System Design and Methodology / Embedded Systems Design

II. Models of Computation and Modeling Languages

TDTS07/TDDI08
VT 2026

Ahmed Rezine

(Based on material by Petru Eles and Soheil Samii)

Institutionen för datavetenskap (IDA)
Linköpings universitet

Models of Computation and Modeling Languages

1. System Specification
2. System Modeling and Formal Models
3. Models of Computation: What's that?
4. Concurrency
5. Communication & Synchronisation
6. Common Models of Computation

From Specifications to Implementations

- Specification: A description of basic requirements and properties of a system

  - The designer gets a *specification* as an input and, finally, has to produce an *implementation*.

    This is usually done as a sequence of *refinement steps*.

  - Specifications can be:
    - informal (natural language)
    - more detailed and unambiguous (based on a formal notation)

System Specifications

- A specification captures:
  - The basic required behaviour of the system
    - E.g. as a relation between inputs and outputs
  - Other (non-functional) requirements
    - time constraints
    - power/energy constraints
    - safety requirements
    - environmental aspects
    - cost, weight, etc.

System Model

- As an early step in the design flow, a *system model* is produced (you remember the design flow!).

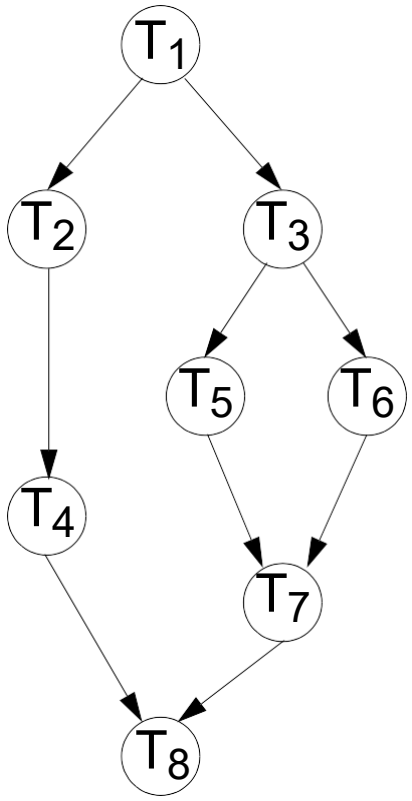
- The model is a description of certain aspects/properties of the system. Models are abstract, in the sense that they omit details and concentrate on aspects that are significant for the design process.

System Model

- As an early step in the design flow, a *system model* is produced (you remember the design flow!).

- The model is a description of certain aspects/properties of the system. Models are abstract, in the sense that they omit details and concentrate on aspects that are significant for the design process.

You remember our task graph example!



System Model

- Models are formulated using *modeling languages*

- **Modeling language:**
  - well-suited to expressing the basic system properties and basic aspects of system behaviour in a succinct and clear manner
  - lends itself well to the, preferably automatic, checking of requirements and synthesis of implementations.

- Depending on the particularities of the system, an adequate modeling language has to be chosen.

  The language has to contain the appropriate language constructs in order to express the system's functionality and requirements.

System Model

- Modeling Languages can be
  - graphical
  - textual

- Modeling languages can be
  - “ordinary” programming languages (C, C++)
  - hardware description languages (VHDL, Verilog)
  - languages specialised for modeling of systems in particular areas, and with particular features;
    they are often based on particular *models of computation*.

System Model

What do we want to do with the model of an embedded system?

System Model

What do we want to do with the model of an embedded system?

1. To validate the system description in order to check that the specified functionality is the desired one and the requirements are stated correctly:
   - by formal verification
   - by simulation

2. To synthesise efficient implementations

Semantics of System Models

We would like modeling languages to have well defined semantics $\Rightarrow$ models are unambiguous.

- The *semantics* is the set of rules which associate a meaning to *syntactical* constructs (combination of symbols) of the language.

- The semantics of the language is based on the underlying *model of computation*.

  It depends on this underlying model of computation what kind of sys- tems can be described with the language.

  The model of computation decides on the *expressiveness* of the language.

Semantics of System Models

Do we want large expressiveness (we can describe anything we want)? Not exactly!

- Large expressive power: imperative model (e.g. unrestricted use of C or Java):
  - Can specify “anything”.
  - No formal reasoning possible (or extremely complex).

- Limited expressive power, based on well chosen computation model:
  - Only particular systems can be specified.
  - Formal reasoning is possible.
  - Efficient (possibly automatic) synthesis.

Language L1

```
process P1
{   .................
    send m to P2;
    ..................}

process P2
{   .................
    receive m from P1;
    ..................}
```

Language L1

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

Synchronous:
send and *receive* blocking; P1 and P2 are waiting for each other to handshake and hand over the message:
- No buffering needed.
- P1 and P2 run at the same rate in lockstep.

Language L1

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

Synchronous:
send and *receive* blocking; P1 and P2 are waiting for each other to handshake and hand over the message:
- No buffering needed.
- P1 and P2 run at the same rate in lockstep.

Language L2

```
module P1
{   .....
    m!P2;
    ..... }

module P2
{   .....
    m?P1;
    ..... }
```

Language L1

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

Synchronous:
send and *receive* blocking; P1 and P2 are waiting for each other to handshake and hand over the message:
- No buffering needed.
- P1 and P2 run at the same rate in lockstep.

Language L2

```
module P1
{   .....
    m!P2;
    ..... }

module P2
{   .....
    m?P1;
    ..... }
```

Language L1

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

Synchronous:
send and *receive* blocking; P1 and P2 are waiting for each other to handshake and hand over the message:
- No buffering needed.
- P1 and P2 run at the same rate in lockstep.

Language L2

```
module P1
{   .....
    m!P2;
    ..... }

module P2
{   .....
    m?P1;
    ..... }
```

Language L3

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

Language L1

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

Synchronous:
send and *receive* blocking; P1 and P2 are waiting for each other to handshake and hand over the message:
- No buffering needed.
- P1 and P2 run at the same rate in lockstep.

Language L2

```
module P1
{   .....
    m!P2;
    ..... }

module P2
{   .....
    m?P1;
    ..... }
```

Language L3

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

Asynchronous:
receive blocking but *send* not; P1 and P2 are not waiting for each other; P2 only waits if there is no message available:
- Buffering is needed!
- P1 and P2 can run at different rates.

Language L1

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

Synchronous:
send and *receive* blocking; P1 and P2 are waiting for each other to handshake and hand over the message:
- No buffering needed.
- P1 and P2 run at the same rate in lockstep.

Language L2

```
module P1
{   .....
    m!P2;
    ..... }

module P2
{   .....
    m?P1;
    ..... }
```

Language L3

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

Asynchronous:
receive blocking but *send* not; P1 and P2 are not waiting for each other; P2 only waits if there is no message available:
- Buffering is needed!
- P1 and P2 can run at different rates.

Language L4

```
module P1
{   .....
    m!P2;
    ..... }

module P2
{   .....
    m?P1;
    ..... }
```

Language L1

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

Synchronous:
send and *receive* blocking; P1 and P2 are waiting for each other to handshake and hand over the message:
- No buffering needed.
- P1 and P2 run at the same rate in lockstep.

Language L2

```
module P1
{   .....
    m!P2;
    ..... }

module P2
{   .....
    m?P1;
    ..... }
```

Language L3

```
process P1
{   .....
    send m to P2;
    ..... }

process P2
{   .....
    receive m from P1;
    ..... }
```

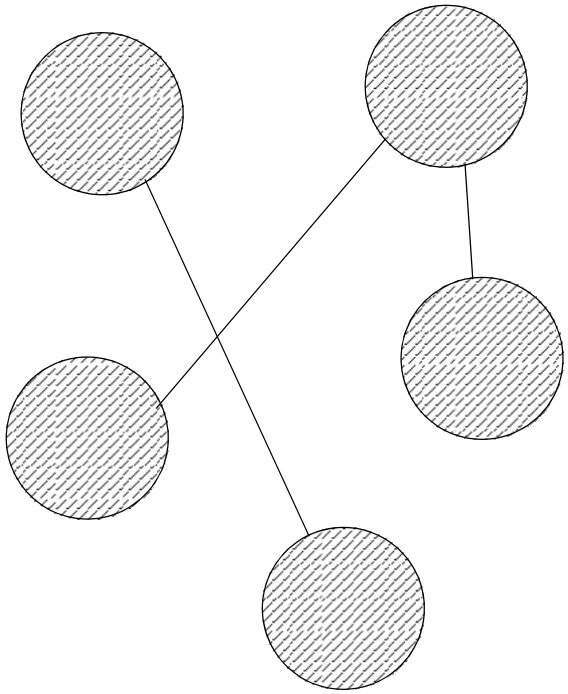
Asynchronous:
receive blocking but *send* not; P1 and P2 are not waiting for each other; P2 only waits if there is no message available:
- Buffering is needed!
- P1 and P2 can run at different rates.

Language L4

```
module P1
{   .....
    m!P2;
    ..... }

module P2
{   .....
    m?P1;
    ..... }
```

Models of Computation

The *model of computation* deals with the set of theoretical choices that build the execution model of the language.

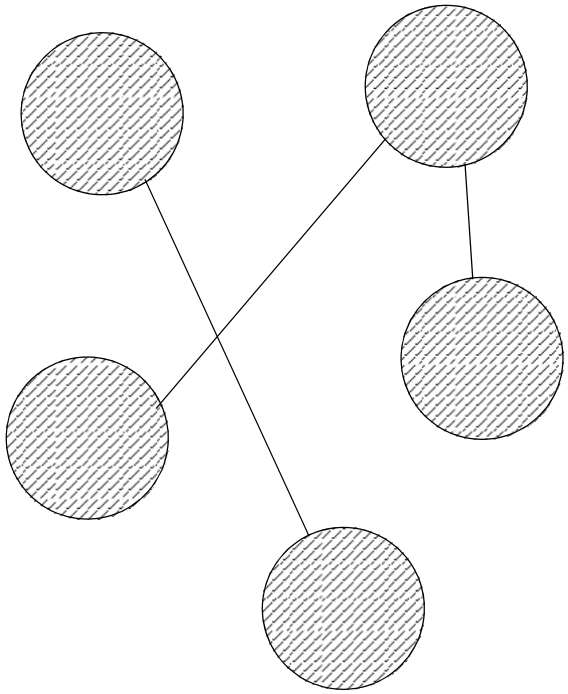


- A design is represented as a set of components, which can be considered as isolated monolithic modules (often called *processes* or *tasks*), interacting with each other and with the environment.

  The *model of computation* defines the behavior and interaction mechanisms of these modules.

Models of Computation

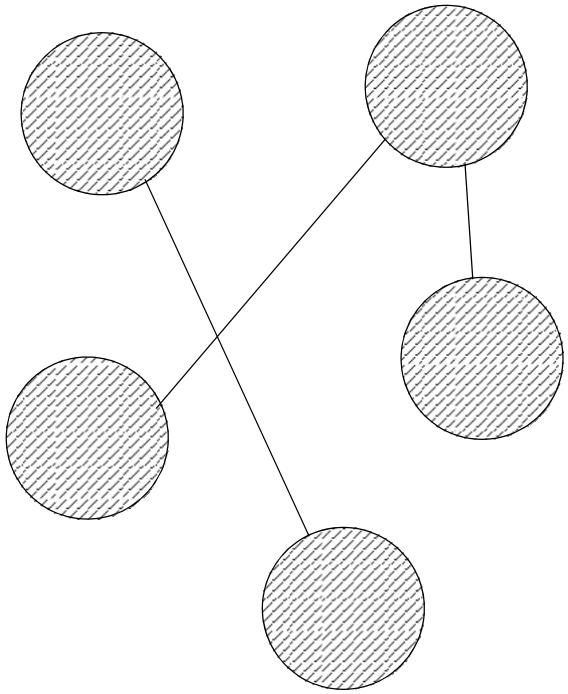
The *model of computation* deals with the set of theoretical choices that build the execution model of the language.



- Models of computation usually refer to:
  - how each module (process or task) performs internal computation
  - how they transfer information between them
  - how they relate in terms of concurrency

Models of Computation

The *model of computation* deals with the set of theoretical choices that build the execution model of the language.



- Models of computation usually refer to:
  - how each module (process or task) performs internal computation
  - how they transfer information between them
  - how they relate in terms of concurrency
- Some models of computation do not refer to aspects related to the internal computation of the modules, but only to module interaction and concurrency.
- The main aspects we are interested in:
  - Concurrency
  - Communication&Synchronization
  - Time
  - Hierarchy

Concurrency

- A system consists of several activities (processes or tasks) which potentially can be executed in parallel. Such activities are called *concurrent*.

How to express concurrency?

- This is one aspect in which models of computation differ!
  - Data-driven concurrency
  - Control-driven concurrency

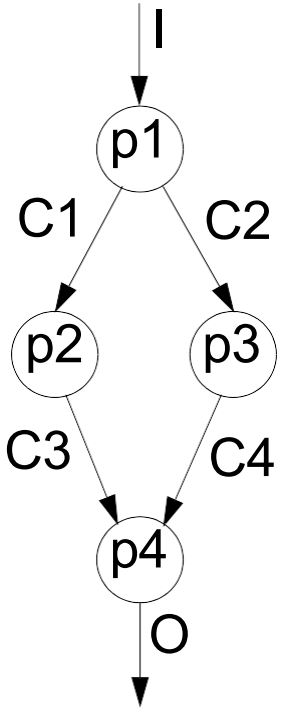
Data-driven Concurrency

The system is modelled as a set of processes without any *explicit* specification of the ordering of executions.

The execution order of processes (and, implicitly, the potential of parallelism) is fixed solely by data dependencies

- Appropriate e.g. for many DSP applications

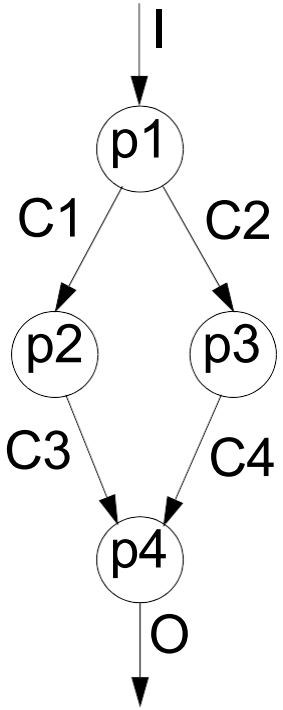
Data-driven Concurrency



Data-driven Concurrency

```
Process p1( in int a, out int x, out int y) {
.............
}

Process p2( in int a, out int x) {
.............
}

Process p3( in int a, out int x) {
.............
}

Process p4( in int a, in int b, out int x) {
.............
}

channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p3(C2, C4);
p4(C3, C4, O);
```

It doesn't matter in which order I have written this.



Control-driven Concurrency

- The execution order of processes is given explicitly in the system model.
- Explicit constructs are used to specify sequential execution and concurrency.

Control-driven Concurrency

```
module p1:
.....
end module

module p2:
.....
end module

module p3:
.....
end module

module p4:
.....
end module

run p1;
[ run p2 || run p3];
run p4
```

Here, the order in which we write is essential!

- The execution order of processes is given explicitly in the system model.
- Explicit constructs are used to specify sequential execution and concurrency.

Communication

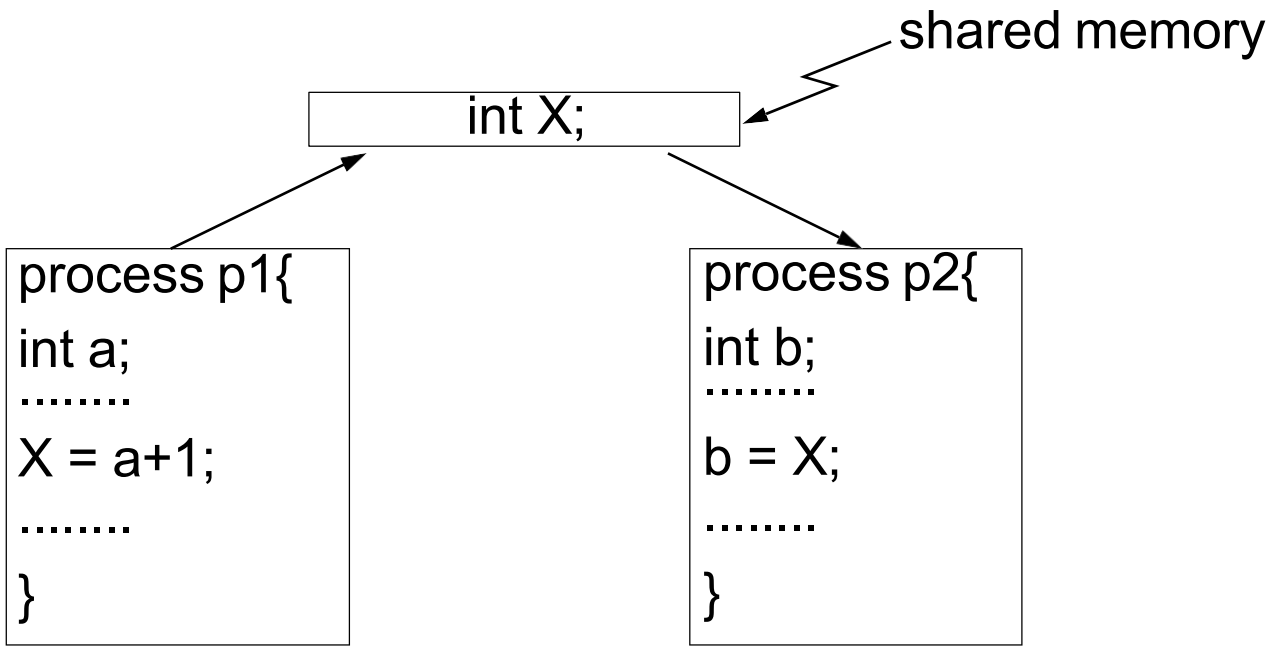
- Processes have to communicate in order to exchange information.

  Various communication mechanisms are used in different computation models:

  - shared memory
  - message passing
    - blocking
    - non-blocking

Shared Memory Communication

- Each sending process writes to shared variables which can be read by a receiving process.



Private variables:

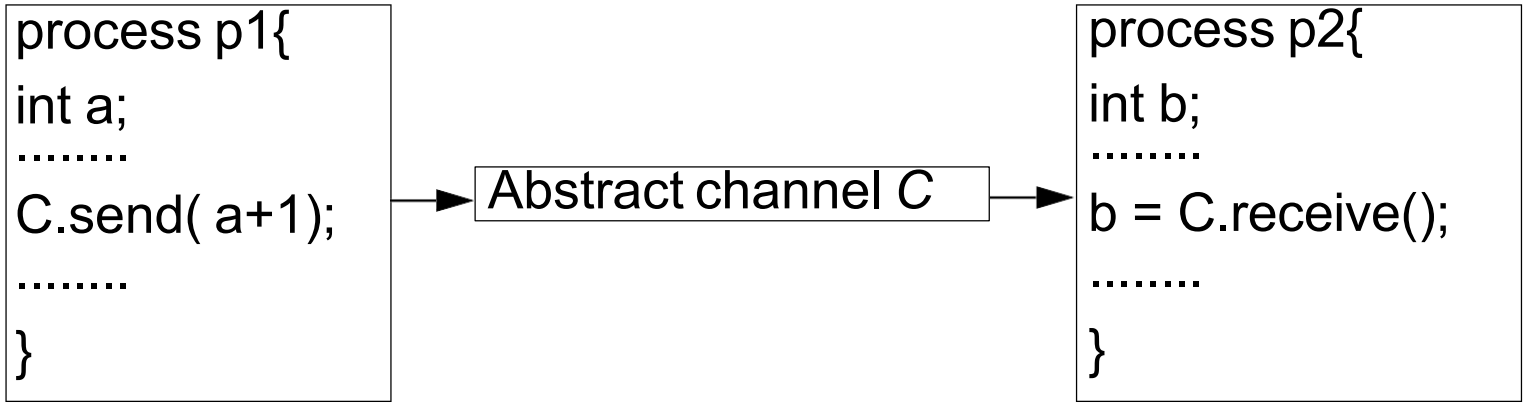
- *a*: local to p1
- *b*: local to p2

Shared variable:

- X

Message-passing Communication

- Data (messages) are passed over an abstract communication medium called *channel*.



- This communication model is adequate for modeling of distributed systems.

Message-passing Communication

- **Blocking communication**

  A process which communicates over the channel *blocks* itself (suspends) until the other process is ready for the data transfer.

  The two processes have to synchronize before data transfer can be initiated.

Message-passing Communication

- **Non-blocking communication**

  Processes do not have to synchronize for communication!

  Additional storage (buffer) has to be associated with the channel if no messages are to be lost!

  - **The sending process** places the message into the buffer and continues execution.

    The receiving process reads the message from the channel whenever it is ready to do it.

Synchronization

- Synchronization cannot be separated from communication.

  Any interaction between processes implies a certain degree of communication and synchronization.

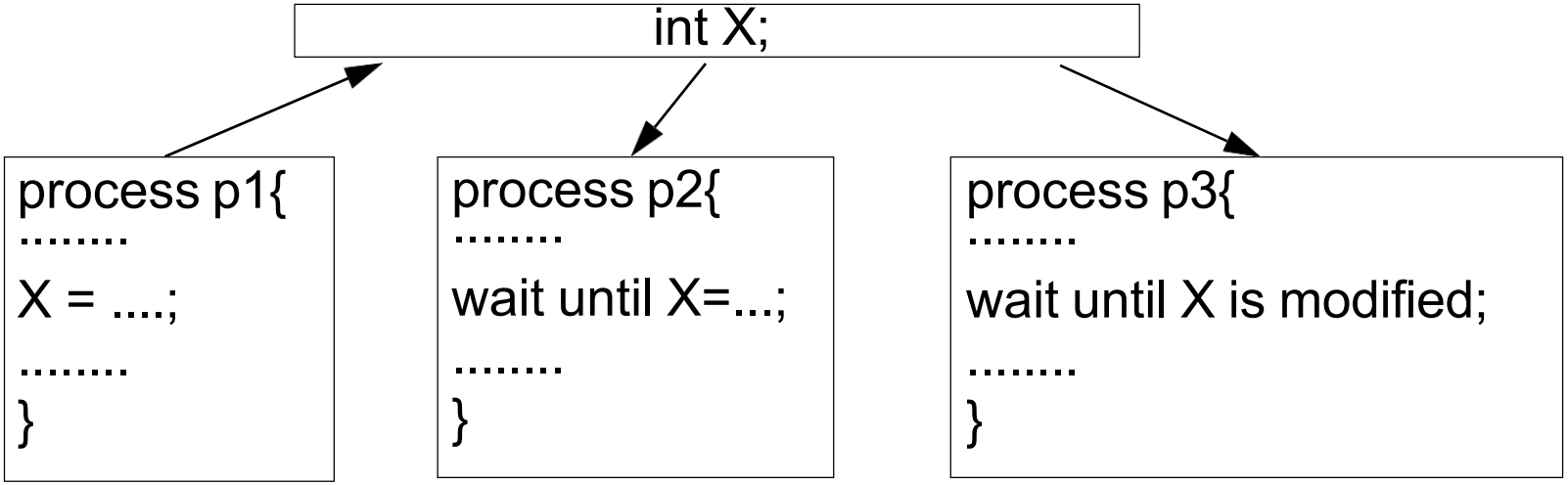
- Synchronization:One process is suspended until another one reaches a certain point in its execution.
  - Control-dependent synchronization
  - Data-dependent synchronization

Control-dependent Synchronization: Example

```
module p1:
.....
end module

module p2:
.....
end module

module p3:
.....
end module

module p4:
.....
end module

run p1;
[ run p2 || run p3];
run p4
```

- With control-dependent synchronization the con- trol structure is responsible for synchronization
- In the example we have several synchronization points specified:
  - between completion of p1 and starting of p2 and p3;
  - between completion of p2 and p3, and starting of p4.

Data-dependent Synchronization: Example



And don’t forget: *Time*!

- How is time handled?

This makes a great difference between models of computation!

Common Models of Computation

In this course, we will analyze some of the models of computation commonly used to describe embedded systems:

- Dataflow Models
- Petri Nets
- Discrete Event
- (Synchronous) Finite State Machines
- Globally Asynchronous Locally Synchronous Models
- Timed & Hybrid Automata