Software Models and Representations: Introduction

Key Focus of Software Engineering

- How to describe software products?
- Processes to build software products?
  - And assure that the products are "good" at the end
- How to grow and evolve such products
  - At acceptable costs
  - And improving quality

Software Engineering = Products + Processes

Stakeholder Perspective Suggests an Approach

- How to manage the creation and maintenance of a software product that satisfies all needs of all stakeholders?
  - Implies understanding who stakeholders are; what questions they need answered; to what degree of thoroughness
  - Implies supporting reasoning needed to provide adequate answers to stakeholder questions (at acceptable cost)
  - Implies need for representation(s) of product sufficient to support such reasoning; and reasoning techniques

Stakeholder questions and concerns

The Problem of Providing Satisfactory Answers to Stakeholder Questions

- Most important questions are vague
  - Eg. Is this software user-friendly?
  - (ATCS) How to interact with pilots to assure user interfaces are going to be effective?
- Most important questions have open-ended answers
  - Eg. How fast is this system?
  - (ATCS) Different platforms, situations, conditions
- Different stakeholders require different degrees of assurance
  - Affects the degree of thoroughness of (eg.) testing
  - (ATCS) Some functions more critical than others
- Different stakeholders have differing degrees of technical sophistication
  - Affects the choice of formalism in which to couch answers

Using Models to Answer Questions

- What is a model?
- How have they been used elsewhere?
  - Most previous models have been tangible and have typically been models of physical objects
- Software Models
  - Model non-physical objects, are not tangible
  - What should they be like?
  - Should be driven by what they are for
No one representation can be expected to suffice for all purposes
- That is the case in manufacturing other kinds of products:
  - Houses: Multiple diagrams and views; appliance user manuals; inspection reports and certificates, etc.
  - Cars: Owners manuals; shop manuals; EPA reports; NTSB reports;
  - Laws: Legislative blueprints; law; opinions; decisions in cases involving the law; impact assessments; agency budgeting reports
  - Movies: Story treatment; script; actual film; reviews; profit and loss accountings
- Representations are quite varied: Notations expressing them are too
- We should expect the same for computer software–products and processes
- Notations are better evolved, more effective, in cases of better established manufacturing disciplines

Typical Approach
- Select a (set of [interconnected?]?) representation(s?) (some of which are?) effective in communication with stakeholder constituencies?
- Derive information/answers expressed in that representation that are able to satisfy stakeholder(s)
- Assure that the information is consistent with other parts of the product (eg. the code!)
- ATCS: FAA needs proof that all collisions will be detected
  - Statement of proof needed
  - Other artifacts: body of proof, code, code structure representations, etc. Must be derived and shown to be consistent with each other

Key Model Desiderata
- Precise
- Detailed
- Broad
- Clear

Precise
- Based upon rigorously defined semantics
- So that there is a meaning that can be definitively established to the satisfaction of all
  - To resolve disputes and disagreements
- So that definitive reasoning is possible
  - Deriving properties
  - Answering specific stakeholder questions

Detailed
- Stakeholder questions may need answers that require details
- Model must support getting down to the level of detail that is required
  - By the relevant stakeholder(s)

Broad
- Many semantic issues may need to be covered
  - Functionality, timing, robustness, precision, resource utilization....
- More semantic richness is better
  - It is capable of supporting answering more questions from more stakeholder types
- Broad and Precise and NOT the same
Clear

- Understandable to the relevant stakeholder(s)
- So that they can understand what the model is saying
  - And what the results of reasoning mean
- This is definitely relative to the stakeholder community

The Great Compromise

- No single modeling approach is likely to excel in all four dimensions
- What is each one likely to be good for?
- Different needs under different circumstances
  - During conceptualization, contract negotiation
  - During development
  - During test and evaluation
  - During utilization
  - During evolution
- How to select one(s) that are appropriate for the stakeholder communities?
  - At the relevant time(s)

One of the most important tasks for a software engineer

Some Software Representation Approaches

- Programming Languages
- Natural Language
- Structured Language
- Box-and-Arrow Charts
- Graphs
  - Flowgraphs
  - Callgraphs
  - Dataflow graphs
  - Petri Nets
- Databases
- Charts, Diagrams
- Mathematical formalisms (e.g. predicate logic)

Which of these are best for answering which types of questions for which types of stakeholders?

Roadmap

- Look at some of them now
  - Separate and apart from when, where, and how they might eventually be used
- Understand about "precise, detailed, broad, clear"
- Then examine needs at different points during software development
- We will see many combinations of the basic approaches
  - Understand the basic approaches first

Do We Need Anything More than a Programming Language?

- Code is the ultimate
- Describes what is actually being done
- What more can you ask for?
Programming Languages

- Primary stakeholder constituency is Developers
  - Helps answer: What does the product do?
- Less useful to other stakeholder constituencies:
  - How do you get the product to do what it does? (users)
  - What is the product supposed to do? (users/investors)
  - Will the product ever fail disastrously? (bystanders)
  - Is the product’s development (almost) done? (managers/investors)
  - How extensible/adaptable is the product? (maintainers/managers/investors)
- (Some) Differences in Programming Languages are intended to help them be (more) effective in answering some of these sorts of questions.
- But additional representations are more suitable for most of these questions and stakeholder constituencies.

Even Developers Need More

- "What does it do"—for
  - Complicated programs
  - Large programs
  - Badly written programs
  - Programs written in poor programming languages
- "How does it work"
  - NB: Languages that separate specification part from implementation part are attempting to address this question
  - "What would happen if I changed....."
  - "How should our team go about changing ...."
Advantages of Natural Language

- Easy to train users
- Clarity is possible (but may be difficult)
- Completeness is possible (but by no means assured)
- Easily modified
- It is the "least common denominator"

But-- Disadvantages are far more numerous and serious

Disciplined Use of Natural Language

- Response to natural language problems of:
  --imprecision
  --ambiguity
  --inconsistency (especially when due to size)
  --inability to reason effectively and definitively

- Familiar approaches:
  --Restricted use of reserved terms
  --Structuring (paragraph numbering, outline form, templates, etc.)

- Other, earlier examples of disciplined use of natural language:
  --Laws and other legal documents
  --Recipes
  --Help systems

Discipline Mechanisms in PSL

- Use of keywords (defined elsewhere in specification)
  --helps support consistency determination: some keyword fields have defined relations to others (eg. input-to and output-from)

- Use of templates
  --facilitates determination of completeness
  --fosters clarity
  --facilitates consistency checking

- Use of structure:
  HIERARCHY:
  --Standard practice for dealing with size, complexity
  --Exploits innate human capacity for abstraction
  DATA FLOW:
  --Can determine which data objects flow to what activities
  CONTROL FLOW:
  --Can see that some activities precede/follow others

Evaluation of Disciplined Natural Language

- Big step in the right direction: improvement over straight natural language
- Possible to determine some kinds of consistency thru:
  --mechanisms for reducing ambiguity
  --mechanisms for fostering completeness
  --structuring mechanisms for dealing with complexity

BUT--

- Stilted form reduces clarity: less suitable for some key stakeholder groups
- Some residual reliance on natural language means ambiguity remains
- Size is still a problem: PSL specs (for example) can be huge: consistency determination is long/error prone

Natural Language Disadvantages

- Determining consistency between natural language artifacts and anything else is hard/subjective
  --Ambiguity in natural language is easy and often intentional
  --Clear natural language expression is very difficult
  --The longer the text, the more information, the more the risk of inconsistency, the harder it is to determine

- No way of knowing when a specification is "complete"

Main Problems are:

- Cannot reason definitively about natural language
- Cannot be sure that natural language artifacts are consistent with other artifacts
- Assurances to stakeholders are shaky

A Software Engineering Example:

PSL (Problem Statement Language)

DESCRIPTION:
  this process performs those actions needed to interpret time records to produce a pay statement for each hourly employee.

KEYWORDS:
  independent:

ATTRIBUTES ARE:
  complexity-level high;

GENERATES:
  pay-statement, error-listing;

RECEIVES:
  time-card;

SUBPARTS ARE:
  hourly-paycheck-validation, hourly-emp-update, h-report-entry-generates, hourly-paycheck-production;

PART OF:
  payroll-processing;

DERIVES:
  pay-statement;

USING:
  time-card, hourly-employee-record;

DERIVES:
  hourly-employee-report;

USING:
  time-card, hourly-employee-record;

DERIVES:
  error-listing;

USING:
  time-card, hourly-employee-record;

PROCEDURE:
  read record, add up hours, multiply by pay rate......

HAPPENS:
  number-of-payments TIMES-PER pay-period;

TRIGGERED BY:
  hourly-emp-processing-event;

TERMINATION-CAUSES:
  new-employee-processing-event;

SECURITY IS:
  company-only;
Pictorial and Diagrammatic Approaches

- Diagrams composed of visual elements having rigorously defined semantics
- Diagrams used as modeling devices
- Diagrams depict key structural aspects of system
- Reduction or removal of natural language
- Clarity improved greatly
- Consistency improved greatly
- Completeness of diagrams facilitated
- Possibility of ambiguity considerably reduced

**ALTHOUGH:**

- Modifiability significantly reduced
- Completeness impeded by restrictions in semantics defined
  → more on these issues later.....

---

**Data Flow Diagram**

![Data Flow Diagram](image1)

**Control Flow Diagram**

![Control Flow Diagram](image2)

**Finite State Machine**

![Finite State Machine](image3)

**Petri Net**

![Petri Net](image4)

**Pictures are not enough**

- Want to be able to reason about them
  - Verify relations, properties
- Pictures can leave ambiguous impressions
- How to be sure what they say?
An Example: A Data Flow Diagram

What is this diagram specifying?

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

The power of annotation:

Improper use of notation is not saved by annotation here:

There is ambiguity and misuse of notation here:
- one circle is a test, others are functions
- some edge annotations are data, some predicates
- multiple arrows in and out "and" or "or"?
Very Simple DFD

Attempt to make this more precise

But iteration complicates matters

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), where r = (s_i, s_j, ..., s_k)
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: \( R = \{ (s_i, s_j) | \text{PRED}(s_i, s_j) = \text{True}, \text{for some predicate, PRED} \} \)

If the tuples are ordered, the relation is called an ordered relation
If the tuples, \( s_1, s_2, ..., s_n \) are unordered, the relation is an unordered relation

Some Examples

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

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

Let L = \{all statements L in a program, P\},
Define ImmFol = \{ (L_i, L_j) | the execution of L_j may immediately follow the execution of L for some execution of P \}
Some Properties of Relations

Some familiar properties of ordered binary relations, R, over the set S={s1,s2,...,sn};
- Symmetry: s R s, R s, for all s and s in S
- Reflexivity: s R s, for all s in S
- Transitivity: s R s1 and s1 R s2, s R s2, for all s, s1, s2 in S

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

If R = {(s, s)} is transitive, then C={(s, s)} there exists a sequence, i1,i2,...,in, such that s=R s i1, s i1 R s i2, s i2 R s in, ...
C={(s, s)} is called the transitive closure of R

Antisymmetry: s R s, R s, for all s in S

Examples

If S=all subroutines written in Fortran) s R s if and only if s calls s, then R is an irreflexive relation

Let PS={(c, c)}, all the statements in a program that consists of a set of modules, M={(m1,...,mn)}
INMOD={(c, c)|c and 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 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 arrowsheads.
  - If not, the graph is called an undirected graph.

Graphs

A Graph, G = (N, E) is an ordered pair, consisting of a node set, N, and an edge set, E = {(n1, n2)}.

If OG=(N, E) is an ordered graph with E={(n1, n2)}, then its unordered version, Ug=(N, U), where U={(<n1, n2>)}

Paths

A path, P, through an ordered graph G=(N, E) is a sequence of nodes, (n1, n2), (n3, n4), ..., (nk, nk+1)
such that nk+1 = nk for all 2<= k <= n.

A path, UP, thru an unordered graphUG=(N, U) is a sequence of nodes, <n1, n2>, <n3, n4>, ..., <nk, nk+1>
such that all of the <nk, nk+1> can be ordered to assure that nk+1 = nk for all 2<=k <=n.

In either case, n1 is called the start node and nk 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, n1, n2, 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.

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) and C is a collection of pairs of nodes (edges) with repetitions allowed (ie. 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 = (SN, E) where SN is a node set that is the union of two disjoint subsets, N. U N2, and no edge in E has both nodes in either N or N2.

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.

Differences in Graphs Result from Different Choices for Nodes & Relations

- Hierarchy:
  - Models "consists of" or "is a part of"
  - Key to divide-and-conquer approaches to understanding
- Data Flow:
  - Nodes represent sets of work that is generated/used
  - Each edge is a (data generated, data used) edge pair
- Control Flow:
  - Nodes represent units of functionality
  - (n, m) is an edge in this graph if and only if unit n can execute immediately after unit m executes (ImmFol relation)
- Finite State Machines:
  - Nodes represent all possible different "execution states"
  - (s, a) is an edge if and only if it is possible for state s to immediately succeed s1. Called a transition from s to s1.
  - Edges annotated with transition condition
  - Annotations are relations too
- Petri Nets:
  - Multiple node and edge types in the same diagram

Flowgraphs

Let S = (all statements s. in a program, P)
Let ImmFol = { (s, s) | The execution of s immediately follows the execution of s for some execution of P }

Then: If FG = (S, 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. that the program P comprises) }
Let CALLS = { (S, S) | S is directly invoked from S 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.
**Formalizing DFDs**

- DataFlow\((i, j)\) if node \(i\) creates data that node \(j\) uses
- INPUT—set of nodes \{input\(i\)\} such that input is a provider of input from external source
- OUTPUT—set of nodes \{output\(i\)\} such that output is a conveyor of computed artifacts to external source
- EdgeAnnotation\((e, operand)\) where operand is the identification of the artifact that flows along edge \(e\)
  - Preferably the data artifact is defined rigorously
- NodeAnnotation\((n, text)\) if the string text describes the functioning of node \(n\)

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?

**Adding Hierarchy**

- Supports increased detail
- Increased precision too
- But creates complications

**Consistency is a principal concern**

- Are the diagrams consistent with each other?
- Top view consistent with elaborations?
  - Arrows consistent
  - Data flows consistent
  - Other semantics?
- Invitation to subtle errors