Outline

• Behavioral Design: Models of Computation
  • FSMs
  • Data Flow Models
  • Discrete Event Systems
  • Petri Nets
  • The Tagged Signal Model

Design

• From an idea…
• ... build something that performs a certain function
• Never done directly:
  • some aspects are not considered at the beginning of the development
  • the designer wants to explore different possible implementations in order to maximize (or minimize) a cost function
• Models can be used to reason about the properties of an object
**Formalization**

Model of a design with precise unambiguous semantics:

- Implicit or explicit relations: inputs, outputs and (possibly) state variables
- Properties
- “Cost” functions
- Constraints

\[
\text{Formalization of Design + Environment} = \text{closed system of equations and inequalities over some algebra.}
\]

---

**Models of Computation: And There are More...**

- Continuous time (ODEs)
- Spatial/temporal (PDEs)
- Discrete time
- Rendezvous
- Synchronous/Reactive
- Dataflow
- ...

Each of these provides a formal framework for reasoning about certain aspects of embedded systems.
Model Of Computation

- A MoC is a framework in which to express what sequence of actions must be taken to complete a computation
- Examples: Finite State Machine, Turing Machine, differential equation
- Why different models?
  - Different models = different properties
  - Turing-complete models are too powerful!
  - Some problems may be undecidable
- MOC needs to
  - be powerful enough for application domain
  - have appropriate synthesis and validation algorithms

Usefulness of a Model of Computation

- Expressiveness
- Generality
- Simplicity
- Compilability/ Synthesizability
- Verifiability

The Conclusion

One way to get all of these is to mix diverse, simple models of computation, while keeping compilation, synthesis, and verification separate for each MoC. To do that, we need to understand these MoCs relative to one another, and understand their interaction when combined in a single system design.
**Common Models of Computation**

- Finite State Machines
  - finite state
  - no concurrency nor time
- Data-Flow
  - Partial Order
  - Concurrent and Determinate
  - Stream of computation
- Discrete-Event
  - Global Order (embedded in time)
- Continuous Time

The behavior of a design in general is described by a composition

---

**Notion of Time**

- Continuous time
- Discrete time
- Totally-ordered discrete events
- Multirate discrete time
- Partially-ordered discrete events
- Synchronous/reactive

Salvador Dalí, *The Persistence of Memory*, 1931
Control versus Data Flow

- Fuzzy distinction, yet useful for:
  - specification (language, model, ...)
  - synthesis (scheduling, optimization, ...)
  - validation (simulation, formal verification, ...)

- 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)
    - value matters most

Control versus Data Flow

- 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
      (Dataflow scheduling for efficient pipelining)
    - all events and processes are equal
**Telecom/MM applications**

- Heterogeneous specifications including
  - data processing
  - control functions
- **Data processing**, e.g. encryption, error correction...
  - computations done at regular (often short) intervals
  - efficiently specified and synthesized using DataFlow 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**

- 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
Models Of Computation for reactive systems

• We need to consider essential aspects of reactive systems:
  • time/synchronization
  • concurrency
  • heterogeneity

• Classify models based on:
  • how specify behavior
  • how specify communication
  • implementability
  • composability
  • availability of tools for validation and synthesis

Main MOCs:
• Communicating Finite State Machines
• Dataflow Process Networks
• Petri Nets
• Discrete Event
• (Abstract) Codesign Finite State Machines

Main languages:
• StateCharts
• Esterel
• Dataflow networks
**Finite State Machines**

- Functional decomposition into states of operation
- Typical domains of application:
  - control functions
  - protocols (telecom, computers, ...)
- Different communication mechanisms:
  - synchronous
    (classical FSMs, Moore ‘64, Kurshan ‘90)
  - asynchronous
    (CCS, Milner ‘80; CSP, Hoare ‘85)

---

**FSM Example**

- 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
**FSM Example**

![FSM Diagram]

If no condition is satisfied, implicit self-loop in the current state

**FSM Definition**

- **FSM** = (I, O, S, r, δ, λ)
- I = {KEY_ON, KEY_OFF, BELT_ON, END_TIMER_5, END_TIMER_10}
- O = {START_TIMER, ALARM_ON, ALARM_OFF}
- S = {OFF, WAIT, ALARM}
- r = OFF
- Set of all subsets of I (implicit “and”)
  All other inputs are implicitly absent

- **δ** : 2 × S → S
  - e.g. δ({KEY_OFF}, WAIT) = OFF

- **λ** : 2^1 × S → 2^0
  - e.g. λ({KEY_ON}, OFF) = {START_TIMER}
Non-deterministic FSMs

- \( \delta \) and \( \lambda \) may be relations instead of functions:
  - \( \delta \subseteq 2^S \times S \times S \)
  - \( \lambda \subseteq 2^S \times S \times 2^S \)
  - e.g. \( \delta((\text{KEY\_OFF}, \text{END\_TIMER\_5}), \text{WAIT}) = \{(\text{OFF}), (\text{ALARM})\} \)
  - \( \lambda \subseteq 2^S \times S \times 2^S \)

- Non-determinism can be used to describe:
  - an unspecified behavior
    - (incomplete specification)
  - an unknown behavior
    - (environment modeling)

NDFSM: incomplete specification

- E.g. error checking first partially specified:

- Then completed as even parity:

- Could be implemented as CRC later
**NDFSM: unknown behavior**

- Modeling the *environment*
- Useful to:
  - optimize (don’t care conditions)
  - verify (exclude impossible cases)
- E.g. driver model:

```
  s0
  => KEY_ON or KEY_OFF or BELT_ON
```

- Can be refined
  E.g. introduce timing constraints
  (minimum reaction time 0.1 s)

**NDFSM: time range**

- Special case of unspecified/unknown behavior, but so common to deserve special treatment for efficiency
- E.g. undetermined 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)

**Finite State Machines**

- **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**

- Need to compose parts described by FSMs
- Describe the system using a number of FSMs and interconnect them
- How do the interconnected FSMs talk to each other?

**FSM Composition**

- Bridle complexity via hierarchy: *FSM product yields an FSM*
- Fundamental hypothesis:
  - all the FSMs change state together (*synchronicity*)
- System state = Cartesian product of component states
  (state explosion may be a problem...)
- E.g. seat belt control + timer
**FSM Composition**

![FSM Diagram]

**FSM Composition**

• Given
  • \( M_1 = (I_1, O_1, S_1, r_1, \delta_1, \lambda_1) \) and
  • \( M_2 = (I_2, O_2, S_2, r_2, \delta_2, \lambda_2) \)

• Find the composition
  • \( M = (I, O, S, r, \delta, \lambda) \)

• given a set of constraints of the form:
  • \( C = \{ (o, i_1, \ldots, i_n) : o \text{ is connected to } i_1, \ldots, i_n \} \)
**FSM Composition**

- **Unconditional product** $M' = (I', O', S', r', \delta', \lambda')$
  - $I' = I, U I_2$
  - $O' = O, U O_2$
  - $S' = S_1 x S_2$
  - $r' = r_1 x r_2$
  - $\delta' = \{ (A_1 U A_2, s_1 x s_2, t_1 x t_2) : (A_1, s_1, t_1) \in \delta_1 \text{ and } (A_2, s_2, t_2) \in \delta_2 \}$
  - $\lambda' = \{ (A_1 U A_2, s_1 x s_2, B_1 U B_2) : (A_1, s_1, B_1) \in \lambda_1 \text{ and } (A_2, s_2, B_2) \in \lambda_2 \}$

- **Note:**
  - $A_1 \subseteq I_1, A_2 \subseteq I_2, B_1 \subseteq O_1, B_2 \subseteq O_2$
  - $2^X U Y = 2^X \times 2^Y$

---

**FSM Composition**

- **Constraint application**
  - $\lambda = \{ (A_1 U A_2, s_1 x s_2, B_1 U B_2) : (o, i_1, \ldots, i_n) \in C \quad o \in B_1 U B_2 \quad \text{if and only if} \quad i_j \in A_1 U A_2 \quad \text{for all } j \}$

- The application of the constraint rules out the cases where the connected input and output have different values (present/absent).
FSM Composition

- \( I = I_1 \cup I_2 \)
- \( O = O_1 \cup O_2 \)
- \( S = S_1 \times S_2 \)
- Assume that \( o_1 \in O_1, i_2 \in I_2, o_1 = i_2 \) (communication)
- \( \delta \) and \( \lambda \) are such that, e.g., for each pair:
  - \( \delta(\{i_1\}, s_1) = t_1, \lambda(\{i_1\}, s_1) = \{o_1\} \)
  - \( \delta(\{i_2, i_3\}, s_2) = t_2, \lambda(\{i_2, i_3\}, s_2) = \{o_2\} \)
- we have:
  - \( \delta(\{i_1, i_2\}, s_1, s_2) = (t_1, t_2) \)
  - \( \lambda(\{i_1, i_2, i_3\}, s_1, s_2) = \{o_1, o_2\} \)
- i.e. \( i_3 \) is in input pattern iff \( o_1 \) is in output pattern

FSM Composition

- Problem: what if there is a cycle?
  - Moore machine: \( \delta \) depends on input and state, \( \lambda \) only on state
    - composition is always well-defined
  - Mealy machine: \( \delta \) and \( \lambda \) depend on input and state
    - composition may be undefined
    - what if \( \lambda(\{i_1\}, s_1) = \{o_1\} \) but \( o_2 \in \lambda(\{i_2\}, s_2) \)?

- Causality analysis in Mealy FSMs (Berry '98)
Moore vs. Mealy

- Theoretically, same computational power (almost)
- In practice, different characteristics
- Moore machines:
  - non-reactive
    (response delayed by 1 cycle)
  - easy to compose
    (always well-defined)
  - good for implementation
    - software is always “slow”
    - hardware is better when I/O is latched

Moore vs. Mealy

- Mealy machines:
  - reactive
    (0 response time)
  - hard to compose
    (problem with combinational cycles)
  - problematic for implementation
    - software must be “fast enough”
      (synchronous hypothesis)
    - may be needed in hardware, for speed
Hierarchical FSM models

- 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 pre-emption and exceptions

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

Non-determinism:
- used to abstract behavior

Models Of Computation for reactive systems

- Main MOCs:
  - Communicating Finite State Machines
  - Dataflow Process Networks
  - Petri Nets
  - Discrete Event
  - Codesign Finite State Machines

- Main languages:
  - StateCharts
  - Esterel
  - Dataflow networks
**StateCharts**

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

**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** : no parent states (top of hierarchy)
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:
  - event condition and action are optional
  - an event can be the changing of a value
  - standard comparisons are allowed as conditions and assignment statements 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: $\text{start}(A), \text{stopped}(B)$
**Graphical Hierarchical FSM Languages**

- Multitude of commercial and non-commercial variants:
  - StateCharts, UML, StateFlow, ...
- Easy to use for control-dominated systems
- Simulation (animated), SW and HW synthesis
- Original StateCharts have problems with causality loops and instantaneous events:
  - circular dependencies can lead to paradoxes
  - behavior is implementation-dependent
  - not a truly synchronous language
- Hierarchical states necessary for complex reactive system specification

**Synchronous vs. Asynchronous FSMs**

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

- **A-synchronous FSMs:**
  - 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

**Asynchronous communication**

- **Blocking vs. non-Blocking**
  - Blocking read
    - process can not test for emptiness of input
    - must wait for input to arrive before proceed
  - Blocking write
    - process must wait for successful write before continue
  - blocking write/blocking read (CSP, CCS)
  - non-blocking write/blocking read (FIFO, CFSMs, SDL)
  - non-blocking write/non-blocking read (shared variables)
Asynchronous communication

- Buffers used to adapt when sender and receiver have different rate
  - what size?
- 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

Communication Mechanisms

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

<table>
<thead>
<tr>
<th></th>
<th>Transmitters</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</td>
<td>one</td>
<td>unbounded</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Bounded FIFO</td>
<td>one</td>
<td>one</td>
<td>bounded</td>
<td>no</td>
<td>maybe</td>
<td>yes</td>
</tr>
<tr>
<td>Single Rendezvous</td>
<td>one</td>
<td>one</td>
<td>one</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Multiple Rendezvous</td>
<td>many</td>
<td>many</td>
<td>one</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>