Outline

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

• Kahn Process Networks
  – sequential processes

• Dataflow Networks
  – actors, tokens and firings

• 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

• Used for
  – simulation
  – scheduling
  – memory allocation
  – code generation

  for Digital Signal Processors

• Partially-ordered model (no over-specification)

• Deterministic execution independent of scheduling
  (output sequences do not depend on order or \textit{time of firing} of actors)
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
Kahn Process networks

- Network of sequential processes running concurrently and communicating through single-sender single-receiver unbounded FIFOs.
- Process: mapping from input sequences to output sequences of tokens (streams)
- Blocking read: process attempting to read from empty channel stalls until the buffer has enough tokens
- Determinacy: the order in which processes are fired does not affect the final result (unique output sequences for unique input sequences)
- Difficult to schedule
  - Dataflow Networks
Determinacy

- Process: “continuous mapping” of input sequence to output sequences
- Continuity: process uses prefix of input sequences to produce prefix of output sequences. Adding more tokens does not change the tokens already produced
- The state of each process depends on token values rather than their arrival time
- Unbounded FIFO: the speed of the two processes does not affect the sequence of data values
Scheduling

- Multiple choices for ordering process execution
- Dynamic scheduling
  - Context switching overhead
- Data-driven scheduling
  - Run processes as soon as data is available
- Demand-driven scheduling
  - Run a process when its output is needed as input by another process
- Bounded scheduling [Parks96]
  - Define bounds on buffers, if program deadlocks extend capacity
Data-flow networks

- A Data-flow network is a collection of actors which are connected and communicate over unbounded FIFO queues
- Actors firing follows firing rules
  - Firing rule: number of required tokens on inputs
  - Function: number of consumed and produced tokens
- Breaking processes of KPNs down into smaller units of computation makes implementation easier (scheduling)
- Tokens carry values
  - integer, float, audio samples, image of pixels
- Network state: number of tokens in FIFOs
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
Filter example

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

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

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

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

- Example: FIR filter
  - single input sequence \( i(n) \)
  - single output sequence \( o(n) \)
  - \( o(n) = c1 \ i(n) + c2 \ i(n-1) \)
Filter example

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

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

\[ i * c_1 * c_2 * c_1 + \]
Filter example

• Example: FIR filter
  – single input sequence \( i(n) \)
  – single output sequence \( o(n) \)
  – \( o(n) = c1 \, i(n) + c2 \, i(n-1) \)
Filter example

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

- Example: FIR filter
  - single input sequence i(n)
  - single output sequence o(n)
  - o(n) = c1 i(n) + c2 i(n-1)
Filter example

- **Example: FIR filter**
  - single input sequence $i(n)$
  - single output sequence $o(n)$
  - $o(n) = c_1 i(n) + c_2 i(n-1)$
Examples of Data Flow actors

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

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

- Number of tokens produced and consumed in each firing is fixed
- SDF networks can be \textit{statically scheduled} at compile-time
  - no runtime 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)

• Objective: find schedule that is *valid*, i.e.:
  – *admissible*
    (only fires actors when fireable)
  – *periodic (cyclic schedule)*
    (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

\[ v_S(A) \, n_p = v_S(B) \, n_c \]

must be satisfied for each edge

- Repetitions (or firing) vector \( v_S \) of schedule \( S \): number of firings of each actor in \( S \)
Balance equations

- \( 3v_s(A) - v_s(B) = 0 \)
- \( v_s(B) - v_s(C) = 0 \)
- \( 2v_s(A) - v_s(C) = 0 \)
- \( 2v_s(A) - v_s(C) = 0 \)
Balance equations

- $M v_S = 0$ iff $S$ is periodic
- Full rank (as in this case)
  - no non-zero solution
  - no periodic schedule
  
  (too many tokens accumulate on A->B or B->C)

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

\[ M = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{bmatrix} \]
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
- ABABBCBCCCC is non-minimal valid schedule
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 on A->C makes BACBACBA valid
- Making a periodic schedule admissible is always possible, but changes specification...
From repetition vector to schedule

- Repeatedly schedule fireable actors up to number of times in repetition vector
  \[ q = \begin{bmatrix} 1 & 2 & 2 \end{bmatrix}^T \]

- 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, ...)

- Code generation by *code stitching*
  (chaining custom code for each actor)

- Issues in code generation
  - execution speed (pipelining, vectorization)
  - code size minimization
  - data memory size minimization (allocation to FIFOs)
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:

\[ q = \begin{bmatrix} 100 & 100 & 10 & 1 \end{bmatrix}^T \]

- 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

- **General case: thread-based dynamic scheduling (Parks ‘96)**
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)
Petri Nets

Marco Sgroi
(sgroi@eecs.berkeley.edu)
EE249 - Fall 2003
Outline

• Petri nets
  – Introduction
  – Examples
  – Properties
  – Analysis techniques
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
- Basic PN model
  - Asynchronous model (partial ordering)
  - Main drawback: no hierarchy
Petri Net Graph

- Bipartite weighted directed graph:
  - Places: circles
  - Transitions: bars or boxes
  - Arcs: arrows labeled with weights
- Tokens: black dots
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 \((m_1,m_2,m_3…)\), 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 \(M_0\).

Places/Transition: conditions/events
Transition firing rule

- A marking is changed according to the following rules:
  - A transition is **enabled** if there are enough tokens in each input place
  - An enabled transition **may or may not fire**
  - The firing of a transition modifies marking by **consuming** tokens from the input places and **producing** tokens in the output places
Concurrency, causality, choice

Concurrent executions:
- t1 → t2
- t3 → t4 → t5 → t6
Concurrency, causality, choice

Concurrency

Concurrency, causality, choice

Causality, sequencing

\[ \text{t1} \rightarrow \text{t2} \rightarrow \text{t3} \rightarrow \text{t4} \rightarrow \text{t5} \rightarrow \text{t6} \]
Concurrency, causality, choice
Concurrency, causality, choice

Choice, conflict
Communication Protocol

P1

- Send msg
- Receive Ack
- Receive Ack

P2

- Receive msg
- Send Ack

47
Communication Protocol

P1

Send msg

Receive Ack

Receive msg

Send Ack

P2
Communication Protocol

P1

Send msg

Receive Ack

Send Ack

Receive msg

P2

Receive Ack

49
Communication Protocol

P1

Send msg

Receive Ack

Send Ack

Receive msg

P2

Receive Ack
Communication Protocol

P1

Send msg
Receive Ack
Send Ack
Receive msg
Send Ack
Receive msg

P2
Communication Protocol

P1

Receive Ack

Send msg

Receive msg

Send Ack

Receive Ack

P2
Producer-Consumer Problem
Producer-Consumer Problem

Produce

Buffer

Consume
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem

Produce → Buffer → Consume
Producer-Consumer Problem
Producer-Consumer Problem

Produce

Buffer

Consume
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem

Produce

Buffer

Consume
Producer-Consumer Problem

Produce

Buffer

Consume
Producer-Consumer with priority

Consumer B can consume only if buffer A is empty.

Inhibitor arcs
PN properties

• Behavioral: depend on the initial marking (most interesting)
  – Reachability
  – Boundedness
  – Schedulability
  – Liveness
  – Conservation

• Structural: do not depend on the initial marking (often too restrictive)
  – Consistency
  – Structural boundedness
Reachability

- Marking M is **reachable** from marking M₀ if there exists a sequence of firings \( \sigma = M₀ t₁ M₁ t₂ M₂ \ldots M \) that transforms M₀ to M.

- The reachability problem is decidable.

\[
\begin{align*}
M₀ &= (1,0,1,0) \\
M₁ &= (1,0,0,1) \\
M &= (1,1,0,0)
\end{align*}
\]
Liveness

- **Liveness**: from any marking any transition can become fireable
  - Liveness implies deadlock freedom, not vice versa
Liveness

- **Liveness**: from any marking any transition can become fireable
  - Liveness implies deadlock freedom, not vice versa
Liveness

- **Liveness**: from any marking any transition can become fireable
  - Liveness implies deadlock freedom, not vice versa
Liveness

- **Liveness**: from any marking any transition can become fireable
  - Liveness implies deadlock freedom, not vice versa
Boundedness

• **Boundedness**: the number of tokens in any place cannot grow indefinitely
  
  – (1-bounded also called *safe*)

  – Application: places represent buffers and registers (check there is no overflow)
Boundedness

- **Boundedness**: the number of tokens in any place cannot grow indefinitely
  - (1-bounded also called *safe*)
  - Application: places represent buffers and registers (check there is no overflow)
Boundedness

- **Boundedness**: the number of tokens in any place cannot grow indefinitely
  - (1-bounded also called *safe*)
  - **Application**: places represent buffers and registers (check there is no overflow)
Boundedness

- **Boundedness**: the number of tokens in any place cannot grow indefinitely
  - (1-bounded also called *safe*)
  - Application: places represent buffers and registers (check there is no overflow)
• **Boundedness**: the number of tokens in any place cannot grow indefinitely
  
  – (1-bounded also called *safe*)
  
  – Application: places represent buffers and registers (check there is no overflow)
Conservation

- **Conservation**: the total number of tokens in the net is constant
Conservation

- Conservation: the total number of tokens in the net is constant

Not conservative
Conservation

- **Conservation**: the total number of tokens in the net is constant
Analysis techniques

• **Structural analysis techniques**
  – Incidence matrix
  – T- and S- Invariants

• **State Space Analysis techniques**
  – Coverability Tree
  – Reachability Graph
Incidence Matrix

- Necessary condition for marking $M$ to be reachable from initial marking $M_0$:
  
  there exists firing vector $v$ s.t.:
  
  $$M = M_0 + A \cdot v$$
State equations

- E.g. reachability of $M = |0 0 1|^T$ from $M_0 = |1 0 0|^T$

$$A = \begin{bmatrix} -1 & 0 & 0 \\ 1 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$

$$v_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ 0 & 1 & 1 & -1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

but also $v_2 = |1 1 2|^T$ or any $v_k = |1 (k) (k+1)|^T$
Necessary Condition only

Firing vector: (1,2,2)  Deadlock!!
State equations and invariants

- Solutions of $Ax = 0$ (in $M = M_0 + Ax, M = M_0$)

**T-invariants**
- sequences of transitions that (if fireable) bring back to original marking
- periodic schedule in SDF
- e.g. $x = |0 1 1|^T$

$$A = \begin{pmatrix}
-1 & 0 & 0 \\
1 & 1 & 0 \\
0 & -1 & 1
\end{pmatrix}$$
Application of T-invariants

- **Scheduling**
  - *Cyclic schedules*: need to return to the initial state

\[
\text{Schedule: } i \cdot k_2 \cdot k_1 + o
\]

\[
\text{T-invariant: } (1,1,1,1,1)
\]

87
State equations and invariants

- Solutions of $yA = 0$

**S-invariants**
- sets of places whose weighted total token count does not change after the firing of any transition ($yM = yM'$)
- e.g. $y = |1 1 1 |^T$

$$A^T = \begin{bmatrix} -1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & -1 & 1 \end{bmatrix}$$
Application of S-invariants

• Structural Boundedness: bounded for any finite initial marking $M_0$

• Existence of a positive S-invariant is CS for structural boundedness
  – initial marking is finite
  – weighted token count does not change
Summary of algebraic methods

- Extremely efficient (polynomial in the size of the net)
- Generally provide only necessary or sufficient information
- Excellent for ruling out some deadlocks or otherwise dangerous conditions
- Can be used to infer structural boundedness
Coverability Tree

- Build a (finite) tree representation of the markings

**Karp-Miller algorithm**

- Label initial marking $M_0$ as the root of the tree and tag it as *new*
- While new markings exist do:
  - select a new marking $M$
  - if $M$ is identical to a marking on the path from the root to $M$, then tag $M$ as *old* and go to another new marking
  - if no transitions are enabled at $M$, tag $M$ *dead-end*
  - while there exist enabled transitions at $M$ do:
    - obtain the marking $M'$ that results from firing $t$ at $M$
    - on the path from the root to $M$ if there exists a marking $M''$ such that $M'(p) \geq M''(p)$ for each place $p$ and $M'$ is different from $M''$, then replace $M'(p)$ by $\omega$ for each $p$ such that $M'(p) > M''(p)$
    - introduce $M'$ as a node, draw an arc with label $t$ from $M$ to $M'$ and tag $M'$ as *new*.
Coverability Tree

- Boundedness is decidable

with *coverability tree*
• Boundedness is decidable

with coverability tree
Coverability Tree

- Boundedness is decidable
  with *coverability tree*
Coverability Tree

- Boundedness is decidable
  with *coverability tree*
Coverability Tree

- Boundedness is decidable
  with *coverability tree*
Coverability Tree

• Is (1) reachable from (0)?
Coverability Tree

- Is (1) reachable from (0)?
Coverability Tree

- Is (1) reachable from (0)?
Coverability Tree

• Is (1) reachable from (0)?

• Cannot solve the reachability problem
Reachability graph

- For bounded nets the Coverability Tree is called Reachability Tree since it contains all possible reachable markings.
For bounded nets the Coverability Tree is called Reachability Tree since it contains all possible reachable markings.
• For bounded nets the Coverability Tree is called Reachability Tree since it contains all possible reachable markings
For bounded nets the Coverability Tree is called Reachability Tree since it contains all possible reachable markings.
Subclasses of Petri nets

- Reachability analysis is too expensive
- State equations give only partial information
- Some properties are preserved by reduction rules e.g. for liveness and safeness
- Even reduction rules only work in some cases
- Must restrict class in order to prove stronger results
Marked Graphs

- Every place has at most 1 predecessor and 1 successor transition
- Models only causality and concurrency (no conflict)
State Machines

- Every transition has at most 1 predecessor and 1 successor place
- Models only causality and conflict
  - (no concurrency, no synchronization of parallel activities)
Free-Choice Petri Nets (FCPN)

Free-Choice (FC)

Confusion (not-Free-Choice)  Extended Free-Choice

Free-Choice: the outcome of a choice depends on the value of a token (abstracted non-deterministically) rather than on its arrival time.
Free-Choice nets

- Introduced by Hack (‘72)
- Extensively studied by Best (‘86) and Desel and Esparza (‘95)
- Can express concurrency, causality and choice without confusion
- Very strong structural theory
  - necessary and sufficient conditions for liveness and safeness, based on decomposition
  - exploits duality between MG and SM
MG (& SM) decomposition

- **Allocation** is a control function that chooses which transition fires among several conflicting ones (A: P → T).
- Eliminate the subnet that would be inactive if we were to use the allocation...
- **Reduction Algorithm**
  - Delete all unallocated transitions
  - Delete all places that have all input transitions already deleted
  - Delete all transitions that have at least one input place already deleted
- Obtain a **Reduction** (one for each allocation) that is a conflict free subnet
MG reduction and cover

• Choose one successor for each conflicting place:
MG reduction and cover

• Choose one successor for each conflicting place:
MG reduction and cover

• Choose one successor for each conflicting place:
MG reduction and cover

- Choose one successor for each conflicting place:
MG reduction and cover

- Choose one successor for each conflicting place:
MG reductions

- The set of all reductions yields a **cover of MG components** (T-invariants)
MG reductions

- The set of all reductions yields a cover of MG components (T-invariants)
Hack’s theorem (‘72)

- Let N be a Free-Choice PN:
  - N has a live and safe initial marking (well-formed)
    if and only if
      - every MG reduction is strongly connected and not empty, and
        the set of all reductions covers the net
      - every SM reduction is strongly connected and not empty, and
        the set of all reductions covers the net
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

• Example of non-live (but safe) FCN
Hack’s theorem

• Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

• Example of non-live (but safe) FCN
Hack’s theorem

• Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN
Hack’s theorem

- Example of non-live (but safe) FCN

Deadlock
Summary of LSFC nets

• Largest class for which structural theory really helps

• Structural component analysis may be expensive
  (exponential number of MG and SM components in the worst case)

• But…
  – number of MG components is generally small
  – FC restriction simplifies characterization of behavior
Petri Net extensions

• Add interpretation to tokens and transitions
  – Colored nets (tokens have value)

• Add time
  – Time/timed Petri Nets (deterministic delay)
    • type (duration, delay)
    • where (place, transition)
  – Stochastic PNs (probabilistic delay)
  – Generalized Stochastic PNs (timed and immediate transitions)

• Add hierarchy
  – Place Charts Nets
PNs Summary

- PN Graph: places (buffers), transitions (actions), tokens (data)
- Firing rule: transition enabled if there are enough tokens in each input place
- Properties
  - Structural (consistency, structural boundedness…)
  - Behavioral (reachability, boundedness, liveness…)
- Analysis techniques
  - Structural (only CN or CS): State equations, Invariants
  - Behavioral: coverability tree
- Reachability
- Subclasses: Marked Graphs, State Machines, Free-Choice PNs
References

• T. Murata Petri Nets: Properties, Analysis and Applications

• http://www.daimi.au.dk/PetriNets/