EECS249: Models of Computation

Alberto Sangiovanni Vincentelli
Luciano Lavagno
Roberto Passerone
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
Design Process

Abstract Specification

More Refined Spec

C-code
Assembly
Bits

RTL

Gate Level

XNF

Mask

“Implementation”

Design Flow
Informal specification leads to:
- unambiguous specification
- various stages not logically connected
- costly redesign

We need:
- Formal specification
- Set of Properties
- Set of Performance Indices
- Set of Constraints

E.g.
- await i; emit o
- AG i → o
- Time (i,o)
- Time (i,o) < 10 s

Formal Model of Design: why?
Functions vs. computation

- Functions specify only a relation between two sets of variables (input and output).
- Computations describe how the output variables can be derived from the value of the input variables.
Model of Computation

- A MoC is a framework in which to express what sequence of actions must be taken to complete a computation.
- An instance of a model of computation is a representation of a function under a particular interpretation of its constituents.
- Not necessarily a bijection (in fact almost never!)
- Examples: Finite State Machine, Turing Machine, differential equation.
Model relationship

MODEL

MOC

FUNCTION

OBJECT

interpretation

interpretation

interpretation

interpretation
Partial recursive functions

- Functions need not be computable
- A particular set of functions is obtained by recursive definitions with unbounded search. They are called Partial Recursive Functions.
- Church’s Thesis: “The set of computable functions is the set of Partial Recursive Functions”
- Accept or reject!
- Models which can represent all partial recursive functions are called Turing-complete
- Example: Turing Machines
So why different models?

- Different models = different properties
- Turing-complete models are too powerful!
- Some problems may be undecidable
Properties of a Design

Property:
A (redundant) assertion of the behavior of a design

Verification of Property:  
Inherent in model of computation  
E.g. Determinacy

or  
Verify syntactically

or  
Verify semantically

Dataflow network
FSM network
Petri net
Need to distinguish between *model* and *language*

Language needs to
- be expressive enough for application domain
  (write things once…)
- have semantics in desired MOC

MOC needs to
- be powerful enough for application domain
- have appropriate synthesis and validation algorithms
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
Administra-trivial

- Who is taking the class?
- Are you all on the mailing list?
- Pair up for paper presentation.
- Projects: if you have one, propose it!
- OK to have group projects.
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
Finite State Machines

- Functional decomposition into states of operation
- Inputs and outputs are sequences of events
- 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

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
- δ: 2^I × S → S
  - e.g. δ( {KEY_OFF}, WAIT ) = OFF
- λ: 2^I × S → 2^O
  - e.g. λ( {KEY_ON}, OFF ) = {START_TIMER}
  - note: self-loop not implied in the function
Non-deterministic FSMs

- $\delta$ and $\lambda$ may be \textit{relations} instead of \textit{functions}:
  - $\delta \subseteq 2^I \times S \times S$
  - $\lambda \subseteq 2^I \times S \times 2^O$

  e.g. $\delta(\{\text{KEY_OFF, END_TIMER_5}\}, \text{WAIT}) = \{\{\text{OFF}\}, \{\text{ALARM}\}\}$

- 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:

  0 \rightarrow 1 \rightarrow \ldots \rightarrow 7 \rightarrow 8

  \text{BIT or not BIT} \Rightarrow 

  \text{BIT or not BIT} \Rightarrow 

  \text{BIT or not BIT} \Rightarrow \text{ERR}

- Then completed as even parity:

  0 \rightarrow p1 \rightarrow \ldots \rightarrow p7 \rightarrow 8

  \text{NOT BIT} \Rightarrow 

  \text{BIT} \Rightarrow 

  \text{BIT} \Rightarrow \text{ERR}
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

START => SEC => SEC => END

START => 1 SEC => 2 SEC => 3 SEC => 4 SEC =>

0 SEC => END

1 SEC => 0

2 SEC => 0

3 SEC => 0

4 SEC => 0

5 SEC => 0

6 SEC => 0

7 SEC => 0

8 SEC => 0

9 SEC => 0

END SEC => SEC => SEC => SEC =>
NDFSMs and FSMs

- Formally FSMs and NDFSMs are equivalent (Rabin-Scott construction, Rabin ‘59)
- In practice, NDFSMs are often more compact
- *Language-theoretic* non-determinism (equivalence-oriented) is subtly different from *FSM* non-determinism (containment-oriented)
  - we need one *FSM compatible with the NDFSM*
- Two classes of FSM non-determinism
  - Output (deterministic in language theoretic sense)
  - State
Modeling Concurrency

- Need to compose parts described by FSMs
- Interconnected FSMs specify system
- 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 Diagram ]
FSM Composition

KEY_ON and START_TIMER =>
START_TIMER must be coherent

OFF, 0

not SEC and
(KEY_OFF or BELT_ON) =>

WAIT, 1

SEC and
not (KEY_OFF or BELT_ON) =>

WAIT, 2

not SEC and
(KEY_OFF or BELT_ON) =>

OFF, 1

SEC and
(KEY_OFF or BELT_ON) =>

OFF, 2

Belt Control

Timer
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 $M' = (I', O', S', r', \delta', \lambda')$
  - $I' = I_1 \cup I_2$
  - $O' = O_1 \cup O_2$
  - $S' = S_1 \times S_2$
  - $r' = r_1 \times r_2$
  - $\delta' = \{ (A_1, A_2, s_1, s_2, t_1, t_2) : (A_1, s_1, t_1) \in \delta_1 \text{ and } (A_2, s_2, t_2) \in \delta_2 \}$
  - $\lambda' = \{ (A_1, A_2, s_1, s_2, B_1, B_2) : (A_1, s_1, B_1) \in \lambda_1 \text{ and } (A_2, s_2, B_2) \in \lambda_2 \}$

- Note:
  - $A_1 \subseteq I_1$, $A_2 \subseteq I_2$, $B_1 \subseteq O_1$, $B_2 \subseteq O_2$
  - $2^{x \cup y} \cong 2^x \times 2^y$
FSM Composition

■ Constraint application
  - $\lambda = \{ (A_1, A_2, s_1, s_2, B_1, B_2) \in \lambda' : \text{for all } (o, i_1, \ldots, i_n) \in C$
    \[ o \in B_1 \cup B_2 \text{ if and only if } i_j \in A_1 \cup A_2 \text{ for all } j \} \]

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

■ Outcomes:
  - $\lambda$ is empty or incompletely specified
  - $\lambda$ is a function
  - $\lambda$ is a relation
FSM Composition

- \( I = I_1 \cup I_2 \)
- \( O = O_1 \cup O_2 \)
- \( S = S_1 \times S_2 \)
- Assume that
  \( o_1 \in O_1, i_3 \in I_2, o_1 = i_3 \) (communication)
- \( \delta \) and \( \lambda \) are such that, e.g., for each pair:
  - \( \delta_1(\{ i_1 \}, s_1) = t_1, \quad \lambda_1(\{ i_1 \}, s_1) = \{ o_1 \} \)
  - \( \delta_2(\{ i_2, i_3 \}, s_2) = t_2, \quad \lambda_2(\{ i_2, i_3 \}, s_2) = \{ o_2 \} \)
we have:
  - \( \delta(\{ i_1, i_2, i_3 \}, (s_1, s_2)) = (t_1, t_2) \)
  - \( \lambda(\{ i_1, i_2, i_3 \}, (s_1, s_2)) = \{ o_1, o_2 \} \)
i.e. \( i_3 \) is in input pattern iff \( o_2 \) 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 $\lambda_1(\{i_1\}, s_1) = \{o_1\}$ but $o_2 \not\in \lambda_2(\{i_3\}, s_2)$? Is $o_1$ output or not?

- 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)
    - Esterel compilation algorithm
  - 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:
  - 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
- Extended with arithmetics
- Original StateCharts have problems with *instantaneous reaction* (micro-steps):
  - behavior is implementation-dependent
  - not a truly synchronous language!!!!
Summary of 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)
  ♦ Numerical computations cannot be specified compactly
    (need extended FSMs)
Synchronous Languages

- Assumptions:
  - the system continuously reacts to internal and external events by emitting other events
  - events can occur only at discrete instants
  - zero (negligible) reaction time

- Both control (Esterel) and data flow (Lustre, Signal)

- Very simple syntax and clean semantics (based on FSMs)

- Deterministic behavior

- Simulation, software and hardware synthesis, verification
ESTEREL

- Designed at INRIA by Berry et al.
- Concurrent modules:
  - interface signals, possibly with values
  - local signals and variables
  - statements, e.g.:
    - `await` (single or multiple signals)
    - `do stmt1 watching signal [timeout stmt2]`
      (instantaneous killing of stmt1)
    - `trap exception in stmt1 [handle do stmt2]`
      (allow stmt1 to terminate)
  - allows “external” procedures and functions
Example: readable counter

module counter:
input go, reset, req; output ack(integer);
var t:integer in
loop do
  t:=0;
  every go do
    t:=t+1;
    await req; emit ack(t)
  end
watching reset
end
end.
Other communicating FSM models

- **Synchronous** (Esterel, StateCharts):
  all FSMs make a transition simultaneously

- **A-synchronous** (not synchronous):
  communication is mediated by “channels”:
  - blocking write/blocking read (CSP, CCS)
    (rendez-vous: both partners must be ready)
  - non-blocking write/blocking read (CFSMs, SDL)
    (bounded or unbounded FIFOs)
  - non-blocking write/non-blocking read
    (shared variables)