Outline

- Part 3: Models of Computation
  - FSMs
  - Discrete Event Systems
  - CFSMs
  - Data Flow Models
  - Petri Nets
  - The Tagged Signal Model
Data-flow networks

• A bit of history
• Syntax and semantics
  – actors, tokens and firings
• Scheduling of Static Data-flow
  – static scheduling
  – code generation
  – buffer sizing
• Other Data-flow models
  – Boolean Data-flow
  – Dynamic Data-flow
Data-flow networks

- Powerful formalism for data-dominated system specification
- Partially-ordered model (no over-specification)
- Deterministic execution independent of scheduling
- Used for
  - simulation
  - scheduling
  - memory allocation
  - code generation

for Digital Signal Processors (HW and SW)
A bit of history

• Karp computation graphs (‘66): seminal work
• Kahn process networks (‘58): formal model
• Dennis Data-flow networks (‘75): programming language for MIT DF machine
• Several recent implementations
  – graphical:
    – Ptolemy (UCB), Khoros (U. New Mexico), Grape (U. Leuven)
    – SPW (Cadence), COSSAP (Synopsys)
  – textual:
    – Silage (UCB, Mentor)
    – Lucid, Haskell
Data-flow network

- A Data-flow network is a collection of functional nodes which are connected and communicate over unbounded FIFO queues.
- Nodes are commonly called actors.
- The bits of information that are communicated over the queues are commonly called tokens.
Intuitive semantics

• (Often stateless) actors perform computation

• Unbounded FIFOs perform communication via sequences of tokens carrying values
  – integer, float, fixed point
  – matrix of integer, float, fixed point
  – image of pixels

• State implemented as self-loop

• Determinacy:
  – unique output sequences given unique input sequences
  – Sufficient condition: blocking read
  – (process cannot test input queues for emptiness)
Intuitive semantics

• At each time, one actor is **fired**
• When firing, actors **consume** input tokens and **produce** output tokens
• Actors can be fired only if there are enough tokens in the input queues
Intuitive semantics

• Example: FIR filter
  – single input sequence i(n)
  – single output sequence o(n)
  – o(n) = c1 i(n) + c2 i(n-1)
Intuitive semantics

- Example: FIR filter
  - single input sequence $i(n)$
  - single output sequence $o(n)$
  - $o(n) = c_1 i(n) + c_2 i(n-1)$
Intuitive semantics

- Example: FIR filter
  - single input sequence $i(n)$
  - single output sequence $o(n)$
  - $o(n) = c_1 i(n) + c_2 i(n-1)$
Intuitive semantics

- Example: FIR filter
  - single input sequence \( i(n) \)
  - single output sequence \( o(n) \)
  - \( o(n) = c_1 i(n) + c_2 i(n-1) \)
Intuitive semantics

- Example: FIR filter
  - single input sequence $i(n)$
  - single output sequence $o(n)$
  - $o(n) = c_1 i(n) + c_2 i(n-1)$
Intuitive semantics

- Example: FIR filter
  - single input sequence \(i(n)\)
  - single output sequence \(o(n)\)
  - \(o(n) = c_1 i(n) + c_2 i(n-1)\)
Intuitive semantics

• Example: FIR filter
  – single input sequence $i(n)$
  – single output sequence $o(n)$
  – $o(n) = c_1 \, i(n) + c_2 \, i(n-1)$
Intuitive semantics

• Example: FIR filter
  – single input sequence i(n)
  – single output sequence o(n)
  – o(n) = c1 i(n) + c2 i(n-1)
Intuitive semantics

• Example: FIR filter
  – single input sequence $i(n)$
  – single output sequence $o(n)$
  – $o(n) = c_1 i(n) + c_2 i(n-1)$
Intuitive semantics

- Example: FIR filter
  - single input sequence $i(n)$
  - single output sequence $o(n)$
  - $o(n) = c_1 i(n) + c_2 i(n-1)$
Questions

• Does the order in which actors are fired affect the final result?
• Does it affect the “operation” of the network in any way?
• Go to Radio Shack and ask for an unbounded queue!!
Formal semantics: sequences

- Actors operate from a sequence of input tokens to a sequence of output tokens.
- Let tokens be noted by $x_1$, $x_2$, $x_3$, etc…
- A sequence of tokens is defined as $X = [x_1, x_2, x_3, ...]$
- Over the execution of the network, each queue will grow a particular sequence of tokens.
- In general, we consider the actors mathematically as functions from sequences to sequences (not from tokens to tokens).
Ordering of sequences

• Let $X_1$ and $X_2$ be two sequences of tokens.
• We say that $X_1$ is less than $X_2$ if and only if (by definition) $X_1$ is an initial segment of $X_2$
• Homework: prove that the relation so defined is a partial order (reflexive, antisymmetric and transitive)
• This is also called the prefix order
• Example: $[x_1, x_2] \leq [x_1, x_2, x_3]$
• Example: $[x_1, x_2]$ and $[x_1, x_3, x_4]$ are incomparable
Chains of sequences

- Consider the set $S$ of all finite and infinite sequences of tokens
- This set is partially ordered by the prefix order
- A subset $C$ of $S$ is called a chain iff all pairs of elements of $C$ are comparable
- If $C$ is a chain, then it must be a linear order inside $S$ (otherwise, why call it chain?)

- Example: $\{ [x_1], [x_1, x_2], [x_1, x_2, x_3], \ldots \}$ is a chain
- Example: $\{ [x_1], [x_1, x_2], [x_1, x_3], \ldots \}$ is not a chain
(Least) Upper Bound

- Given a subset $Y$ of $S$, an upper bound of $Y$ is an element $z$ of $S$ such that $z$ is larger than all elements of $Y$.

- Consider now the set $Z$ (subset of $S$) of all the upper bounds of $Y$.

- If $Z$ has a least element $u$, then $u$ is called the least upper bound (lub) of $Y$.

- The least upper bound, if it exists, is unique.

- Note: $u$ might not be in $Y$ (if it is, then it is the largest value of $Y$).
Complete Partial Order

• Every chain in S has a least upper bound
• Because of this property, S is called a Complete Partial Order
• Notation: if C is a chain, we indicate the least upper bound of C by lub( C )
• Note: the least upper bound may be thought of as the limit of the chain
Processes

- Process: function from a p-tuple of sequences to a q-tuple of sequences

\[ F : S^p \rightarrow S^q \]

- Tuples have the induced point-wise order:

\[ Y = (y_1, \ldots, y_p), \ Y' = (y'_1, \ldots, y'_p) \text{ in } S^p : Y \leq Y' \text{ if } y_i \leq y'_i \text{ for all } 1 \leq i \leq p \]

- Given a chain C in \( S^p \), \( F(C) \) may or may not be a chain in \( S^q \)

- We are interested in conditions that make that true
Continuity and Monotonicity

• Continuity: \( F \) is continuous iff (by definition) for all chains \( C \), \( \text{lub}( F( C ) ) \) exists and

\[
F( \text{lub}( C ) ) = \text{lub}( F( C ) )
\]

• Similar to continuity in analysis using limits

• Monotonicity: \( F \) is monotonic iff (by definition) for all pairs \( X, X' \)

\[
X \leq X' \Rightarrow F( X ) \leq F( X' )
\]

• Continuity implies monotonicity
  – intuitively, outputs cannot be “withdrawn” once they have been produced
  – timeless causality. \( F \) transforms chains into chains
Least Fixed Point semantics

• Let $X$ be the set of all sequences
• A network is a mapping $F$ from the sequences to the sequences

$$X = F(X, I)$$

• The behavior of the network is defined as the unique least fixed point of the equation

• If $F$ is continuous then the least fixed point exists $LFP = \text{LUB}( \{ F^n(\perp, I) : n \geq 0 \} )$
From Kahn networks to Data Flow networks

• Each process becomes an *actor*: set of pairs of
  – firing rule
    (number of required tokens on inputs)
  – function
    (including number of consumed and produced tokens)
• Formally shown to be equivalent, but actors with firing are more intuitive
• *Mutually exclusive* firing rules imply monotonicity
• Generally simplified to *blocking read*
Examples of Data Flow actors

- **SDF**: Synchronous (or, better, Static) Data Flow
  - fixed input and output tokens

- **BDF**: Boolean Data Flow
  - control token determines consumed and produced tokens
Static scheduling of DF

• Key property of DF networks: output sequences do not depend on time of firing of actors

• SDF networks can be statically scheduled at compile-time
  – execute an actor when it is known to be fireable
  – no overhead due to sequencing of concurrency
  – static buffer sizing

• Different schedules yield different
  – code size
  – buffer size
  – pipeline utilization
Static scheduling of SDF

- Based only on *process graph* (ignores functionality)
- Network state: number of tokens in FIFOs
- Objective: find schedule that is *valid*, i.e.:
  - *admissible*
    (only fires actors when fireable)
  - *periodic*
    (brings network back to initial state firing each actor at least once)
- Optimize cost function over admissible schedules
Balance equations

- Number of produced tokens must equal number of consumed tokens on every edge

- Repetitions (or firing) vector $v_S$ of schedule $S$: number of firings of each actor in $S$
  - $v_S(A) \ n_p = v_S(B) \ n_c$
  - must be satisfied for each edge
Balance equations

- Balance for each edge:
  - $3v_S(A) - v_S(B) = 0$
  - $v_S(B) - v_S(C) = 0$
  - $2v_S(A) - v_S(C) = 0$
  - $2v_S(A) - v_S(C) = 0$
Balance equations

- \( M \cdot v_S = 0 \)
  - iff \( S \) is periodic

- Full rank (as in this case)
  - no non-zero solution
  - no periodic schedule

\( M = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix} \)

(too many tokens accumulate on A->B or B->C)
Balance equations

- Non-full rank
  - infinite solutions exist (linear space of dimension 1)
- Any multiple of \( q = [1 \ 2 \ 2]^T \) satisfies the balance equations
- ABCBC and ABBCC are minimal valid schedules
- ABABBCBCCC is non-minimal valid schedule

\[
M = \begin{bmatrix}
2 & -1 & 0 \\
0 & 1 & -1 \\
2 & 0 & -1 \\
2 & 0 & -1
\end{bmatrix}
\]
Static SDF scheduling

- Main SDF scheduling theorem (Lee ‘86):
  - A connected SDF graph with $n$ actors has a periodic schedule iff its topology matrix $M$ has rank $n-1$
  - If $M$ has rank $n-1$ then there exists a unique smallest integer solution $q$ to $M q = 0$

- Rank must be at least $n-1$ because we need at least $n-1$ edges (connected-ness), providing each a linearly independent row

- Admissibility is not guaranteed, and depends on initial tokens on cycles
Admissibility of schedules

- No admissible schedule:
  BACBA, then deadlock…

- Adding one token (delay) on A->C makes
  BACBACBA valid

- Making a periodic schedule admissible is always possible, but changes specification…
Admissibility of schedules

- Adding initial token changes FIR order
From repetition vector to schedule

- Repeatedly schedule fireable actors up to number of times in repetition vector
  \[ q = [1 \ 2 \ 2]^T \]

- Can find either ABCBC or ABBCC
- If deadlock before original state, no valid schedule exists (Lee ‘86)
From schedule to implementation

- Static scheduling used for:
  - behavioral simulation of DF (extremely efficient)
  - code generation for DSP
  - HW synthesis (Cathedral by IMEC, Lager by UCB, …)

- Issues in code generation
  - execution speed (pipelining, vectorization)
  - code size minimization
  - data memory size minimization (allocation to FIFOs)
  - processor or functional unit allocation
Compilation optimization

• Assumption: code stitching
  (chaining custom code for each actor)
• More efficient than C compiler for DSP
• Comparable to hand-coding in some cases
• Explicit parallelism, no artificial control dependencies
• Main problem: memory and processor/FU allocation depends on scheduling, and vice-versa
Code size minimization

- Assumptions (based on DSP architecture):
  - subroutine calls expensive
  - fixed iteration loops are cheap
    (“zero-overhead loops”)
- Absolute optimum: *single appearance schedule*
  e.g. ABCBC -> A (2BC), ABBCC -> A (2B) (2C)
  - may or may not exist for an SDF graph…
  - buffer minimization relative to single appearance schedules
    (Bhattacharyya ‘94, Lauwereins ‘96, Murthy ‘97)
Buffer size minimization

• Assumption: no buffer sharing

• Example:

\[ q = [100 \ 100 \ 10 \ 1]^T \]

• Valid SAS: (100 A) (100 B) (10 C) D
  - requires 210 units of buffer area

• Better (factored) SAS: (10 (10 A) (10 B) C) D
  - requires 30 units of buffer areas, but…
  - requires 21 loop initiations per period (instead of 3)
Dynamic scheduling of DF

• SDF is limited in modeling power
  – no run-time choice
  – cannot implement Gaussian elimination with pivoting

• More general DF is too powerful
  – non-Static DF is Turing-complete (Buck ‘93)
    – bounded-memory scheduling is not always possible

• BDF: semi-static scheduling of special “patterns”
  – if-then-else
  – repeat-until, do-while

• General case: thread-based dynamic scheduling
  – (Parks ‘96: may not terminate, but never fails if feasible)
Example of Boolean DF

- Compute absolute value of average of $n$ samples
Example of general DF

- Merge streams of multiples of 2 and 3 in order (removing duplicates)

```
a = get (A)
b = get (B)
forever {
    if (a > b) {
        put (O, a)
a = get (A)
    } else if (a < b) {
        put (O, b)
b = get (B)
    } else {
        put (O, a)
a = get (A)
b = get (B)
    }
}
```
Summary of DF networks

• Advantages:
  – Easy to use (graphical languages)
  – Powerful algorithms for
    – verification (fast behavioral simulation)
    – synthesis (scheduling and allocation)
  – Explicit concurrency

• Disadvantages:
  – Efficient synthesis only for restricted models
    – (no input or output choice)
  – Cannot describe reactive control (blocking read)
Outline

- Part 3: Models of Computation
  - FSMs
  - Discrete Event Systems
  - CFSMs
  - Data Flow Models
  - Petri Nets
  - The Tagged Signal Model