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 Graphs

• Semantics are different from DFG'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

- Semantics are different from DFG’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 $OPS = \{ops\}$ nodes where computation is done
  $TEST = \{test\}$ nodes where tests are done
- Edges: $ImmFol$ 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 $fe_i$
- $I \{(fe_i, fe_k)\}$ where $fe_i \in TEST = 2$
- $I \{(fe_i, fe_k)\}$ where $fe_i \in OPS = 1$
- If so, then OPS and TEST nodes don’t need to look different (!)
- $FG$ is a directed, connected graph
- Maybe(?): $FN = OPS \cup TEST \cup INPUT \cup OUTPUT$ where
  - $fn_i \in OUTPUT \Rightarrow \nexists (fn_i, fn_k) \in FE$
  - $fn_i \in INPUT \Rightarrow \nexists (fn_k, fn_i) \in FE$
- Other semantics?
  - What does it mean for a node to have $>1$ inedge?
  - What about data flow?

Control Flow Graph

```
Read Floor, Direction
BB_Read: curr-length, Floors(.), Directions(.)
Direction = Up
  incr <-- 1
  incr <-- -1
  NFloor <-- Floor; increment NFloor;
  Directions[I] = incr ?
    yes
    no
    yes
    no
    yes
    no
  NFloor=Floors(I)
    for any 1<= I <= curr-length
    Directions[I] = incr ?
      yes
      no
      yes
      no
      yes
      no
other car is nearer ?
  output new floor
```

Another possible control flow graph logic could be:

```
BB_Read: curr-length, Floors(.), Directions(.)
Direction = Up
  incr <-- 1
  incr <-- -1
  NFloor <-- Floor; increment NFloor;
  Directions[I] = incr ?
    yes
    no
    yes
    no
    yes
    no
  NFloor=Floors(I)
    for any 1<= I <= curr-length
    Directions[I] = incr ?
      yes
      no
      yes
      no
      yes
      no
other car is nearer ?
  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

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 $(C, S) \in E$
  - “and” semantics
- 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;

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

Directions[i] = incr ?

other car is nearer?

output new floor

NFloor out of bounds

increment NFloor

complement Direction
Control Flow Graph

Read Floor, Direction

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

Direction = Up

 incr <-- 1

 incr <-- -1

yes

no

NFloor <-- Floor;

Directions[NFloor] = incr ?

yes

other car is nearer ?

no

output new floor

increment NFloor

Complement Direction (Incr <-- -Incr)

yes

no

NFloor out of bounds

Control Flow Graph with Data Flow

Read Floor, Direction

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

Direction = Up

 incr <-- 1

 incr <-- -1

yes

no

NFloor <-- Floor;

Directions[NFloor] = incr ?

yes

other car is nearer ?

no

output new floor

increment NFloor

Complement Direction (Incr <-- -Incr)
Control Flow Graph with Data Flow

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, \( \partial \)), 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
  - \( \partial \) is a function, \( \partial: Q \times I \rightarrow Q \) such that if the system is in state \( q \) and event \( i \) occurs, then the system transitions to state \( \partial(q, i) \)

- 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 for Digital Watch

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
Statecharts: More Complex FSMs

States

- Initialize
  - do: Initialize course object
- Unassigned
  - do: Assign professor to course
- Canceled
  - do: Send cancellation notices
- Closed
  - do: Report course is full
- RegistrationComplete
  - do: Generate class roster

addStudent/numStudents = 0

Open

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

addStudent

Unassigned

- do: Assign professor to course

Canceled

- do: Send cancellation notices

Closed

- do: Report course is full

Canceled

- do: Send cancellation notices

Unassigned

- do: Assign professor to course

CancelCourse

Activity to be performed in this state

RegistrationComplete

- do: Generate class roster

CS 520/620 Spring 2014 Univ. of Massachusetts

Copyright L. Osterweil, all rights reserved
Statecharts: More Complex FSMs

- Initialize
  - do: Initialize course object

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

- Open
  - entry: Register a student
  - cancelCourse
  - registration closed
    - numStudents < 3
    - numStudents >= 3

- Closed
  - numStudents = 10
  - registration closed
    - numStudents < 3
    - numStudents >= 3
  - do: Report course is full
  - RegistrationComplete
    - do: Generate class roster

- Canceled
  - do: Send cancellation notices

- Unassigned
  - activity on this transition

- addStudent
  - condition on this transition
Statechart with Nested States

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
Scenario Definition with Petri Nets

• A marking represents a scenario
• Applying firing rules creates a simulation
• Different markings support exploring dynamic behaviors of a system

More Details in
Ghezzi, et al., Chapter 5.5.3
Unmarked Petri Net

A Scenario: Car going up with UP button pressed
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
Unmarked Petri Net--Two Elevator Cars

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

Car 1 approaching
Floor n, going up

Up Button
Pressed at
Floor n

Car 2 approaching
Floor n, going up

No Press

Stop at
Floor

Go to
Next
Floor
Up

Car 1 approaching
Floor n, going up

Up Button
Pressed at
Floor n

Car 2 approaching
Floor n, going up

No Press

Stop at
Floor

Go to
Next
Floor
Up

Up Button
Pressed at
Floor n

Car 1 approaching
Floor n, going up

No Press

Stop at
Floor

Go to
Next
Floor
Up

Up Button
Pressed at
Floor n

Car 2 approaching
Floor n, going up

No Press

Stop at
Floor

Go to
Next
Floor
Up
Petri Net--Two Elevator Cars
Car 1 Wins Race

Car 1 approaching Floor n, going up

Up Button Pressed at Floor n

Car 2 approaching Floor n, going up

No Press

Go to Next Floor

Stop at Floor

Stop at Floor

Go to Next Floor

Up Button Pressed at Floor n

Car 2 approaching Floor n, going up

No Press

Go to Next Floor

Stop at Floor

Stop at Floor

Go to Next Floor

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

Car 1 approaching Floor n, going up

Car 2 approaching Floor n, going up

Up Button Pressed at Floor n

No Press

Stop at Floor

Go to Next Floor

Up

Car 1 approaching Floor n, going up

No Press

Stop at Floor

Go to Next Floor

Up Button Pressed at Floor n

No Press

Stop at Floor

Go to Next Floor

Up

Car 2 approaching Floor n, going up

Car 2 approaching Floor n, going up

Go to Next Floor

Up

Car 1 approaching Floor n, going up

No Press

Stop at Floor

Go to Next Floor

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

Petri Net for a Different Elevator


Marking for moving up to pick up a passenger

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

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

CS 520/620 Spring 2014 Univ. of Massachusetts
Copyright L. Osterweil, all rights reserved
Marking for moving up to pick up a passenger

Queues at places?

Car going up

Up button higher

Move up to Nearest Up Button

Car going down

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

Move up to Nearest Up Button

Car going down

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

Car going down
- Down button lower
  - No Down Button Lower
  - Move to nearest Down Button
    - Visit floor, Turn off button

Move to nearest Down Button
- Visit floor, Turn off button
Marking for both up and down buttons

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

Marking for neither up nor down buttons

- Car going up
  - Up button higher
  - No Up Button Higher
  - Move to Nearest Up Button
    - Visit floor
  - Car going down
    - No 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_j, place_i)\},
    - 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_j, t_i) Marked (place_j) = True
- After a transition t_i ∈ Transitions (PN) fires
  - Marked (p_j) <- False for all places, p_j such that (p_j, 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?