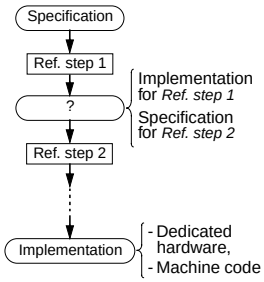
Models of Computation and Specification Languages

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

Specifications and Implementations

☞ Specification: A description of the behavior and other 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) or
  - more detailed and unambiguous (e.g. a C program)



System Specifications

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

System Specifications (cont'd)

- Specifications can be formulated in
  - natural language
  - using *specification languages*
- Specification language:
  - well-suited to expressing the basic system properties and basic aspects of system behaviour in a succint 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 specification language has to be chosen.
The language has to contain the appropriate language constructs in order to express the systems' functionality and requirements.

System Specifications (cont'd)

- Specification Languages can be
  - graphical
  - textual
- Specification languages can be
  - "ordinary" programming languages (C, C++)
  - hardware description languages (VHDL, Verilog)
  - languages specialised for specification of systems in particular areas, and with particular features; they are often based on particular *models of computation*.

System Specifications (cont'd)

What do we want to do with the specification 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

Formal Models and System Specifications

☞ We would like specification languages to have well defined semantics ⇒ specifications are unambiguous.

- The *semantics* is the set of rules which associate a meaning (interpretation) 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 systems can be described with the language.
  The model of computation decides on the *expressiveness* of the language.

Formal Models and System Specifications (cont'd)

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.

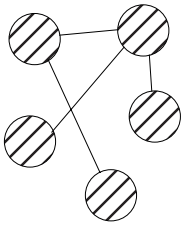
Models of Computation

☞ 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 (cont'd)



- Models of computation usually refer to:
  - how each module (process or task) performs internal computation
  - how the modules 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.

Models of Computation (cont'd)

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 computational models differ!

- Data-driven concurrency
- Control-driven concurrency

Data-driven Concurrency

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

⇩

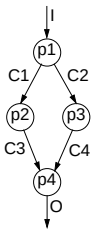
The execution order of processes (and, implicitly, the potential of parallelism) is fixed solely by data dependencies

- Typical for many DSP applications

Data-driven Concurrency (cont'd)

```
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 specification.

☞ Explicit constructs are used to specify sequential execution and concurrency.

Control-driven Concurrency (cont'd)

☞ This example is in ESTEREL

```
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!

- p1 is started first and has to finish before the starting of p2 and p3;
- p2 and p3 are started in parallel;
- both p2 and p3 have to finish before p4 is started.

Communication

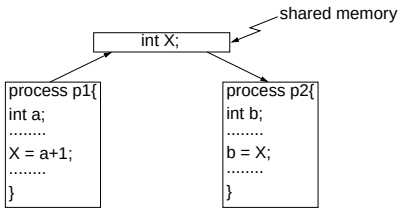
☞ Processes have to communicate in order to exchange information.

Various communication mechanisms are used in the 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.

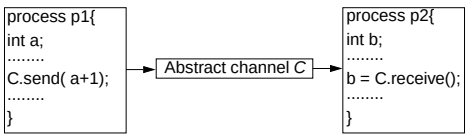


Private variables:
- *a*: local to p1
- *b*: local to p2

Shared variable:
- *X*

Message-passing Communication

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



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

Message-passing Communication (cont'd)

☞ 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 (cont'd)

☞ Non-blocking communication

Processes do not have to synchronize for communication.

But

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

Our example illustrating Control driven concurrency (slide 16):

```
module p1:
.........
end module
module p2:
.........
end module
module p3:
.........
end module
module p4:
.........
end module
run p1;
[run p2 || run p3];
run p4
```

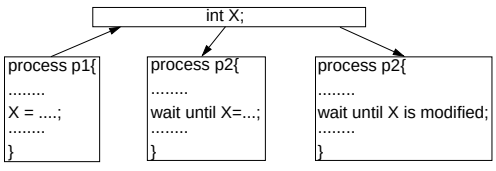
☞ With control-dependent synchronization the control structure is responsible for synchronization

☞ 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

☞ Communication mechanisms implicitly imply synchronization.

- Shared memory based synchronization



Data-dependent Synchronization (cont'd)

- Synchronization by message passing
  - Blocking communication with messages, automatically implies the synchronization between sender and receiver (slide 20).

And don't forget it: *Time*!

☞ How is time handled?

This makes a great difference between computation models!

Common Models of Computation

In the following lectures 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
- Synchronous/Reactive Languages
- Codesign Finite State Machines
- Timed Automata

Summary

- Design can be viewed as a sequence of refinement steps leading from specification to implementation.
- Specifications are formulated using specification languages.
- We would like the specification language to have a well defined semantics.
- The semantics of the system specification language is based on an underlying model of computation.

Summary (cont'd)

- The key aspects of "how difficult it is to write the model?" and "what we can do with the specified model?", are depending on the particular model of computation.
  The basic trade-off is expressiveness vs. the power of formal reasoning and efficient (possibly automatic) synthesis.
- The main aspects of a computation model we are interested in are: concurrency, communication&synchronisation, time, and hierarchy.
- In the following, we will study seven of the most representative models of computation for the embedded systems area.