle
    - Time-cycle
  - ➢ Partial temporal order with barriers
- **Reference simulators**
  - Cooperative multi-threading
  - ➢ A single thread is active at any time!
  - ➢ Cannot exploit multiple parallel cores
  - Example: SystemC



Source: R. Doemer, UC Irvine

Parallel Discrete Event Simulation (PDES)

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

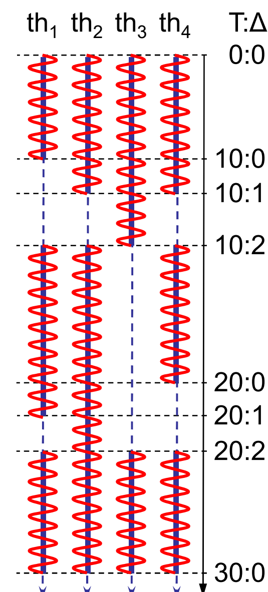


Source: R. Doemer, UC Irvine

Parallel Discrete Event Simulation (PDES)

- **Parallel DES**
  - Threads execute in parallel *iff*
    - in the same delta cycle, *and*
    - in the same time cycle
  - ➢ Significant speed up!
  - ➢ But: Amdahl's Law still applies!
    - Cycle bounds are absolute barriers

➢ **Aggressive PDES**
  - ➢ Optimistic
    - ➢ Let threads run ahead in time
    - ➢ Rollback if conflict detected (event in the past)
  - ➢ Conservative
    - ➢ Only run ahead if guaranteed no conflicts
    - ➢ E.g. static code/dependency analysis



Source: R. Doemer, UC Irvine

Lecture 5: Summary

- **System architecture modeling**
  - Basis of any design flow
  - At varying levels of abstraction
  - Computation & communication
- **System-level design languages**
  - Capture system models in executable form
  - Structural concurrency, synchronization & time
  - Discrete-event model of computation