Computer Science 520/620  
Spring 2013  
Prof. L. Osterweil

Software Models and Representations  
Part 2  
Data Flow Graphs

Being  
Clear  
Precise  
Comprehensive  
Detailed  
About Software
Approaches that are Problematic

- Natural Language
- Disciplined Natural Language
- Code

They have their advantages, but serious disadvantages too

Using Different Models to Represent Different Aspects of Software

- Model represents certain specific features
- Others are suppressed, not represented
- Aim is to clarify some issues by not having to worry about the others
- But this leads to different representations of reality
- How to see/infer the whole picture?
Plato’s Allegory of the Cave

The question of how to synthesize multiple views of “reality” is at least 2000 years old.

What is most tangible is least “real”
What is least tangible is most “real”
Plato could have been (was?) a great software engineer.
Graphs as Visualization Aids

- Graphs are mathematical structures with obvious visualizations that seem often to help many stakeholder communities to visualize key relations.
- A graph’s edges visually represent the ordered pairs that compose the relation.
- If the pairs in $E$ are ordered, then $G$ is a directed graph, and its edges are depicted with arrowheads.
  - If not, the graph is called an undirected graph.

Example: Elevator Controller

- What is it supposed to do?
  - Stop on every floor it is called to
    » Maybe not so easy: multiple elevators
  - Service users “first come, first served”
    » May conflict with optimal strategies
- These are hard to be precise about, reason about in “plain English”
- Code helps with some of these, but not all
  - What should it never do?
    - Allow elevator to move with doors open
- Graphs are frequently used to specify this kind of thing
  - Many kinds of graphs for many purposes
  - A graph is a picture of a relation
Data Flow Graph

- Current Floor
- Select New Floor
- New Floor
- Move Elevator
- New Floor
- Get copy of Request List
- Request List
- Update Request List
- Turn on Light
- Turn off Light
- Delete Request Just Satisfied
- New Request

Better Edge Annotations

- Current Floor
- Select New Floor
- Floor
- Move Elevator
- New Floor
- View of Request List
- Request List
- Request
- Update Request List
- Turn on Light
- Turn off Light
- Request Deletion Packet
Control Flow Graph

**Read Floor, Direction**

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

**Direction = Up**

incr <-- 1

** 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

Finite State Machine

**Doors Closed, Elevator Idle**

Request to Move to new floor

**Doors closed Elevator in Motion**

Request to Move to new floor

Arrival at New Floor

Queue empty

**Doors open, Elevator Stopped**

No new request timeout

**Get New Floor request**

Queueing new floor request

**Arrival at New Floor**

Request to Move to new floor

**Doors open Elevator in Motion**

Request to Move to new floor

**Doors Closed, Elevator Idle**

Request to Move to new floor

**Request to Move to new floor**

**Request to Move to new floor**

**Request to Move to new floor**

**Request to Move to new floor**
**Petri Net**

![Petri Net Diagram](image)

**Pictures are not enough**

- Want to be able to reason about them
- Answer (stakeholder) questions
  - Can this data item ever reach that statement?
  - Can these two events ever happen in sequence?
  - What is the maximum time to execute that sequence?
- Pictures can leave ambiguous impressions
  - How to be sure what they say?
- Graphs, representing mathematical relations, can support deriving definitive answers
  - About the relation
What is This Graph Specifying?

Annotations Provide Intuitions

But are they suggesting too much?
What’s wrong with this diagram

There is ambiguity and misuse of notation here:
• one circle is a test, others are functions
• some edge annotations are data, some predicates
• are multiple arrows in and out “and” or “or”?

Back to Basics

• Review fundamental Finite Mathematics
  – Set theory
  – Graph theory
  – Predicate Calculus
  – Etc.
Relations:
A RELATION, R, over a set, \( S = \{s_i\} \) is a set of tuples

\[ R = \{r_i\}, \text{ where } r_i = (s_{i,1}, s_{i,2}, \ldots, s_{i,n}) \]

An n-ary relation is a relation where all of the tuples are n-tuples

A Binary relation is a relation where all the tuples are 2-tuples

If \((s_i, s_j)\) is an element of \( R \), then we often write \( s_i R s_j \)

Another view of relations:

The relation, \( R \), over the set \( S \) can be defined as:

\[ R = \{ (s_i, \ldots, s_j) \mid \text{PRED}(s_i, \ldots, s_j) = \text{True}, \text{ for some predicate, PRED} \} \]

If the tuples are ordered, the relation is called an ordered relation

If the tuples, \( <t_{i,1}, t_{i,2}, \ldots, t_{i,n}> \) are unordered, the relation is an unordered relation

Some Examples

Let \( I = \{\text{all integers}\} \),
Define \( Q = \{ (x,y,z) \mid x, y, z \text{ are integers and} \)

\[ y = x**2, \text{ } z = x**3 \} \]

Let \( S = \{\text{all states of the U.S.}, S_i\}, \)
Define \( B = \{ (S_i, S_j) \mid S_i \text{ and } S_j \text{ are states that share a border} \} \)

Let \( L = \{\text{all statements } L_i \text{ in a program, } P\}, \)
Define \( \text{ImmFol} = \{ (L_i, L_j) \mid \text{the execution of } L_i \text{ may immediately follow the} \)

\[ \text{execution of } L_i \text{ for some execution of } P \} \]
Some Properties of Relations

Some familiar properties of ordered binary relations, $R$, over the set $S=\{s_i\}$:
- **Symmetry**: $s_i R s_j \Rightarrow s_j R s_i$ for all pairs, $s_i$ and $s_j$ in $S$
- **Reflexivity**: $s R s$, for all $s$ in $S$
- **Transitivity**: $s_i R s_j$ and $s_j R s_k \Rightarrow s_i R s_k$, for all $s_i$, $s_j$ and $s_k$ in $S$

A relation that is symmetric, reflexive and transitive is called an **equivalence relation**

If $R = \{(s_i, s_j)\}$ is transitive, then $C = \{(s_a, s_b) | \text{ there exists a sequence, } i_1, i_2, \ldots, i_n, \text{ such that } s_a = s_{i_1} R s_{i_2}, s_{i_2} R s_{i_3}, \ldots, s_{i_{n-1}} R s_{i_n} = s_b \}$ is called the transitive closure of $R$

**Antisymmetry**: $s_i R s_j \Rightarrow \neg(s_j R s_i)$ for all pairs, $s_i$ and $s_j$ in $S$

**Irreflexivity**: $s \sim R s$ for all $s$ in $S$

Examples

If $S=\{\text{all subroutines written in Fortran}\}$, $s_1 R s_2$ if and only if $s_1$ calls $s_2$, then $R$ is an irreflexive relation

Let $PS = \{c_s, \text{ all the statements in a program that consists of a set of modules, } M=\{m_t\}\}$,
- $INMOD = \{(c_e, c_f) | c_e \text{ and } c_f \text{ appear in the same module } m_t\}$
- $INMOD$ is an equivalence relation

The relation $\text{ImmFol}$ (earlier slide) is not transitive

Change $\text{ImmFol}$ to $\text{Fol}$, by defining $\text{Fol} = \{(L1, L2) | \text{ the execution of } L2 \text{ may follow the execution of } L1 \text{ for some execution of } P\}$
Examples

If S={all subroutines written in Fortran} s₁ \ R \ s₂ if and only if s₁ calls s₂, then R is an irreflexive relation

Let PS =\{c, all the statements in a program that consists of a set of modules, M={m} \},
INMOD = \{ (c₁, c₂) | c₁ and c₂ appear in the same module m \}
INMOD is an equivalence relation

The relation ImmFol (earlier slide) is not transitive

Change ImmFol to Fol, by defining Fol = \{ (L₁, L₂) | the execution of L₂ may follow the execution of L₁ for some execution of P \}
Fol is still not transitive

Graphs

A **Graph**, G = (N, E) is an ordered pair, consisting of a node set, N, and an edge set, E = \{(nᵢ, nⱼ) \}

If OG=(N, E) is an ordered graph with E=\{(nᵢ, nⱼ)\} then its unordered version, UG=(N, U), where U=\{<nᵢ, nⱼ>\}
Graphs as Visualization Aids

- Graphs are mathematical structures with obvious visualizations that seem often to help many stakeholder communities to visualize key relations.
- A graph’s edges visually represent the ordered pairs that compose the relation.
- If the pairs in E are ordered, then G is a directed graph, and its edges are depicted with arrowheads.
  - If not, the graph is called an undirected graph.

Paths

A *path*, P, through an ordered graph G=(N,E) is a sequence of edges, \((n_{1,1}, n_{1,2}), (n_{2,1}, n_{2,2}), \ldots, (n_{t,1}, n_{t,2})\) such that \(n_{j,k-1} = n_{i,k}\) for all \(2 \leq k \leq n\).

A path, UP, thru an unordered graph UG=(N,U) is a sequence of edges, \(<n_{1,1}, n_{1,2}>, <n_{2,1}, n_{2,2}>, \ldots,<n_{t,1}, n_{t,2}>\) such that all of the \(<n_{i,k}, n_{i,k}>\) can be ordered to assure that \(n_{i,k-1} = n_{i,k}\) for all \(2 \leq k \leq n\).

In either case, \(n_{1,1}\) is called the start node and \(n_{t,2}\) is called the end node.

The *length* of a path is the number of edges in the path.

A graph G is *connected* if and only if, for every pair of nodes, \(n_1, n_2\), there is path from one of them to the other with G considered to be an unordered graph.

These graph constructs appeal visually to many stakeholders and often effectively support answering their questions.
Trees

A **cycle** in a graph $G$ is a path whose start node and end node are the same.

A **simple cycle** in a graph $G$ is a cycle such that all of its nodes are different (except for the start and end nodes).

If a graph $G$ is **connected** and has no path through it that is a cycle, then the graph is called acyclic.

An acyclic, connected, unordered graph is called a **tree**.

A collection of trees is called a **forest**.

If the unordered version of an ordered connected graph is acyclic, the graph is called a **directed tree** (in this course we drop this distinction and call both graphs trees).

If the unordered version of an ordered graph has cycles, but the ordered graph itself has no cycles, then the graph is called a **Directed Acyclic Graph (DAG)**.

Other Types of Graphs

A **Multigraph** $MG$ is an ordered pair $MG = (N, C)$ where $N$ is a set of nodes $\{n_i\}$ and $C$ is a collection of pairs of nodes (edges) with repetitions allowed (i.e. $C$ can be a multiset).

A **Hypergraph** $HG$ is an ordered pair $HG = (N, T)$ where $N$ is a set of nodes $\{n_i\}$ and $T$ is a set of $t$-tuples of nodes, where $t > 2$.

A **Hypermultigraph** is a hypergraph where the set of $t$-tuples can be a multiset.

A **bipartite** graph $BG$ is an ordered pair, $BG = \{BN, E\}$ where $BN$ is a node set that is the union of two disjoint subsets, $N_1 \cup N_2$, and no edge in $E$ has both nodes in either $N_1$ or $N_2$.

A **bipartite graph** is often called a 2-colorable graph.

A **$k$-colorable** graph is defined analogously, with $BN$ being the disjoint union of $k$ subsets.
DATA FLOW Graph (Again)

- Capture system functionality: What does the system do? How?
- Basic components of a data flow diagram:
  - Nodes, represented by circles (boxes), are functional units
  - Edges, represented by arrows, are data flows between units
  - Both augmented by separate annotation relations
  - Boxes (sometimes circles), represent I/O data

EG.

```
input
width
height
check arg. values
args OK
compute area
area
print area

input width
width
output error msg.
args bad
output error msg.

args
OK
area

OK
area
area
error msg.
print error msg
```

Differences in Graphs Result from Different Relations

- Data Flow:
  - Nodes represent the set of sites where data is generated/used
  - Each edge is a (data generated, data used) node pair
- Control Flow:
  - Nodes represent units of functionality
  - \((n_1, n_2)\) is an edge in this graph if and only if unit \(n_2\) can execute immediately after \(n_1\) executes (ImmFol relation)
- Hierarchy:
  - Models "consists of" or "is a part of"
  - Key to divide-and-conquer approaches to understanding
- Finite State Machines
  - Nodes represent all possible different "execution states"
  - \((s_1, s_2)\) is an edge if and only if it is possible for state \(s_1\) to immediately succeed \(s_2\). Called a transition from \(s_1\) to \(s_2\)
  - Edges annotated with transition condition
- Petri Nets
  - Multiple node and edge types in the same diagram
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 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 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?
- How does a node do its job?

Flowgraphs

Let \(S = \{\text{all statements } s_i \text{ in a program, } P\}\)

Let \(\text{ImmFol} = \{ (s_i, s_j) \mid \text{The execution of } s_j \text{ immediately follows the execution of } s_i \text{ for some execution of } P \}\)

Then: If \(FG = (S, \text{ImmFol})\), \(FG\) is called the flowgraph of \(P\)

\(FG\) is an ordered graph

Every execution sequence (ie. the sequence in which the statements of \(P\) are executed for a given execution of \(P\)) corresponds to a path in \(FG\).

However---the converse is not true. A path through \(FG\) may not correspond to an execution sequence for \(P\)

A loop in \(P\) appears as a cycle in \(FG\)
Callgraphs

Let PROC = \{ procedures $S_i$ that the program P comprises \}

Let CALLS = \{ ($S_i$, $S_j$) | $S_j$ is directly invoked from $S_i$ during some execution of P \}

Then CG = (PROC, CALLS) is called the Call Graph of P

CG is a directed graph

If P is written in a language that does not allow recursion, then CG will be acyclic

A cycle in CG indicates that the nodes along the cycle participate in a recursive calling chain

NOTE: DEPICTIONS OF THESE GRAPHS MAY BE SUPERIMPOSED OVER EACH OTHER TO CLARIFY (?!)

THINGS

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

Plato’s Allegory of the Cave

The question of how to synthesize multiple views of “reality” is at least 2000 years old
Plato’s Allegory of the Cave

What is most tangible is least “real”
What is least tangible is most “real”
Plato could have been (was?) a great software engineer

Simple Hierarchical Elaboration

Check args.
- height
- width
- args OK
- args not OK

Check args.
- height
- width
- valid pair
- height > width
- width > 0
- height > 0
- args OK
- args not OK
Something a Little More Complex

Hierarchical Elaboration (of “Select New Floor”)
Hierarchical Elaboration (of “Select New Floor”)

Current Floor

Find Nearest Floor

Floor

New Floor

Consider Other Cars

Floor

Direction

Floors in the right direction

Floors in the right direction

Consider Other Cars

New Status

Do Updates

Floor to Move to

Request Deletion Packet

View of Request List

New Status

Request List

Is this consistent with its parent?

Inconsistencies with parent
Hierarchical Elaboration (of “Select New Floor”)

(Some) Consistency definitions

- Let \((n, \text{somenode}) \in \text{NodeAnnotation}_{\text{ParentGraph}}\)
  for some \(n\), \(\text{somenode} \in N\), where ParentGraph = \((N, E)\)
- Let \(\text{somenode} = (N', E')\)
  - This is the DFG elaborating on “\text{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 \(\text{Cardinality} (\text{OUTPUT(\text{somenode})}) = k\)
Back to Previous Example

Elaboration of “Select New Floor” Suggests Artifact Issues
Data needs precise specification too

- DFG's focus on functionality, using data as a vehicle
- Data shown as unstructured atomic units—usually unrealistic
- Complex functions cannot be adequately defined without delving into the details of how they handle structured data
- Sub-DFG's can show how the high level data that high level DFG's deal with is decomposed
  -- But this is implicit data definition
  -- Can be hard to read/inconsistent
- Data specification is worth doing explicitly, carefully
- Usually using Disciplined Natural Language—eg. Templates
- Formal specification of data is an important future topic

IDEF0

- Commercial DFG formalism
- Some formality and rigor behind it
- Primarily pictorial
- In wide use
- Additional semantics in IDEF1, IDEF2, etc.
Note: The edges here do not comprise a set. They comprise a collection.
Broadening DFG Semantics

- Node cannot begin until data arrives along all in-edges
  - Let $DFG = (N, E)$, where $N$ is set of nodes, $\{n_i\}$. If $n_i$ is a function, then it must be defined on the set of all artifacts $IA_i = \{ia_{j,i}\}$, such that $(n_j, n_i, ia_{j,i}) \in EdgeAnnotaiton_{DFG}$
  - 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
- 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

This composite actor is meant to be used in the PN domain to wrap around an actor with one input and one output.

This actor raises a subworkflow if functionPass is false up to at most count number of times. It outputs the last functionOutput and functionPass.
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

Given that Precision is essential

- What about the other three dimensions?
- Detail
  - Gained from hierarchical elaboration
- Breadth
  - Comes from different (sub)types of DFG
- Clarity
  - Seems to be reduced by increased Detail and 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