Lecture 2 – Models of Computation, Languages

Andreas Gerstlauer
Electrical and Computer Engineering
University of Texas at Austin
gerstl@ece.utexas.edu
Lecture 2: Outline

- **Overview**
  - Modeling, computational models, languages

- **Models of Computation (MoCs)**
  - Process models
  - State machine models

- **System-Level Design Languages (SLDLs)**
  - Goals, requirements
  - Communication and computation
Modeling

• **Design models as an abstraction of a design instance**
  • Representation (model) of some aspect of reality
    – Virtual prototyping of what has been decided
  • Specification for further implementation/synthesis
    – Describe desired functionality of what still to build
  ➢ Usually a combination of both
    ➢ Different parts of the model or different use cases for the same model

• **Needed in every step of the design process**
  ➢ Documentation and specification of outputs and inputs
    ➢ Capability to capture complex systems
    ➢ Precise, complete and unambiguous
  ➢ Observability and predictability to reason about properties
    ➢ Validation through simulation
    ➢ Analysis through formal methods
Models of Computation (MoCs)

• **Define a class of models**
  • Formal, abstract description of a system
    – Decomposition into pieces and their relationship
    – Well-defined objects and composition rules
  • Various degrees of
    – supported features
    – complexity
    – expressive power

• **Examples**
  • Sequential, imperative program models
    – Sequence of statements that manipulate program state
  • Process models: networks, dataflow, calculi
    – Concurrency, data dependencies/causality
  • State machine models: evolution from FSM to PSM
    – Control flow and state enumeration
Models vs. Languages

- Computation models describe system behavior
  - Conceptual notion, e.g., recipe, sequential program

- Languages capture models
  - Concrete form, e.g., English, C

- Variety of languages can capture one model
  - E.g., sequential program model -> C, C++, Java

- One language can capture variety of models
  - E.g., C++ -> sequential program model, object-oriented model, state machine model

- Certain languages better at capturing certain models

Text vs. Graphics

- Models versus languages not to be confused with text versus graphics
  - Text and graphics are just two types of languages
    - Text: letters, numbers
    - Graphics: circles, arrows (plus some letters, numbers)

Lecture 2: Outline

✓ Overview

• Models of Computation (MoCs)
  • Process models
    – Process networks
    – Dataflow
    – Process calculi
  • State machine models: evolution from FSM to PSM
    – Finite State Machine (FSM)
    – FSM with Data (FSMD)
    – Super-state FSMD
    – ...
    – Program State Machine (PSM)

• System-Level Design Languages (SLDLs)
Process Models

- **Set of processes**
  - Processes execute in parallel
    - Concurrent composition
  - Each process is internally sequential
    - Imperative program

- **Inter-process communication?**
  - Shared variables
    - Synchronization: critical section/mutex, monitor, …
  - Message passing
    - Synchronous, rendezvous (blocking send)
    - Asynchronous, queues (non-blocking send)

- Examples: MPI, Java
- Implementation: OS processes or threads
  - Single or multiple processors/cores
Deadlocks

- Circular chain of 2 or more processes which each hold a shared resource that the next one is waiting for
  - Circular dependency through shared resources

```
m1.lock();
m2.lock();
...
m2.unlock();
m1.unlock();
```

```
m2.lock();
m1.lock();
...
m1.unlock();
m2.unlock();
```

- Prevent chain by using the same precedence
- Use timeouts (and retry), but: livelock

- Dependency can be created when resources are shared
  - Side effects, e.g. when blocking on filled queues/buffers
Non-determinism

- **Deterministic**: same inputs always produce same outputs
- **Random**: probability of certain behavior
- **Non-deterministic**: undefined behavior (for some inputs)
  - Undefined execution order
    - Statement evaluation in imperative languages: $f(a++, a++)$
    - Concurrent process race conditions:
      \[
      \begin{align*}
      x &= a; \\
      y &= b; \\
      a &= 1; \\
      b &= 2;
      \end{align*}
      \]
      \[x = ?, y = ?\]

- **Can be desired or undesired**
  - **How to ensure correctness?**
    - Simulator must typically pick one behavior
  - **But: over-specification?**
    - Leave freedom of implementation choice
Kahn Process Network (KPN) [Kahn74]

- C-like process description (Algol)
- Unbounded (infinite), uni-directional, point-to-point FIFO queues (channels) for communication
  - Sender \(\text{send}()\) never blocks
  - Receiver \(\text{wait}()\) blocks until data available

**Deterministic**
- Process can’t peek into channels and can only wait on one channel at a time
- Output data produced by a process does not depend on the order of its inputs
  - Terminates on global deadlock: all process blocked on \(\text{wait}()\)

**Formal mathematical representation**
- Process = continuous function mapping input to output streams
Kahn Process Networks (KPN) (2)

• **Turing-complete, undecidable (in finite time)**
  • Terminates?
  • Can run in bounded buffers (memory)?

• **Scheduling [Parks95]**
  • Start with smallest bounded buffers
  • Schedule with blocking \texttt{send()} until (artificial) deadlock
  • Increase size of smallest blocked buffer and continue

- **Behavior does not depend on scheduling strategy**
  - Focus on causality, not order (implementation issue)

- **Difficult to schedule (Park’s algorithm)**
  - Dynamic memory allocation
  - Memory usage depends on scheduling strategy
Dataflow Networks

• Breaking processes down into network of *actors*
  • Atomic blocks of computation, executed when *firing*
  • Fire when required number of input *tokens* are available
    – Consume required number of tokens on input(s)
    – Produce number of tokens on output(s)

  ➢ Separate computation & communication/synchronization
    ➢ Actors (indivisible units of computation) may fire simultaneously, any order
    ➢ Tokens (units of communication) can carry arbitrary pieces of data

• Unbounded FIFOs on arcs between actors

  ➢ Signal-processing applications
Synchronous Dataflow (SDF) [Lee86]

• **Fixed number of tokens per firing**
  • Consume fixed number of inputs
  • Produce fixed number of outputs

  ![Synchronous Dataflow Diagram]

  - Can be scheduled statically
    - Solve system of linear equations for relative rates
    - Periodically schedule actors in proportion to their rates

  - Find a sequence of firings in each period
    - Trade-off code size and buffer sizes
      - Single-appearance vs. memory-minimal schedule

  **Initialization**
  - Delay
  - Prevent deadlock
Process Calculi

• **Rendezvous-style, synchronous communication**
  • Communicating Sequential Processes (CSP) [Hoare78]
  • Calculus of Communicating Systems (CCS) [Milner80]

• **Formal, mathematical framework: process algebra**
  • Algebra = \langle \text{objects, operations, axioms} \rangle
    – Objects: processes \((P, Q)\), events \((a, b)\)
    – Composition operators: parallel \((P \parallel Q)\), prefix/input \((a \rightarrow P)\), choice \((P\mid Q)\)
    – Axioms: indemnity \((\emptyset \parallel P = P)\), reduction \((a\rightarrow P \parallel a\rightarrow Q = P \parallel Q)\)
  • Manipulate processes by manipulating expressions

- Examples: LOTOS, Occam (Transputer), HandleC (Celoxica)
Lecture 2: Outline

✓ Overview

• Models of Computation (MoCs)
  ✓ Process models
    ✓ Process networks
    ✓ Dataflow
    ✓ Process calculi
  • State machine models: evolution from FSM to PSM
    – Finite State Machine (FSM)
    – FSM with Data (FSMD)
    – Super-state FSMD
    – ...
    – Program State Machine (PSM)

• System-Level Design Languages (SLDLs)
State Machine Models

- **Finite State Machine (FSM)**
  - Basic model for describing control
  - States and state transitions
    - $FSM = <S, I, O, f, h>$
  - Two types:
    - Mealy-type FSM (input-based)
    - Moore-type FSM (state-based)

![FSM model](image_url)

Source: R. Doemer, UC Irvine
State Machine Models

- **Finite State Machine (FSM)**
- **Data Flow Graph (DFG)**
  - Basic model for describing computation (variant of SDF)
  - Directed graph
    - Nodes: operations
    - Arcs: dependency of operations

Source: R. Doemer, UC Irvine
State Machine Models

- Finite State Machine (FSM)
- Data Flow Graph (DFG)
- Finite State Machine with Data (FSMD)
  - Combined model for control and computation
    - FSMD = FSM + DFG
  - Implementation: controller plus datapath

Source: R. Doemer, UC Irvine
State Machine Models

- Finite State Machine (FSM)
- Data Flow Graph (DFG)
- Finite State Machine with Data (FSMD)
- Super-State FSM with Data (SFSMD)

FSMD with complex, multi-cycle states
- States described by procedures in a programming language
State Machine Models

- Finite State Machine (FSM)
- Data Flow Graph (DFG)
- Finite State Machine with Data (FSMD)
- Super-State FSM with Data (SFSMD)
- Hierarchical Concurrent FSM (HCFSM)
  - FSM extended with hierarchy and concurrency
    - Multiple FSMs composed hierarchically (OR) and in parallel (AND)

- Example: StateCharts (graphical), Esterel/Lustre (textual)
  - Synchronous (lock-step) composition of concurrent FSMs

Source: R. Doemer, UC Irvine
State Machine Models

- Finite State Machine (FSM)
- Data Flow Graph (DFG)
- Finite State Machine with Data (FSMD)
- Super-State FSM with Data (SFSMD)
- Hierarchical Concurrent FSM (HCFSM)
- Program State Machine (PSM)

- HCFSMD plus programming language
  - States described by procedures in a programming language

Example: SpecC!

```c
... 
a = 42;
while (a<100)
{ 
b = b + a;
  if (b > 50)
    c = c + d;
  else
    c = c + e;
  a = c;
}
...
```

Source: R. Doemer, UC Irvine
Lecture 2: Outline

✓ Overview

✓ Models of Computation (MoCs)
  ✓ Process models
  ✓ State machine models

• System-Level Design Languages (SLDLs)
  • Goals, requirements
  • Communication and computation
Design Languages

- Represent a model in machine-readable form
  - Apply algorithms and tools

- Syntax provides the handle
  - Grammar defines possible strings over an alphabet

- Semantics connects/maps strings to a model
  - Operational semantics
    - Defined through execution of an abstract state machine
  - Denotational semantics
    - Define through mapping into a mathematical domain (e.g. functions)

- General underlying model of computation
  - Discrete-event semantics as common denominator
    - Event = (value, tag) tuples
    - Signal = sequence of events
    - Ordering of tags (time) defines order of events
  - Able to represent different MoCs
  - Defines interpretation of language into other MoCs
Simulation vs. Synthesis

- 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
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**

Source: R. Doemer, UC Irvine
### System-Level Design Languages (SLDLs)

- **Requirements**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>C++</th>
<th>Java</th>
<th>VHDL</th>
<th>Verilog</th>
<th>HardwareC</th>
<th>Statecharts</th>
<th>SpecCharts</th>
<th>SpecC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavioral</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hierarchy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structural</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hierarchy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concurrency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>handling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transitions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Composite</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data types</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **not supported**
- **partially supported**
- **supported**

Source: R. Doemer, UC Irvine
System-Level Design Languages (SLDLs)

- **Examples in use today**
  - C/C++
    - ANSI standard programming languages, software design
    - traditionally used for system design because of practicality, availability
  - SystemC
    - C++ API and library
    - initially developed at UCI, supported by Open SystemC Initiative
  - SpecC
    - C extension
    - developed at UCI, supported by SpecC Technology Open Consortium
  - SystemVerilog
    - Verilog with C extensions
  - Matlab
    - specification and simulation in engineering, algorithm design
  - UML
    - unified modeling language, software specification, graphical
  - SDL
    - telecommunication area, standard by ITU, used in COSMOS
  - SLDL
    - formal specification of requirements, not executable
  - etc.

*Source: R. Doemer, UC Irvine*
Separation of Concerns

- Fundamental principle in modeling of systems
- Clear separation of concerns
  - address separate issues independently
- System-Level Description Language (SLDL)
  - orthogonal concepts
  - orthogonal constructs
- System-level Modeling
  - Computation
    - encapsulated in modules / behaviors
  - Communication
    - encapsulated in channels

Source: R. Doemer, UC Irvine
Computation vs. Communication

• **Traditional model**

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

Source: R. Doemer, UC Irvine
Computation vs. Communication

• **Traditional model**
  - Processes and signals
  - Mixture of computation and communication
  - Automatic replacement impossible

• **SpecC model**
  - Behaviors and channels
  - Separation of computation and communication
  - Plug-and-play

Source: R. Doemer, UC Irvine
Computation vs. Communication

• Protocol Inlining
  • Specification model
  • Exploration model
    – Computation in behaviors
    – Communication in channels

Source: R. Doemer, UC Irvine
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

Source: R. Doemer, UC Irvine
Intellectual Property (IP)

- Computation IP: Wrapper model

Source: R. Doemer, UC Irvine
Intellectual Property (IP)

- **Computation IP: Wrapper model**

Source: R. Doemer, UC Irvine
Intellectual Property (IP)

- **Computation IP: Wrapper model**
  
  ![Diagram of Computation IP: Wrapper model]

  - Synthesizable behavior
  - Transducer
  - IP in wrapper
  - Replacable at any time

- **Protocol inlining with wrapper**
  
  ![Diagram of Protocol inlining with wrapper]

- Source: R. Doemer, UC Irvine
Intellectual Property (IP)

- Computation IP: Adapter model

Source: R. Doemer, UC Irvine
Intellectual Property (IP)

- **Computation IP: Adapter model**

  - Synthesizable behavior
  - Transducer
  - Adapter
  - IP

  ![Diagram of Computation IP: Adapter model](image)

  - Replacable at any time

- **Protocol inlining with adapter**

  - Before
  - After

  ![Diagram of Protocol inlining with adapter](image)

  - Source: R. Doemer, UC Irvine
• Communication IP: Channel with wrapper

Virtual channel

IP protocol channel in wrapper

Source: R. Doemer, UC Irvine
Intellectual Property (IP)

- Communication IP: Channel with wrapper

Virtual channel

IP protocol channel in wrapper

- Protocol inlining with hierarchical channel

Source: R. Doemer, UC Irvine
• **Incompatible busses: Transducer insertion**

---

Source: R. Doemer, UC Irvine
Intellectual Property (IP)

- Incompatible busses: Transducer insertion

- Protocol inlining with transducer

Source: R. Doemer, UC Irvine