Behavioral Models and Specification Languages

**Behavioral Models**
- Finite State Machine (FSM)
- NDFSM
- composed FSM
- Petri Net (PN)
- Data Flow Graph (DFG)
- Control Flow Graph (CFG)
- Control/Data Flow Graph (CDFG)

**Basic Concepts**
- concurrency
- hierarchy
- communication
- synchronisation
- exception handling
- non-determinism
- timing

**Specification Languages**
- StateCharts
- SDL
- VHDL
- SystemC
- ...
Why do we need Behavioral Models?

Models abstract from the real world to some extent (describing part of the system from a specific view)

- Models may **serve very different purposes**, e.g. to describe, document or reason about
  - the behavior of a system
  - the performance of a system
  - or some other purpose (safety, reliability, shock, temperature, ...)

**Behavioral Models** (or Models of Computation) represent **formal frameworks** to describe the behavior of systems or parts hereof

- Formal definitions **ease reasoning and refinement**

- The description of the behavior may be **abstract or concrete**, e.g.
  - an abstract state machine describing the behavior of an elevator control
  - a C program, assembler program or VHDL description

- Behavioral models may concentrate on
  - the externally-visibly behavior (analysis phase) or/and
  - the internal details of the system (design and implementation phase)
Behavioral Models and Specification Languages

Behavior models (e.g. FSM, PN, DFG) represent well defined basic mechanisms with underlying syntax and semantics to model specific aspects of a system or parts hereof.

Specification languages (e.g. Statecharts, SDL, VHDL) employ a set of basic modeling mechanisms useful to specify systems of a specific application domain.

Behavioral models and specification languages allow to:
- capture unambiguously the required functionality
- verify correctness of the functional specification wrt. given properties
- synthesize part of the specification
- use a variety of tools

Important characteristics of behavior models and specification languages:
- Expressiveness
- Generality/applicability
- Simplicity
- Compilability/synthesizability
- Verifiability
Behavioral Models (discrete time and state)

**State-oriented** models (behavior)
- Finite State Machines
- Petri Nets
- Hierarchical concurrent FSM

**Action-oriented** models (behavior)
- data flow graph
- control flow graph

**Structural Models**
- Structure-oriented models
  - component connectivity graph
- Data-oriented models
  - entity relationship diagram
  - Jackson’s diagram

**Heterogeneous** models
- control/data flow graph
- structure chart
- programming language paradigm
- object-oriented model
- program-state machine
- queueing model
More Models and Languages ... to complete the confusion!

More models ...
- continuous time
- continuous state
- structure-oriented
- data-oriented
- ...

Each of these provides a formal framework for reasoning about certain aspects of the system

Focus here
- behavior rather that structure
- discrete time rather than continuous
- discrete state (digital) rather than continuous (analog)
Behavioral Models – Application Needs

**Telecommunication**
- Heterogeneous specifications including
  - data processing
  - control functions
- **Data processing**, e.g. encryption, forward error correction, …
  - computations done at regular (often short) intervals
  - efficiently specified and synthesized using Data Flow models
- **Control functions** (data-dependent and real-time)
  - say when and how data computation is done
  - efficiently specified and synthesized using FSM models

Need a common model to perform global system analysis and optimization

**Reactive Real-Time Systems**
- “react” to external environment
- maintain permanent interaction
- ideally never terminate
- timing constraints (real-time)
- As opposed to
  - transformational systems
  - interactive systems
Finite State Machines (FSM)

Functional decomposition into states of operation

- finite states
- transitions between states
- event triggered transitions
- neither concurrency nor time (sequential FSMs)

Typical applications:

- reactive (control) systems
- protocols (telecom, computers, ...)

![Finite State Machine Diagram]

- \( \text{KEY\_ON} \Rightarrow \text{START\_TIMER} \)
- \( \text{KEY\_OFF} \) or \( \text{BELT\_ON} \) => \( \text{OFF} \)
- \( \text{END\_TIMER\_10} \) or \( \text{BELT\_ON} \) or \( \text{KEY\_OFF} \) => \( \text{ALARM\_OFF} \)
- \( \text{END\_TIMER\_5} \) => \( \text{ALARM\_ON} \)
FSM Example: Seat Belt Control

Informal specification:
If the driver
  turns on the key and
  does not fasten the seat belt within 5 seconds
then an alarm beeps
  for 5 seconds, or
  until the driver fastens the seat belt, or
  until the driver turns off the key

Formal specification:

If none of the conditions is satisfied, implicit self-loop in the current state
Non-deterministic FSM (NDFSM): Incomplete Specification

Example: error checking

- Partially specified (incomplete):

Focus on synchronisation after 8th bit, not on parity

- Completely specified as *even parity*.
NDFSM: Unknown Behavior

Modeling the *environment*

- Nondeterminism is useful to:
  - optimize (don’t care conditions)
  - verify (exclude impossible cases)

Example: driver model

- Can be refined
  - e.g. introducing timing constraints
  - (minimum reaction time of 0.1 s between actions)
NDFSM: Time Range

Special case of unspecified/unknown behavior, but so common to deserve special treatment for efficiency

Example: nondeterministic delay (between 6 and 10 s)
NDFSMs and FSMs

- Formally FSMs and NDFSMs are equivalent
  (Rabin-Scott construction, Rabin ‘59)
- In practice, NDFSMs are often more compact
  (exponential blowup for determinization)

Example: non-deterministic selection
of transition a in state s1
Finite State Machines – Discussion

Advantages:

- Easy to use (graphical languages)
- Powerful algorithms for
  - synthesis (SW and HW)
  - verification

Disadvantages:

- Sometimes over-specify implementation
  - (sequencing is fully specified)
- Number of states can be unmanageable
- Numerical computations cannot be specified compactly (need Extended FSMs)
Modeling Concurrency

Systems are typically composed of chunks of rather independent functionalities,
  e.g. seat belt control || timer || driver
Systems may be physically distributed,
  e.g. peer protocol automata
Need to compose parts described by sequential FSMs
  • construct a complete model of the system
  • building the cartesian product results in state explosion
Approach
  • Describe the system using a number of separate FSMs and interconnect them
Issue
  • How do the interconnected FSMs talk to each other?

Fundamental hypothesis:
  all the FSMs change state together (synchronicity)
System state = Cartesian product of component states
  (state explosion may be a problem...)

FSM Composition – Example

Example: seat belt control || timer

START_TIMER => START_TIMER =>

KEY_ON => START_TIMER

KEY_OFF or BELT_ON =>

END_TIMER_5 =>

ALARM_ON

END_TIMER_10 or BELT_ON or KEY_OFF => ALARM_OFF

Belt Control

Timer

OFF

WAIT

ALARM

SEC => SEC => SEC => SEC => SEC => SEC => SEC =>

KEY_OFF =>

ALARM_ON

KEY_OFF or BELT_ON =>

END_TIMER_5 =>

ALARM_OFF

START_TIMER =>

SEC =>

3

4

5

6

7

8

9

0

1

2

START_TIMER =>

END_TIMER_10

SEC =>

END_TIMER_10

SEC =>

END_TIMER_5

SEC =>

END_TIMER_5

SEC =>
FSM Composition – Example

Cartesian product

OFF, 0

WAIT, 1

KEY_ON and START_TIMER =>
START_TIMER must be coherent

SEC and
not (KEY_OFF or BELT_ON) =>

not SEC and
(KEY_OFF or BELT_ON) =>

WAIT, 2

OFF, 1

SEC and
(KEY_OFF or BELT_ON) =>

OFF, 2

not SEC and
(KEY_OFF or BELT_ON) =>
Moore vs. Mealy Automata

Theoretically, same computational power
In practice, different characteristics

- **Moore machines:**
  - non-reactive
    - (response delayed by 1 cycle –
      clocked change of output only)
  - easy to design
    - (always well-defined)
  - good for SW implementation
    - software is always “slow”

- **Mealy machines:**
  - reactive (immediate response
    to changes of input)
  - hard to compose
  - problematic SW implementation
    - due to immediate response to changes of input (interrupts/polling)
    - software must be “fast enough”
    - may be needed in hardware, for speed
Hierarchical FSM models – StateCharts

Problem: how to reduce the size of the representation?

Harel’s classical papers on StateCharts (language) and bounded concurrency (model): 3 orthogonal exponential reductions

- **Hierarchy:**
  - state \( a \) “encloses” an FSM
  - being in \( a \) means FSM in \( a \) is active
  - states of \( a \) are called OR states
  - used to model preemption and exceptions

- **Concurrency:**
  - two or more FSMs are simultaneously active
  - states are called AND states

- **Non-determinism:**
  - used to abstract behavior
StateCharts – Basic Principles

Basic principles:
- An extension of conventional FSMs
- Conventional FSMs are inappropriate for the behavioral description of complex control
  - flat and unstructured
  - inherently sequential in nature
- StateCharts support
  - repeated decomposition of states into sub-states in an AND/OR fashion, combined with a
  - synchronous communication mechanism (instantaneous broadcast)

State decomposition:
- **OR-States** have sub-states that are related to each other by exclusive-or
- **AND-States** have orthogonal state components (synchronous FSM composition)
  - AND-decomposition can be carried out on any level of states (more convenient than allowing only one level of communicating FSMs)
- **Basic States** have no sub-states (bottom of hierarchy)
- **Root State** have no parent states (top of hierarchy)
State U is an abstraction of states S and T

To be in state U the system must be either in state S or in state T
To be in state U the system must be both in states S and T
StateCharts – Syntax

- The general syntax of an expression labeling a transition in a StateChart is $e[c]/a$ where
  - $e$ is the event that triggers the transition
  - $c$ is the condition that guards the transition
    (cannot be taken unless $c$ is true when $e$ occurs)
  - $a$ is the action that is carried out if and when the transition is taken

- For each transition label:
  - condition and action are optional
  - an event can be the changing of a value
  - standard comparisons (e.g. $x > y$) are allowed as conditions
  - assignment statements (e.g. $x := 10$) are allowed as actions
StateCharts – Actions and Events

- An action \( a \) on the edge leaving a state may also appear as an event triggering a transition going into an orthogonal state:
  - a state transition broadcasts an event visible immediately to all other FSMs, that can make transitions immediately and so on
  - executing the first transition will immediately cause the second transition to be taken simultaneously
- Actions and events may be associated to the execution of orthogonal components: \( \text{start}(A), \text{stopped}(B) \)

Statecharts will be covered in detail in IHS 2
Synchronous vs. Asynchronous FSMs

**Synchronous FSMs** (e.g. StateCharts):
- communication by shared variables that are read and written in zero time
- communication and computation happens instantaneously at discrete time instants
- all FSMs execute a transition simultaneously (lock-step)
- may be difficult to implement
  - multi-rate specifications
  - distributed/heterogeneous architectures

**Asynchronous FSMs** (e.g. SDL, CSP):
- free to proceed independently
- do not execute a transition at the same time (except for CSP rendezvous)
- may need to share notion of time: synchronization
- easy to implement

Multitude of commercial and non-commercial graphical languages and tools:
- StateCharts, UML, SDL, StateFlow
- tool support for design, simulation, validation, code generation, HW synthesis, …
Asynchronous Communication

Blocking vs. non-Blocking

- **blocking read** (receiver waits for sender)
  - reading process cannot test for emptiness of input
  - must wait for input to arrive before proceeding

- **blocking write** (sender waits for receiver)
  - writing process must wait for successful write before continue

Languages

- blocking write/blocking read (CSP, CCS)
- non-blocking write/blocking read (FIFO, CFSMs, SDL)
- non-blocking write/non-blocking read (shared variables)
Asynchronous Communication – Buffering

- Buffers used to adapt when sender and receiver have different rate
  - size of buffer?
- Lossless vs. lossy
  - events/tokens may be lost
  - bounded memory: overflow or overwriting
  - need to block the sender
- Single vs. multiple read
  - result of each write can be read at most once or several times
- Pure FIFO
  - prioritized events
  - out of order access to FIFO
Communication Mechanisms

- **Rendez-Vous (CSP)**
  - No space is allocated for shared data, processes need to synchronize in some specific points to exchange data
  - Read and write occur simultaneously
- **Shared memory**
  - Multiple non-destructive reads are possible
  - Writes delete previously stored data
- **Buffered (FIFO)**
  - Bounded (ECFSMs, CFSMs)
  - Unbounded (SDL, ACFSMs, Kahn Process Networks, Petri Nets)
### Communication Models

<table>
<thead>
<tr>
<th></th>
<th>Senders</th>
<th>Receivers</th>
<th>Buffer Size</th>
<th>Blocking Reads</th>
<th>Blocking Writes</th>
<th>Single Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsynchronized</td>
<td>many</td>
<td>many</td>
<td>one</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Read-Modify-write</td>
<td>many</td>
<td>many</td>
<td>one</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Unbounded FIFO</td>
<td>one/many</td>
<td>one</td>
<td>unbounded</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Bounded FIFO</td>
<td>one/many</td>
<td>one</td>
<td>bounded</td>
<td>yes</td>
<td>may be</td>
<td>yes</td>
</tr>
<tr>
<td>Rendezvous</td>
<td>one</td>
<td>one</td>
<td>one</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

- *writer is blocked (e.g. if buffer is full)*
- *reader is blocked (e.g. if buffer is empty)*
- *data may be read once only*
- *writer is blocked (e.g. if buffer is full)*
- *data may be read once only*
Petri Nets (PNs)

- Model introduced by C.A. Petri in 1962
  - Ph.D. Thesis: “Communication with Automata”
- Applications: distributed computing, manufacturing, control, communication networks, transportation, …
- PNs describe explicitly and graphically:
  - sequencing/causality
  - conflict/non-deterministic choice
  - concurrency
- Asynchronous model (partial ordering)
- Main drawback: no hierarchy
Petri Net

- A PN \((N,M_0)\) is a Petri Net Graph \(N\)
  - **Places**: represent distributed state by holding tokens
    - marking (state) \(M\) is an \(n\)-vector \((m_1,m_2,m_3\ldots)\), where \(m_i\) is the non-negative number of tokens in place \(p_i\).
    - initial marking \((M_0)\) is initial state
  - **Transitions**: represent actions/events
    - enabled transition: enough tokens in predecessors
    - firing transition: modifies marking
  - ... and an initial marking \(M_0\)
Concurrency, causality, choice

Concurrent events:
- t1
- t2

Causality, sequencing:
- t3
- t4

Choice, conflict:
- t5
- t6
Communication Protocol
Producer-Consumer Problem

Produce

Buffer

Consume
Petri Nets - Properties

- **Behavioral properties**: depend on the initial marking (most interesting)
  - Reachability (of marking $M$ from marking $M_0$)
  - Boundedness (number of tokens is limited)
  - Conservation (number of tokens remains constant)
  - Liveness (any transition can be fired from any marking $M$)
  - Schedulability

- **Structural properties**: do not depend on the initial marking (often too restrictive)
  - Consistency
  - Structural boundedness
Control vs. Data Flow Applications

Rough classification:
- **control**:
  - don’t know when data arrive (quick reaction)
  - time of arrival often matters more than value
- **data**:
  - data arrive in regular streams (samples)
  - values matter most

Distinction is important for:
- specification (language, model, ...)
- synthesis (scheduling, optimization, ...)
- validation (simulation, formal verification, ...)

Specification, synthesis and validation methods emphasize:
- for **control**:
  - event/reaction relation
  - response time (real-time scheduling for deadline satisfaction)
  - priority among events and processes
- for **data**:
  - functional dependency between input and output
  - memory/time efficiency (data-flow scheduling for efficient pipelining)
  - all events and processes are *equal*
Data Flow Graph (DFG)

Powerful formalism for data-dominated applications

DFG support the specification of transformational systems:
- output is a function of the input
- set of actors (nodes) connected by a set of arcs representing the data flow
- no states, no external events to trigger state changes
- unbounded FIFO queues (main data store)
- no control nodes, e.g. branch, loop

DFG represent a partial ordered model of the computation
- => specification of problem-inherent dependencies only
- => suitable for scheduling and code generation
- => there is a relation between buffer dimensioning and scheduling
  (static scheduling minimizes the number of buffers required)

Languages:
- graphical: Ptolemy (UCB), GRAPE (U. Leuven), SPW (Cadence), COSSAP (Synopsys)
- textual: Silage (UCB, Mentor), Haskell, Lucid
Semantics (informal)

- actors perform computation (often stateless)
- firing of actors when all needed inputs are available
- unbounded FIFOs for unidirectional exchange of data between actors (integer, floats, arrays, etc.)
- extensions to model decisions

Example: FIR (finite impuls response) filter

- single input sequence $i(n)$
- single output sequence $o(n)$
- $o(n) = c1 \cdot i(n) + c2 \cdot i(n-1)$
DFG – Example

\[ x = 3a + b*b - c; \]
\[ y = a + b*x; \]
\[ z = b - c*(a + b); \]
Control Flow Graph (CFG)

- also called flow chart (abstract description of program designs)
- focus on control aspect of a system
- set of nodes and arcs
- trigger of an activity (node) when a particular preceding activity is completed
- different triggers for transitions
- suitable for well defined tasks that do not depend on external events
- imposes a complete order on the execution of activities
  => close to implementation (on conventional computer architecture)
- various variants with various levels of details
  - simple operator level (addition, multiplication, etc)
  - abstract function/procedure level
what_is_this {
    read (a,b);
    done = FALSE;
    repeat {
        if (a>b)
            a = a-b;
        elseif (b>a)
            b = b-a;
        else done = TRUE;
    } until done;
    write (a);
}
Control/Data Flow Graph (CDFG)

- also called sequence graph
- mixture of control and data flow graph
- hierarchy of sequential elements
  - units model data flow
  - hierarchy models control flow
- special nodes (for control operations)
  - start/end node: NOP (no operation) – all inputs needed (AND), all outputs needed (AND)
  - branch node (BR) – one out of many outputs selected (OR)
  - iteration (LOOP) – one out of two outputs selected (OR)
  - procedure call (CALL) – lower hierarchy is executed exactly once
- attributes
  - nodes: execution time, cost, ...
  - arcs: conditions for branches and loops
w = a + b;
x = w * c;
y = b * b;
z = w - c;

Legend:

→ data dependencies

----- control dependencies

Notes: AND dependencies at NOPs (NO Operation), OR dependencies at BRanches and LOOPs
CDFG – Branch

c = a < b;
IF (c) THEN
  p = m + n;
  q = m * n;
ENDIF
x = a - b;

Notes:
• data dependencies are not fully specified
• x = a - b may execute in parallel to IF statement
• computation of p and q within IF statement may execute in parallel
d = 2*x;
WHILE (d<5) DO
  write(d);
  d = d + 1;
ENDWHILE
\[ d = x - y; \]
\[ e = d \times x; \]
\[ \text{sub}(x, y); \]
\[ \ldots \]

PROCEDURE sub \((m, n)\)
\[ p = m + n; \]
\[ q = m \times n; \]
\[ \text{END sub} \]
Review of Models, Concepts and Languages

Behavioral Models
- Finite State Machine (FSM)
- NDFSM
- composed FSM
- Petri Net (PN)
- Data Flow Graph (DFG)
- Control Flow Graph (CFG)
- Control/Data Flow Graph (CDFG)

Basic Concepts
- concurrency
- hierarchy
- communication
- synchronisation
- exception handling
- non-determinism
- timing

Specification Languages
- StateCharts
- SDL
- VHDL
- SystemC
- ...

Basic Concepts
- concurrency
- hierarchy
- communication
- synchronisation
- exception handling
- non-determinism
- timing

Specification Languages
- StateCharts
- SDL
- VHDL
- SystemC
- ...

Behavioral Models
- Finite State Machine (FSM)
- NDFSM
- composed FSM
- Petri Net (PN)
- Data Flow Graph (DFG)
- Control Flow Graph (CFG)
- Control/Data Flow Graph (CDFG)
Summary of Basic Concepts of Models and Languages

State transitions
- events triggering a state transition (simple input, complex conditions)
- computation associated with transition

Concurrency
- decomposition of behavior in concurrent entities
- different levels of concurrency (job, task-, statement-, operation-level)
- data-driven (data dependencies) vs control-driven concurrency (control dependencies)
- reduction of states

Hierarchy
- structural hierarchy (system, block, process, procedure)
- behavioral hierarchy (hierarchical transitions, fork-join)

Programming constructs
- specify sequential algorithm

Communication
- shared variables (broadcast)
- message passing
- synchronous vs. asynchronous

Synchronization
- control-dependent (fork-join)
- data-dependent (data, event, message)

Exception handling
- immediate termination of current behavior

Non-determinism
- choice between multiple transitions
- non-deterministic ordering

Timing
- timeouts
- time constraints (e.g. exec. time)
References

- University of Berkeley: Overview on papers on behavioral modelling
  http://www-cad.eecs.berkeley.edu/~polis/class/index.html