Imperative model of computation

Jian-Jia Chen
(Slides are based on Peter Marwedel)
Informatik 12
TU Dortmund
Germany

These slides use Microsoft clip arts. Microsoft copyright restrictions apply.
# Models of computation considered in this course

<table>
<thead>
<tr>
<th>Communication/local computations</th>
<th>Shared memory</th>
<th>Message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td>Plain text, use cases</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Communicating finite state machines</td>
<td>StateCharts</td>
<td>SDL</td>
</tr>
<tr>
<td>Data flow</td>
<td></td>
<td>Kahn networks, SDF</td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td>C/E nets, P/T nets, …</td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>VHDL*, Verilog*, SystemC*, …</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
</tr>
<tr>
<td>Imperative (Von Neumann) model</td>
<td>C, C++, Java</td>
<td>C, C++, Java with libraries, CSP, ADA</td>
</tr>
</tbody>
</table>

* Classification based on semantic model
Imperative (von-Neumann) model

The von-Neumann model reflects the principles of operation of standard computers:

- Sequential execution of instructions (total order of instructions)
- Possible branches
- Visibility of memory locations and addresses

Example languages

- Machine languages (binary)
- Assembly languages (mnemonics)
- Imperative languages providing limited abstraction of machine languages (C, C++, Java, ....)
Threads/processes

- Initially available only as entities managed by OS
- In most cases:
  - Context switching between threads/processes, frequently based on pre-emption
- Made available to programmer as well
  - Partitioning of applications into threads (same address space)
- Languages initially not designed for communication, but synchronization and communication is needed!
Problems with imperative languages and shared memory

- Access to shared memory leads to anomalies, that have to be pruned away by mutexes, semaphores, monitors
- Potential deadlocks
- Access to shared, protected resources leads to priority inversion (chapter 4)
- Termination in general undecidable
- Timing cannot be specified and not guaranteed
Synchronous message passing: CSP

- CSP (communicating sequential processes) [Hoare, 1985],
  Rendez-vous-based communication:
Example:

process A
  ..
  var a ...
  a:=3;
  c!a; -- output
end

process B
  ..
  var b ...
  ...
  c?b; -- input
end

Determinate!
Synchronous message passing: Ada

After Ada Lovelace (said to be the 1st female programmer). US Department of Defense (DoD) wanted to avoid multitude of programming languages

Definition of requirements

Selection of a language from a set of competing designs (selected design based on PASCAL)

Ada’ 95 is object-oriented extension of original Ada.
Synchronous message passing: Ada-rendez-vous

```ada
task screen_out is
  entry call_ch(val:character; x, y: integer);
  entry call_int(z, x, y: integer);
end screen_out;
task body screen_out is
...
  select
    accept call_ch ... do ..
  end call_ch;
  or
    accept call_int ... do ..
  end call_int;
end select;

Sending a message:
begin
  screen_out.call_ch('Z',10,20);
  exception
    when tasking_error =>
      (exception handling)
end;
```
Java

Potential benefits:

- Clean and safe language
- Supports multi-threading (no OS required?)
- Platform independence (relevant for telecommunications)

Problems:

- Size of Java run-time libraries? Memory requirements.
- Access to special hardware features
- Garbage collection time
- Non-deterministic dispatcher
- Performance problems
- Checking of real-time constraints
Overview over Java 2 Editions

“J2ME … addresses the large, rapidly growing consumer space, which covers a range of devices from tiny commodities, such as pagers, all the way up to the TV set-top box.”

Lee’s conclusion

Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans.

Succinct Problem Statement

Threads are wildly nondeterministic.

The programmer’s job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes).

Improve threads?

Or replace them?

[Edward Lee (UC Berkeley), Artemis Conference, Graz, 2007]
Lifting Level of Abstraction

Model-Based Design
(e.g., Simulink, UML)
Automatic program synthesis: No more programming

High-level languages:
Programming to the application

Code generation from specifications:
still mostly a dream
It is not yet feasible to abstract algorithms.

Compilation:
perhaps “the” success story of computer science
It is feasible to abstract the platform.

The “assembly age”:
Programming to the platform
Comparison of models

Jian-Jia Chen
(slides are based on
Peter Marwedel)
Informatik 12,
TU Dortmund,
Germany

2017年 11 月 08 日

These slides use Microsoft clip arts. Microsoft copyright restrictions apply.
Models of computation considered in this course

<table>
<thead>
<tr>
<th>Communication/local computations</th>
<th>Shared memory</th>
<th>Message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td>Plain text, use cases</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Communicating finite state machines</td>
<td>StateCharts</td>
<td>SDL</td>
</tr>
<tr>
<td>Data flow</td>
<td></td>
<td>Kahn networks, SDF</td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td>C/E nets, P/T nets, ...</td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>VHDL*, Verilog*, SystemC*, ...</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
</tr>
<tr>
<td>Imperative (Von Neumann) model</td>
<td>C, C++, Java</td>
<td>C, C++, Java with libraries CSP, ADA</td>
</tr>
</tbody>
</table>

* Classification based on semantic model
Classification by Stuijk

- **Expressiveness** and **succinctness** indicate, which systems can be modeled and how compact they are.

- **Analyzeability** relates to the availability of scheduling algorithms and the need for run-time support.

- **Implementation efficiency** is influenced by the required scheduling policy and the code size.
The expressiveness/analyzability conflict

[S. Stuijk, 2007]
Properties of processes/threads (1)

- **Number of processes/threads**
  - static;
  - dynamic (dynamically changed hardware architecture?)

- **Nesting:**
  - Nested declaration of processes
    ```plaintext
    process {
        process {
            process {
            }}
    }
    ```
  - or all declared at the same level
    ```plaintext
    process { ... }
    process { ... }
    process { ... }
    ```
Properties of processes/threads (2)

- Different techniques for process creation
  - Elaboration in the source (c.f. ADA)
    \[
    \text{declare} \\
    \quad \text{process P1 ...}
    \]
  - explicit fork and join (c.f. Unix)
    \[
    \text{id} = \text{fork}();
    \]
  - process creation calls
    \[
    \text{id} = \text{create\_process(P1)};
    \]

E.g.: StateCharts comprises a static number of processes, nested declaration of processes, and process creation through elaboration in the source.
How to cope with MoC and language problems in practice?

Mixed approaches:

(RT–) UML or equivalent → SDL → C–programs → Assembly programs → Objectcode

(RT–) UML or equivalent → VHDL → Net list → hardware → Objectcode

(RT–) UML or equivalent → (RT–) Java
Transformations between models

- Transformations between models are possible, e.g.
  - Frequent transformation into sequential code
  - Transformations between restricted Petri nets and SDF
  - Transformations between VHDL and C
- Transformations should be based on the precise description of the semantics (e.g. Chen, Sztipanovits et al., DATE, 2007)
Mixing models of computation: Ptolemy

Ptolemy (UC Berkeley) is an environment for simulating multiple models of computation.

http://ptolemy.berkeley.edu/
(http://ptolemy.berkeley.edu/ptolemyll/ptll8.0/ptll8.0.1/doc/index.htm)

Available examples are restricted to a subset of the supported models of computation.
## Mixing MoCs: Ptolemy
(Focus on executable models; “mature” models only)

<table>
<thead>
<tr>
<th>Communication/local computations</th>
<th>Shared memory</th>
<th>Message passing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Undefined components</strong></td>
<td></td>
<td>Synchronous</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>Communicating finite state machines</td>
<td>FSM, synchronous/reactive MoC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data flow</td>
<td>Kahn networks, SDF, dynamic dataflow, discrete time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>DE</td>
<td>Experimental distributed DE</td>
<td></td>
</tr>
<tr>
<td>Von Neumann model</td>
<td>CSP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wireless</td>
<td>Special model for wireless communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Continuous time</td>
<td>Partial differential equations</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mixing models of computation: UML
(Focus on support of early design phases)

<table>
<thead>
<tr>
<th>Communication/local computations</th>
<th>Shared memory</th>
<th>Message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined components</td>
<td><em>use cases</em></td>
<td>Synchronous</td>
</tr>
<tr>
<td>Communicating finite state machines</td>
<td>State diagrams</td>
<td>sequence charts, timing diagrams</td>
</tr>
<tr>
<td>Data flow</td>
<td>Data flow</td>
<td></td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td>activity charts</td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Von Neumann model</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
UML for embedded systems?

Initially not designed for real-time.

Initially lacking features:
- Partitioning of software into tasks and processes
- specifying timing
- specification of hardware components

Projects on defining profiles for embedded/real-time systems
- Schedulability, Performance and Timing Analysis
- SysML (System Modeling Language)
- UML Profile for SoC
- Modeling and Analysis of Real-Time Embedded Systems
- UML/SystemC, …

Profiles may be incompatible
Modeling levels

Levels, at which modeling can be done:

- System level
- Algorithmic level: just the algorithm
- Processor/memory/switch (PMS) level
- Instruction set architecture (ISA) level: function only
- Transaction level modeling (TML): memory reads & writes are just “transactions“ (not cycle accurate)
- Register-transfer level: registers, muxes, adders, .. (cycle accurate, bit accurate)
- Gate-level: gates
- Layout level

Tradeoff between accuracy and simulation speed
What ‘s the bottom line?

- The prevailing technique for writing embedded SW has inherent problems; some of the difficulties of writing embedded SW are not resulting from design constraints, but from the modeling.
- However, there is no ideal modeling technique.
- The choice of the technique depends on the application.
- Check code generation from non-imperative models.
- There is a tradeoff between the power of a modeling technique and its analyzability.
- It may be necessary to combine modeling techniques.
- **In any case, open your eyes & think about the model before you write down your spec!** Be aware of pitfalls.
- You may be forced, to use imperative models, but you can still implement, for example, finite state machines or KPNs in Java.