Computer Science 520/620
Spring 2014
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.

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.
Graphs as Visualization Aids

- A graph is a picture of a relation
- Graphs are mathematical structures with obvious visualizations that seem often to help many stakeholders visualize these 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**

1. Current Floor
2. Select New Floor
3. View of Request List
4. Request List
5. Update Request List
7. Move Elevator
8. Request Deletion Packet
9. Turn on Light
10. New Floor

**Control Flow Graph**

1. Read Floor, Direction
2. DB_Read: curr-length, Floors( ), Directions( )
3. Direction = Up
4. incr <-- 1
5. yes
6. no
7. NFloor <-- Floor; increment NFloor;
8. incr <-- -1
9. yes
10. no
11. NFloor=Floors(I) for any 1<= I <= curr-length
12. yes
13. no
14. Directions[I] = incr ?
15. yes
16. no
17. other car is nearer ?
18. yes
19. no
20. output new floor
21. Nfloor out of bounds
22. yes
23. no
24. complement Direction
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

Queueing new floor request

Getting new floor request

Request to Move to new floor

Request to Move to new floor

Arrival at New Floor

Doors open,
Elevator Stopped

No new request timeout

Doors open
Elevator in Motion

Petri Net

Car going up

Up button Pressed

Stop at Floor

No Press

Down Button Pressed

Go to Next Floor Up
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?

And what can we safely infer/conclude from it?
Annotations Provide Intuitions

But are they suggesting too much?

What’s wrong with this diagram
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.
- But with an emphasis on why this is of interest in software engineering
Relations:
A RELATION, R, over a set, S = \{s_i\} is a set of tuples
\[ R = \{r_i\}, \text{ where } r_i = (s_{i,1}, s_{i,2}, ..., s_{i,n}) \]
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 \(s_i \mathrel R s_j\)

Another view of relations:

The relation, R, over the set S can be defined as:
\[ R = \{(s_i, ..., s_j) \mid \text{PRED}(s_i, ..., s_j) = \text{True}, \text{for some predicate, PRED}\} \]

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

Some Examples

Let \(I = \{\text{all integers}\}\),
Define \(Q = \{(x, y, z) \mid x, y, z \text{ are integers and} \ y = x^{**2}, z = x^{**3}\}\)

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 \(\text{ImmFol} = \{(L_i, L_j) \mid \text{the execution of } L_j \text{ may immediately follow the execution of } L_i \text{ for some execution of } P\}\)
Some Properties of Relations

Some familiar properties of ordered binary relations, \( R \), over the set \( S=\{s_i\} \):

- **Symmetry:** \( s_i \, R \, s_j \implies s_j \, R \, s_i \) 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 \( s_j \, R \, s_k \implies s_i \, R \, s_k \), for all \( s_i, s_j, s_k \) in \( S \)

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

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

- **Antisymmetry:** \( s_i \, R \, s_j \implies \neg(s_j \, R \, s_i) \) 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 \( PS =\{c_s, \text{ all the statements in a program that consists of a set of modules, } M=\{m_i \} \}, \)
\( \text{INMOD} = \{(c_{e_s}, c_{f_s}) \mid c_{e_s} \text{ and } c_{f_s} \text{ appear in the same module } m_i \} \)
\( \text{INMOD} \) is an equivalence relation.

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

Change \( \text{ImmFol} \) to \( \text{Fol} \), by defining \( \text{Fol} = \{(L_1, L_2) \mid \text{the execution of } L_2 \text{ may follow the execution of } L_1 \text{ for some execution of } P\} \)
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 \( PS = \{ c_e, \text{all the statements in a program that consists of} \} \), a set of modules, \( M=\{m_i\} \),
\( \text{INMOD} = \{ (c_e, c_f) \mid c_e \text{ and } c_f \text{ appear in the same module } m_i \} \)
\( \text{INMOD} \) is an equivalence relation.

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

Change \( \text{ImmFol} \) to \( \text{Fol} \), by defining \( \text{Fol} = \{ (L_1, L_2) \mid \text{the execution of } L_2 \text{ may follow the execution of } L_1 \text{ for some execution of } P \} \)
\( \text{Fol} \) is still not transitive.

Graphs

A Graph, \( G = (N, E) \) is an ordered pair, consisting of a node set, \( N \), and an edge set, \( E = \{(n_i, n_j)\} \).

If \( OG=(N, E) \) is an ordered graph with \( E=\{(n_i, n_j)\} \) then its unordered version, \( UG=(N, U) \), where \( U=\{<n_i, n_j>\} \)
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_{i,1}, n_{j,1}), (n_{i,2}, n_{j,2}), ..., (n_{i,t}, n_{j,t}) ) such that n_{j,k-1} = n_{i,k} for all 2<= k <= n.

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

In either case, n_{i,1} is called the start node and n_{j,t} 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 a 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 \( \{n\} \) 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 birpartite 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 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

<<This is actually yet another relation

EG.

\[
\begin{array}{ccc}
\text{input height} & \text{input width} & \text{check arg. values} \\
\text{compute area} & \text{output error msg.} & \text{print area} \\
\text{print error msg.} & \text{arg. OK} & \text{arg. bad}
\end{array}
\]

Differences in Graphs Result from Different Relations

- Data Flow:
  -- Nodes represent 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 \(s_2\) to immediately succeed \(s_1\).
  -- 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 \(\text{operand}\) 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?

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

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

Then $\text{CG} = (\text{PROC}, \text{CALLS})$ is called the Call Graph of $P$

$\text{CG}$ is a directed graph

If $P$ is written in a language that does not allow recursion, then $\text{CG}$ will be acyclic

A cycle in $\text{CG}$ 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

height > width

valid pair

height > 0

width > 0

no

no

args OK

args not OK
Something a Little More Complex

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

Current Floor

Floor to Move to

Find Nearest Floor

New Floor

Consider Other Cars

Floor Rejection

Request Deletion Packet

Do Updates

New Status

Request List

View of Request List

Process Request List

Floors in the right direction

Direction

Floor

Is this consistent with its parent?

Hierarchical Elaboration (of “Select New Floor”)

Current Floor

Floor to Move to

Find Nearest Floor

New Floor

Consider Other Cars

Floor Rejection

Request Deletion Packet

Do Updates

New Status

Request List

View of Request List

Process Request List

Floors in the right direction

Direction

Floor

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

Current Floor
→ Find Nearest Floor
→ Direction
→ Floors in the right direction
→ Consider Other Cars
→ Do Updates
→ Floor to Move to

Request List

Some Example of “real world” DFGs

- IDEF
- Kepler
- Scientific Workflows
A Kepler Example
Scientific Workflow Graph

```
Raw Data
  ↓
Create Aggregated Data
  ↓
Aggregated Data
  ↓
Create Interpolation Model
  ↓
Interpolation Model
  ↓
Apply Interpolation Model
```

The need for this mathematical precision

- Appeals to some stakeholders
  - Verifiers, regulators, innocent bystanders (e.g. people living near nuclear reactors)
- Supports tools that can verify these consistency relations automatically
- Reduces the need to trust fallible humans
Assume, pictorially….

This is a part of $G$

Input1

somenode

Input2

Output1

Output2

Saying this Rigorously

• Let $(n, \text{somenode}) \in \text{NodeAnnotation}_G$ for some $n, \text{somenode} \in N$, where $G = (N, E)$ is the “parent graph”
• Let $\text{somenode} = (N', E')$
  – This is the DFG elaborating on $\text{somenode} \in N$

THEN
Saying this Rigorously

• Let \((n, \text{somenode}) \in \text{NodeAnnotation}_G\) for some \(n, \text{somenode} \in N\), where \(G = (N, E)\) is the “parent graph”
• Let \(\text{somenode} = (N', E')\)
  – This is the DFG elaborating on \(\text{somenode} \in N\)
• Two easy consistency properties
  – If \(\exists m_i\) such that \((m_i, \text{somenode}) \in E\), then \(\text{Cardinality}(\text{INPUT}_{\text{somenode}}) \neq 0\)
  – If \(\exists m_i\) such that \((\text{somenode}, m_i) \in E\), then \(\text{Cardinality}(\text{OUTPUT}_{\text{somenode}}) \neq 0\)

More specifically
(Some) Consistency definitions

- Let \((n, \text{somenode}) \in \text{NodeAnnotation}_G\) for some \(n, \text{somenode} \in N\), where \(G = (N, E)\)
- Let \(\text{somenode} = (N', E')\)
  - This is the DFG elaborating on \(\text{somenode} \in N\)
- More specifically
  - Let \(\text{INS}_{\text{somenode}, G}\) be the subset of \(E\), \((*, \text{somenode})\)
    for which \(\text{somenode}\) is the second component.
    Then \(|\text{INS}_{\text{somenode}, G}| = |\text{INPUT}_{\text{somenode}}|\)
  - And similarly:
    - Let \(\text{OUTS}_{\text{somenode}, G}\) be the subset of \(E\), \((\text{somenode}, *)\)
      for which \(\text{somenode}\) is the first component.
      Then \(|\text{OUTS}_{\text{somenode}, G}| = |\text{OUTPUT}_{\text{somenode}}|\)

More Semantics

- The annotation on each input to \(\text{somenode} \in N\), the node set of \(G\), must match the annotation on some edge, \(e' \in E'\)
  - \(\text{INPUT}_{\text{somenode}} \subseteq E'\), the edge set of \(\text{somenode} = (N', E')\)
  - MORE PRECISELY
- Suppose \(e = (n, \text{somenode}) \in E\), and
  \((e, \text{name}) \in \text{EdgeAnnotation}_G\)
  Then \(\exists e' = (\text{in'}, n_2') \in \text{somenode}\), where \(\text{in'} \in \text{INPUT}_{\text{somenode}}\)
    for which \((e', \text{name}) \in \text{EdgeAnnotation}_{\text{somenode}}\)
- And conversely
- Similarly for OUTPUT
AND vs OR Pictorially

Specifying AND vs OR

- Node cannot begin until data arrives along all in-edges
  - Let \( G = (N, E) \), then \( \forall n \in N, \text{ let } \text{INS}_n \)

\[
\text{INS}_n = \{ e \in E \mid e = (n', n), \text{ for some } n' \in N \}
\]

and let \( \text{INANNS}_n \)

\[
\text{INANNS}_n = \{ \text{ann} \mid (e, \text{ann}) \in \text{EdgeAnnotation}_G \text{ for some } e \in \text{INS}_n \}
\]

Then \( n \) represents a function,

\( f (i_1, i_2, \ldots, i_t), \text{ where } \{i_j\} = \text{INANNS}_n \)

Need something similar to specify DFG for which ANY Input is sufficient to trigger \( n \)

And something similar for AND and OR outputs

These sorts of constraints can support additional types of reasoning:

Eg. about parallelism
Why the need for this mathematical precision?

- Appeal to some stakeholders
  - Verifiers, regulators, innocent bystanders (e.g. people living near nuclear reactors)
- Supports tools that can verify these consistency relations automatically
  - Reduce the need to trust fallible humans

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
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?
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.
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 takes 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