Formalizing DFGs as Relations

- \((i, j) \in \text{DataFlow}_G\) if node \(i\) creates data that node \(j\) uses in \(G\)
- \(\text{INPUT}_G\) -- set of nodes \(\{\text{input}\}\) such that \(\text{input}\) is a provider of input to \(G\) from an external source
- \(\text{OUTPUT}_G\) -- set of nodes \(\{\text{output}\}\) such that \(\text{output}\) is a conveyor of artifacts computed by \(G\) to an external source
- \((e, \text{operand}) \in \text{EdgeAnnotation}_G\), if \(\text{operand}\) is the name of the artifact that flows along an edge \(e \in E\), where \(G = (N,E)\)
  - Preferably the data artifact is defined rigorously
- \((n, \text{text}) \in \text{NodeAnnotation}_G\), if the string \(\text{text}\) describes the functioning of node \(n \in N\), where \(G = (N,E)\)
  - This may imply different semantics (e.g. abstraction)
- \(\text{InputAnnotation}, \text{OutputAnnotation}\) are similar

Questions this helps answer:
- Why create this data? Who uses this data? What results does the end user see? What does the end user have to input?
- Questions this can’t answer: What is the exact sequence of events? How does a node do its job?
Hierarchy

• Enables incrementally adding detail
• Increased precision too
• Draws upon innate human mental capability
  – Abstraction
  – Encapsulation
• A typical solution to the problem of needing detail, but needing to avoid overload
• But creates potential problems

Consistency is a principal concern

• Are the diagrams consistent with each other?
• Top view consistent with elaborations?
  – Arrows consistent
  – Data flows consistent
  – Other semantics?
• Are we seeing different shadows of the same object?
• Invitation to subtle errors
Simple Hierarchical Elaboration

Check args.

- height
- width
- args OK
- args not OK

- height
- width
- args OK
- args not OK

- valid height
- width > 0
- no
- no
- valid pair
- height > width
- height
- width
- args OK
- args not OK

Something a Little More Complex

Select New Floor

- Current Floor
- View of Request List

Move Elevator

- Floor
- Request Deletion Packet

Update Request List

- Request
- Request List

Button Press

- Turn on Light
- Turn off Light
Hierarchical Elaboration (of “Select New Floor”)

- Current Floor
- Find Nearest Floor
  - Direction
  - Floors in the right direction
- Process Request List
- Request List
- Consider Other Cars
- Do Updates
  - New Status
  - Request Deletion Packet
  - New Floor
  - Rejection
  - Floor to Move to
- Is this consistent with its parent?
Hierarchical Elaboration (of “Select New Floor”)

- Current Floor
- Find Nearest Floor
  - Floors in the right direction
- Consider Other Cars
- Do Updates
- Floor to Move to
- New Status
- Request List
  - View of Request List
  - Request Deletion Packet
- Inconsistencies with parent

Hierarchical Elaboration (of “Select New Floor”)

- Current Floor
- Find Nearest Floor
  - Floors in the right direction
- Consider Other Cars
- Do Updates
- Floor to Move to
- New Status
- Request List
  - View of Request List
  - Request Deletion Packet
- Inconsistencies with parent
(Some) Consistency definitions

- Let \((n, \text{somenode}) \in \text{NodeAnnotation}_{\text{ParentGraph}}\) for some \(n, \text{somenode} \in N\), where \(\text{ParentGraph} = (N, E)\)
- Let \(\text{somenode} = (N', E')\)
  - This is the DFG elaborating on “somenode”
- Some consistency properties
  - If \(\{(m_i, \text{somenode})\} \neq \emptyset\), \(m_i, \text{somenode} \in N\), then \(\text{Cardinality}(\text{INPUT}(\text{somenode})) \neq 0\)
  - If \(\{(\text{somenode}, m_i)\} \neq \emptyset\), \(m_i, \text{somenode} \in N\), then \(\text{Cardinality}(\text{OUTPUT}(\text{somenode})) \neq 0\)
- Maybe some others(?)
  - If \(\text{Cardinality}(\{(m_i, \text{somenode})\}) = k\), \(m_i, \text{somenode} \in N\), then \(\text{Cardinality}(\text{INPUT}(\text{somenode})) = k\)
  - If \(\text{Cardinality}(\{(\text{somenode}, m_i)\}) = k\), \(m_i, \text{somenode} \in N\), then \(\text{Cardinality}(\text{OUTPUT}(\text{somenode})) = k\)
Elaboration of “Select New Floor”
Suggests Artifact Issues

Current Floor

Find Nearest Floor

Floor in the right direction

Floors in the right direction

Direction

Process Request List

View of Request List

Request List

Floor to Move to

New Floor

Rejection

Consider Other Cars

Floor

Do Updates

IDEF0

• Commercial DFG formalism
• Some formality and rigor behind it
• Primarily pictorial
• In wide use
• Additional semantics in IDEF1, IDEF2, etc.
Broadening DFG Semantics

- Node cannot begin until data arrives along all in-edges
  - Let $\text{DFG} = (\text{N, E})$, where $\text{N}$ is set of nodes, $\{n_i\}$. If $n_i$ is a function, then it must be defined on the set of all artifacts $\text{IA}_i = \{\text{ia}_{i,j}\}$, such
    $$(n_j, n_i, \text{ia}_{i,j}) \in \text{EdgeAnnotation}_{\text{DFG}}$$
  - How to adapt this for
  - any semantics?
  - for output semantics?
  - exactly one Output node
  - Etc.

These sorts of constraints can support additional types of reasoning:
- Eg. about parallelism

More Broadening

- Use of "open boxes" to indicate data store
  - A different set, with different semantics
  - Not a computation function
  - Methods are: put, get, search(?)
Still More Broadening

- Different shapes of boxes
- Different pictures instead of boxes
- Different colors
- Different lines
- .......

Central questions: What are the semantics? Does this really help? Or confuse?

Scientific Workflow Graph
Attempt to make this more precise

But iteration complicates matters
Kepler--Another DFG Technology

- Data Flow graph notation
  - And elaborate diagram iconography
- Has hierarchical decomposition
- Capability for specifying DFG semantics
  - For each diagram
  - Can be different at each level of hierarchy (!)
- Based on Ptolemy II system

A Kepler Example
Kepler Elaboration:
Focus of difficulty with iteration

What kinds of questions are well addressed by DFGs?

- Overall structure of functional capabilities
  - What does this piece do?
  - System outputs and inputs
  - How might changes be made?
  - What functions create what data entities
  - Problems with iteration, though
Given that Precision is essential

- What about the other three dimensions?
  - Detail
    - Gained from hierarchical elaboration
  - Breadth
    - Difficult to specify many kinds of iteration
    - Difficult to be clear about concurrency
  - Clarity
    - Seems to be reduced by increased Detail and attempts at Breadth
    - With the need for Precision

Considerable Appeal, but Limited Value, to most stakeholders

- Users think they have sufficient understanding
  - But have trouble being able to see easy things (iteration)
- Developers have same problem
- Managers may only care to see easy things (!)
  - Although they should be interested in more
- Bystanders may be shown only easy things
  - Which could be a real problem
**Final observations**

- Very primitive representation
  -- very limited semantics
- But actually more a family of model types
  -- different sets of semantics
- The actual relation(s) are rarely made clear and precise
- Powerful aid to intuition and efficiency of communication
  -- Clear advantages over natural language
- But is intuition misled by ambiguity, misinterpretation?
- Does not help explain HOW things get done

---

**Control Flow Diagrams**

- Semantics are different from DFD’s semantics
  -- Arrows represent flow of control, rather than flow of data
  -- Different shapes/types of boxes with different semantics
- Basic control flow graph:
  -- Boxes represent functions
  -- Some other shape represents control flow alternation
  -- Arrows represent control flow:
    If there is an arrow from circle A to circle B, it means
    ImmFol: "the execution of B can immediately follow the execution of A for some execution"
  -- Different semantics for arrows between different shapes
  -- Still other shapes represent Start and Stop
  -- Use of hierarchy for elaborating boxes
- Usual enhancements:
  -- Annotate edges with predicates
  -- Special symbols for branching, concurrency control....
Control Flow Diagrams

- Semantics are different from DFD's semantics
  -- Arrows represent flow of control, rather than flow of data
  -- Different shapes/types of boxes with different semantics
- Basic control flow graph:
  -- Boxes represent functions
  -- Some other shape represents control flow alternation
  -- Arrows represent control flow:
    If there is an arrow from circle A to circle B, it means
    ImmFol: "the execution of B can immediately follow
    the execution of A for some execution"
  -- Different semantics for arrows between different shapes
  -- Still other shapes represent Start and Stop
  -- Use of hierarchy for elaborating boxes
- Usual enhancements:
  -- Annotate edges with predicates
  -- Special symbols for branching, concurrency control....

Control flow graphs also address questions like “what does this do” and “how does this do it”

Example Control Flow Graph Semantics

- Square boxes: Functions
  -- Any number of inedges
  -- One outedge
- Edges: ImmFol relation
  -- Function at head executes immediately after
    function at tail concludes
- Round ovals: Branches/decisions
  -- Each oval represents a Boolean function
  -- For each oval there are exactly two outedges, labeled True and False
Example Control Flow Graph Semantics

- \( FG = (FN, FE) \), where 
  \( FN = OPS \cup TEST \) and 
  \( OPS = \{ \text{ops} \} \) nodes where computation is done 
  \( TEST = \{ \text{test} \} \) nodes where tests are done
- Edges: \( \text{ImmFol} \) relation
  if \( (fe_i, fe_k) \in FE \) then there is an execution path for which the execution of \( fe_k \) immediately follows the execution of \( fe_i \)
- \( |(fe_i, fe_k)| \) where \( fe_i \in TEST = 2 \)
- \( |(fe_i, fe_k)| \) where \( fe_i \in OPS = 1 \)
- \( FG \) is a directed, connected graph
- \( \text{Maybe}() \): \( FN = OPS \cup TEST \cup INPUT \cup OUTPUT \) where
  - \( fn_i \in OUTPUT \Rightarrow \exists (fn_i, fn_k) \in FE \)
  - \( fn_i \in INPUT \Rightarrow \exists (fn_k, fn_i) \in FE \)
  - Other semantics?

Control Flow Graph

- Read Floor, Direction
- DB_Read: 
  curr-length, Floors(.), Directions(.)
- Direction = Up
- incr <-- 1
- incr <-- -1
- yes
- no
- NFloor <-- Floor; increment NFloor;
- Directions[I] = incr ?
- other car is nearer ?
- yes
- no
- increment NFloor
- NFloor out of bounds
- Nfloor out of bounds
- complement Direction
- output new floor
Control Flow Graph

Read Floor, Direction

DB_Read:
curr-length, Floors(.),
Directions(.)

Direction = Up

Direction = Up

incr <-- 1

incr <-- -1

yes

no

NFloor <-- Floor;

Directions[NFloor] = incr ?

other car is nearer ?

yes

no

increment NFloor

Complement Direction (Incr <-- -Incr)

NFloor = out of bounds

yes

no

output new floor

More Detail through Hierarchical Elaboration

• The challenges and solutions here are similar to those for DFGs
• Need for consistency between levels
• Semantics can be hard to define, stick to
More Breadth from more (sub)types of CFG

• Concurrency
• Timing
• Data flow augmentations
• Etc.

Concurrency graph with “fork and join”

T1
1
2
3
4
5
   call a
T2
6
7
8
9
10
   accept a

Ra
1
2
3
6
7
8
5
10
Rendezvous Graph

- Rendezvous graph: \( RG = (N, E) \), where \( N = COMP \cup SYNC \)
- \( S \in SYNC \Rightarrow S \) executes only after the execution of all \( C \in COMP \) such that \( (C, S) \in E \)
- If \( (S, C) \in E \), where \( S \in SYNC \), then \( C \) cannot execute until \( S \) has executed

Trace Flow Graph

- Explicitly represents interleaved execution
- The green edges are the ImmFol relation
  - Double-headed arrows represent one arrow in each direction
- Sync nodes are filled in black
- Synch edges are blue
Different variations to address different stakeholders and their needs

- More semantic issues
  - Different graph types for different issues
- More detail
  - Hierarchy helps here
- More clarity
  - Does more elaboration and more graph types help clarity or impede it?

Focus on Clarity

- CFG has more procedural detail
- Does this help?
- What about combining DFG and CFG?
  - Iconography must be clear (?)
Control Flow Graph

Read
Floor, Direction

DB_Read:
curr-length, Floors(.),
Directions(.)

Direction = Up

incr <-- 1

nfloor <-- Floor;
increment nfloor;

Directions[nfloor] = incr ?

other car is nearer ?

output new floor

nfloor out of bounds
complement direction

nfloor = Floors(i) for any 1 <= i <= curr-length

increment nfloor

no

yes

no

yes

no

yes

no

yes

no

yes

no

yes

no

yes
Control Flow Graph with Data Flow

### DB_Read
- `curr-length, Floors(.), Directions(.)`

### Read
- `Floor, Direction`

### Direction
- `Direction = Up`
- `incr <-- 1`
- `incr <-- -1`
  - `yes`
  - `no`

### NFloor
- `NFloor <-- Floor`
- `Directions[NFloor] = incr ?`
  - `yes`
  - `other car is nearer ?`
    - `yes`
    - `no`
  - `no`

### Increment NFloor
- `increment NFloor`
- `Complement Direction (Incr <-- -Incr)`
  - `yes`
  - `no`

### Other car is nearer
- `other car is nearer ?`
  - `yes`
  - `no`

### Out of bounds
- `NFloor out of bounds`
- `output new floor`
What is Control Flow good/not good for?

- Sense of what algorithms to use
- Constraints on data appearing
  - eg. Assuming that the Request List is maintained in sorted order
- Can estimate running speed
- Can reason about functionality
  - Possible to strand requesters
  - Possible to take riders in the wrong direction
- Drawbacks:
  - What about safety?
  - What about data?

What Stakeholders does this address?

- Developers
- ???
Control Flow vs. Data Flow Graphs

- Both shed light on similar questions
- One focuses on data evolution, the other on functional development
- Both are useful, neither removes the need for the other
- Control flow graphs map closely to implementation code written in procedural languages.
  - Good basis for determining consistency of code with ideas expressed as data flow
- Data flow graphs focus more on the product itself, seem better at helping understand if and how it gets evolved
  - Seem better adapted to studying earlier formulations of the problem to be solved, and ways of solving it

Finite State Machines (FSM's)

- FSM's describe behavior of a system:
  -- The sequence of stages/steps/conditions that the system goes through
  -- FSM shows how a system acts/reacts to inputs
  -- Does this by showing progress through different states

- Hypothesis:
  -- The universe in which the system being described must operate can be accurately modeled as always being in exactly one of a finite number of states (situations)
  -- There are only a finite number of possible system inputs
Finite State Machines (FSM's)

- FSM = (Q, I, \( \delta \)), where
  
  Q = \{q_i\}, the set of all possible system states
  
  I = \{i_j\}, the set of all events that can affect the state of the system
  
  \( \delta \) is a function, \( \delta: Q \times I \rightarrow Q \) such that if the system is in state \( q \) and event \( i \) occurs, then the system transitions to state \( \delta(q, i) \)

- Use of hierarchy can help add detail
  - But presents familiar consistency problems

- Other definitions:
  Start state
  Accepting state
  Trap state

Why FSM's?

- Primary appeal is visualizability—clarity
  --Circles represent states
  --(Curved) arrows represent transitions
  --Arrows are annotated with inputs

- Intuitively: Can "watch" a stream of inputs "drive" the behavior of the system as a sequence of movements from state to state

Kinds of Questions FSMs seem adept at helping answer:

- "What is a good way to think about the problem to be solved?"
- "What is the solution approach?"
- "How does this program work?"
Finite State Machine

Doors Closed, Elevator Idle

Doors closed, Elevator in Motion

Request to Move to new floor

Arrival at New Floor

Queueing new floor request

Getting new floor request

Request to Move to new floor

Queue empty

No new request timeout

Doors open, Elevator Stopped

Doors open, Elevator in Motion

Request to Move to new floor

Finite State Machine for Digital Watch

Time display mode

Datebook mode

Alarm display mode

Phone book mode

press button A

press button A

press button A

press button A
More Finite State Machine Details

More Complex FSMs

• State also specifies activities
  – Leans towards a CFG
• Transition may involve computation
  – Considered to be “instantaneous”
• Transition may be conditional
  – Event as well as a condition
• Transition may emit events
  – To drive other FSMs
• FSMs may be hierarchical
  – What exactly are semantics?
Statecharts: More Complex FSMs

Initialize
  do: Initialize course object

Unassigned
  addStudent/
  numStudents = 0
  do: Assign professor to course

Open
  entry: Register a student
  cancelCourse
  registration closed[
    numStudents < 3
  ]
  numStudents = 10
  registration closed[
    numStudents > 3
  ]

Canceled
  addStudent
  do: Send cancellation notices

Closed
  do: Report course is full
  cancelCourse[
    numStudents = 10
  ]
  registration closed[
    numStudents < 3
  ]
  registration closed[
    numStudents >= 3
  ]
  numStudents < 3

Unassigned
  cancelCourse

States

0

CS 520/620 Spring 2013 Univ. of Massachusetts
Copyright L. Osterweil, all rights reserved
Statecharts: More Complex FSMs

Initialize
- do: Initialize course object
- cancelCourse
- addStudent/numStudents = 0

Unassigned
- do: Assign professor to course
- cancelCourse
- addStudent/numStudents = 0

Open
- entry: Register a student
- cancelCourse
- registration closed[ numStudents < 3 ]

Canceled
- do: Send cancellation notices
- cancelCourse
- addStudent/numStudents = 0

Closed
- do: Report course is full
- cancelCourse
- registration closed[ numStudents > 3 ]

RegistrationComplete
- do: Generate class roster
- [ numStudents = 10 ]

Unassigned
- do: Send cancellation notices
- registration closed[ numStudents < 3 ]

Canceled
- do: Send cancellation notices
- registration closed[ numStudents > 3 ]

Closed
- do: Report course is full
- registration closed[ numStudents < 3 ]

RegistrationComplete
- do: Generate class roster
- [ numStudents = 10 ]
Statecharts: More Complex FSMs

- **Initialize**
  - do: Initialize course object

- **Unassigned**
  - do: Assign professor to course

- **Open**
  - entry: Register a student

- **RegistrationComplete**
  - do: Generate class roster

- **Closed**
  - do: Report course is full

- **Canceled**
  - do: Send cancellation notices

- **addStudent**
  - numStudents = 0
  - [numStudents = 10]

- **cancelCourse**
  - registration closed [numStudents < 3]
  - registration closed [numStudents > 3]

Statechart with Nested States

- **Initialize**
  - superstate

- **Unassigned**
  - do: Assign professor to course

- **Open**
  - entry: Register a student

- **RegistrationComplete**
  - do: Generate class roster

- **Closed**
  - do: Report course is closed

- **Canceled**

- **addStudent**
  - numStudents = 0
  - [numStudents = 10]
What is FSM good/not good for?

- Focus on specific issue: safety concern
  - Model unsafe state
  - Model state transitions
  - Can unsafe state be reached?
- Drawbacks
  - No sense of functionality
    » Unless additional semantics
  - No sense of how functionality achieved
    » Except perhaps hierarchy
  - Hard to deal with concurrency
    » Without additional semantics
  - Impossible to reason about timing
    » Unless additional semantics

Petri Nets

- More powerful and intuitive depiction of control flow
  strong on depiction of parallelism and concurrency
- A Petri Net structurally consists of
  - A finite number of places
  - A finite number of transitions
  - A finite set of arrows that connect places to transitions
    (or vice versa)
- If an arrow goes from a place to a transition, then place is said to be an input place of the transition.
- If an arrow goes from a transition to a place, then place is said to be an output place of the transition.
Marking and Firing Petri Nets

- A Petri Net place can be marked by the presence of a token
  - Any collection of places can be marked.
  - Any such marking is said to define a state of the Petri Net
- Petri Nets proceed from one state to another by means of a firing
  - Occurs only when every input place of a transition is marked with a token.
- The effect of the firing of a transition is to
  - Remove all of the tokens from the transition's input places
  - Put tokens in all of the transition's output places

Scenarios Definition with Petri Nets

- A marking represents a scenario
- Applying firing rules creates a simulation
- Different markings support exploring dynamic behavior of a system
More Details in
Ghezzi, et al., Chapter 5.5.3

Unmarked Petri Net

Diagram:
- Car going up
- Up button Pressed
  - Stop at Floor
- Up button Not pressed
  - Go to Next Floor
A Scenario: Car going up with UP button pressed

Car going up

Up button Pressed

Stop at Floor

Go to Next Floor

Up button Not pressed
A Scenario: Car going up with UP button pressed

Unmarked Petri Net--Two Elevator Cars
Petri Net--Two Elevator Cars
Car 1 Wins Race

Petri Net--Two Elevator Cars
Car 1 Wins Race
Petri Net--Two Elevator Cars
Car 1 Wins Race

Petri Net--Two Elevator Cars
Car 2 Wins Race
Petri Net--Two Elevator Cars
Car 2 Wins Race

Car 1 approaching Floor n, going up

No Press
Go to Next Floor Up

Up Button Pressed at Floor n
Stop at Floor

Car 2 approaching Floor n, going up

No Press
Go to Next Floor Up

Up Button Pressed at Floor n
Stop at Floor

Car 2 approaching Floor n, going up

No Press
Go to Next Floor Up

Up Button Pressed at Floor n
Stop at Floor

Car 1 approaching Floor n, going up

No Press
Go to Next Floor Up

Up Button Pressed at Floor n
Stop at Floor

Car 2 approaching Floor n, going up

No Press
Go to Next Floor Up

Up Button Pressed at Floor n
Stop at Floor
Petri Net--Two Elevator Cars
Car 2 Wins Race

Car 1 approaching Floor n, going up
No Press
Go to Next Floor Up
Stop at Floor

Up Button Pressed at Floor n

Car 2 approaching Floor n, going up
No Press
Go to Next Floor Up
Stop at Floor

Car 2 approaching Floor n, going up
Up Button Pressed at Floor n

Car 1 approaching Floor n, going up
No Press
Go to Next Floor Up
Stop at Floor

Up Button Pressed at Floor n

Car 2 approaching Floor n, going up
No Press
Go to Next Floor Up
Stop at Floor

Car 2 approaching Floor n, going up
Up Button Pressed at Floor n

Car 1 approaching Floor n, going up
No Press
Go to Next Floor Up
Stop at Floor

Up Button Pressed at Floor n

Car 2 approaching Floor n, going up
No Press
Go to Next Floor Up
Stop at Floor

Car 2 approaching Floor n, going up
Up Button Pressed at Floor n

Car 1 approaching Floor n, going up
No Press
Go to Next Floor Up
Stop at Floor
Petri Net for a Different Elevator

Marking for moving up to pick up a passenger
Marking for moving up to pick up a passenger

- Car going up
  - Up button higher
  - No Up Button Higher
  - Move up to Nearest Up Button
    - Visit floor, Turn off button
  - No Up Button Lower
  - Car going down
    - Down button lower
    - Move to nearest Down Button
      - Visit floor, Turn off button

Marking for moving up to pick up a passenger

- Car going up
  - Up button higher
  - No Up Button Higher
  - Move up to Nearest Up Button
    - Visit floor, Turn off button
  - No Up Button Lower
  - Car going down
    - Down button lower
    - Move to nearest Down Button
      - Visit floor, Turn off button
Marking for moving up to pick up a passenger

- Car going up
  - Up button higher
  - Move up to Nearest Up Button
  - No Up Button Higher
  - Car going down
  - Down button lower
  - Move to nearest Down Button
  - Visit floor, Turn off button
  - No Down Button Lower

Queues at places?

- Car going up
  - Up button higher
  - Move up to Nearest Up Button
  - No Up Button Higher
  - Car going down
  - Down button lower
  - Move to nearest Down Button
  - Visit floor, Turn off button
  - No Down Button Lower

Queues at places?
Marking when no passengers higher, but passenger lower

Car going up

Up button higher

Move up to Nearest Up Button

Down button lower

Visit floor, Turn off button

Car going down

No Up Button Higher

No Down Button Lower

Move to nearest Down Button

Visit floor, Turn off button
Marking when no passengers higher, but passenger lower

Car going up
- Up button higher
  - No Up Button Higher
  - Move up to Nearest Up Button
    - Visit floor, Turn off button
      - No Down Button Lower
      - Move to nearest Down Button
        - Visit floor
          - Move to nearest Down Button
            - Visit floor

Marking for both up and down buttons

Car going up
- Up button higher
  - No Up Button Higher
  - Move up to Nearest Up Button
    - Visit floor, Turn off button
      - No Down Button Lower
      - Move to nearest Down Button
        - Visit floor
Marking for neither up nor down buttons

- Car going up
  - Up button higher
  - Move up to Nearest Up Button
  - Visit floor
  - Car going down
  - Down button lower
  - Move to nearest Down Button
  - Visit floor

Petri Nets

- More powerful and intuitive depiction of control flow
- Strong on depiction of parallelism and concurrency

- A Petri Net structurally consists of
  - A finite number of places
  - A finite number of transitions
  - A finite set of arrows that connect places to transitions (or vice versa)

- If an arrow goes from a place to a transition, then place is said to be an input place of the transition.
- If an arrow goes from a transition to a place, then place is said to be an output place of the transition.
Marking and Firing Petri Nets

• A Petri Net place can be marked by the presence of a token
  – Any collection of places can be marked.
  – Any such marking is said to define a state of the Petri Net
• Petri Nets proceed from one state to another by means of a firing
  – Occurs only when every input place of a transition is marked with a token.
• The effect of the firing of a transition is to
  – Remove all of the tokens from the transition's input places
  – Put tokens in all of the transition's output places

Some Semantics

• PN = (Places, Transitions, Edges)
  – Places (PN) = \{place\}_i
  – Transitions (PN) = \{transition\}_j
  – Edges (PN) = \{Inedges U Outedges\}
    » Inedges (PN) = \{(place\_i, transition\_j)\} ∧ Outedges (PN) = \{(transition\_i, place\_j)\},
        where place\_i ∈ Places (PN), transition\_j ∈ Transitions (PN)
• Marked: PN X Places -> {True, False}
  – If Marked (PN, place\_i) = True we say that place\_i is *marked*
• A transition, t\_i ∈ Transitions (PN) can *fire* if
  for all of its inedges, (place\_i, t\_i) Marked (place\_i) = True
  – After a transition t\_i ∈ Transitions (PN) fires
    » Marked (p\_i) ← False for all places, p\_i, such that (p\_i, t\_i) ∈ Edges (PN)
    » Marked (place\_k) ← True, for all place\_k, such that
      (t\_i, place\_k) ∈ Edges (PN)
Many Extensions

• Bi-Logic Nets
  – Allows Oring inputs and outputs
• Colored Petri Nets
  – Tokens can now be “typed”
• Hierarchical Petri Nets
  – Usual advantages of hierarchical decomposition
• Queues at places
• Timed Petri Nets
  – For estimating performance
• And various combinations of these

Evaluation of Petri Nets

What are Petri Net representations good for?:
  • How do things get done?
  • Especially parallelism and nondeterminism
  • Helps spot races and deadlocks

What kinds of things are Petri Nets not good for?
  • Too little focus on product (tokens represent it)
  • Don’t scale very well
    Petri Nets get large and complicated fast
  • Many extensions address shortcomings
  • Too many extensions confuse the picture?