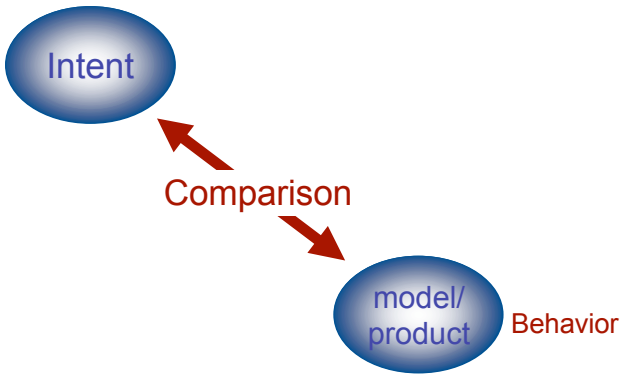
Announcements

- Revised Project 2 (minor) and new Project 3 posted
  - Note: each group (on-campus) needs to arrange a design review during the period 11/29-12/7
  - **No class on 12/6** (can be used for design review)
  - Due Date for Project 2 extended to 12/8
- As alternatives to Rational Rose, I have obtained licenses for Eclipse & Visual Paradigm; licenses are (will be available) on the website.
- I will have comments/grades on Project 1 on 11/29

21-Analysis Overview

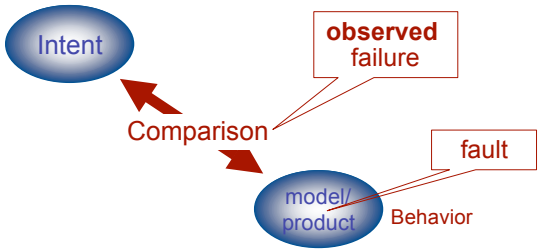
- Readings:
  - GJM03 Chapter 6
  - Fag86 Fagan,, M.E. "Advances in Software Inspections," IEEE Transactions on Software Engineering, July 1986, SE-12(7), pp. 744-751
  - Mil87 Mills, Harlan D., Michael Dyer, and Richard C. Linger, "Cleanroom Software Engineering," IEEE Software, September 1987, pp. 19-25
  - Ost76 Osterweil, Leon J. and L.D. Fosdick, " DAVE--A Validation Error Detection and Documentation System for FORTRAN Programs," Software Practice and Experience, September 1976, Vol. 6., pp. 473-486
  - Ole92 Olender, K. M. and L. J. Osterweil, "Interprocedural Static Analysis of Sequencing Constraints," ACM Transactions on Software Engineering and Methodology, January 1992, 1(1), pp. 21-52.
  - Dwy95 Dwyer , M. B. and Clarke, L. A. ÒA Flexible Architecture for Building Data Flow Analyzers," in CMPSCI Technical Report, August 17, 1995
  - Adr82 Adrion, W.R,. M.A. Branstad, and J.C. Cherniavsky, "Validation, Verification and Testing of Computer Software," ACM Computing Surveys, June 1982, pp.159--192
  - Hoa69 Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," Communications of the ACM, October 1969.
  - Flo67 Floyd, R.W. "Assigning Meaning to Programs", in the Proceedings of Symposium on Applied Mathematics, 1967, pp. 19-32, (Appeared as volume 19 of Mathematical Aspects of Computer Science).
  - Han76 Hantler, S.L. and J.C King,. "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys, September 1976, pp. 278-300.
  - Cla85 Clarke L. A. and D. J. Richardson, "Applications of Symbolic Evaluation," Journal of Systems and Software, January 1985, 5 (1), pp.15-35.

Analysis



Basic Definitions

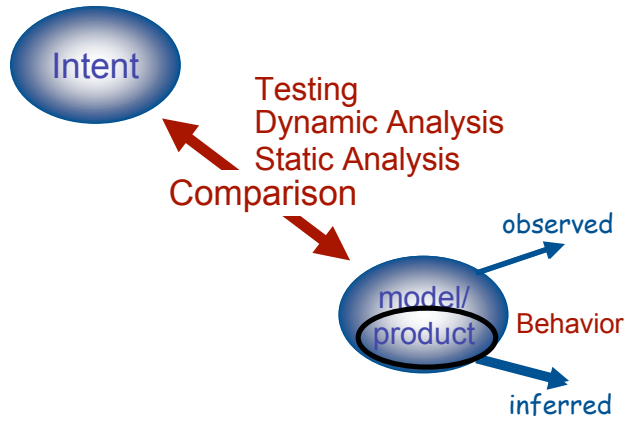
- Failure-- result that deviates from the expected or specified intent
- Fault/defect-- a flaw that could cause a failure
- Error -- erroneous belief that might have led to a flaw that could result in a failure



Approaches

- Static Analysis
  - the static examination of a product or a representation of the product for the purpose of inferring properties or characteristics
- Dynamic Analysis
  - the "interpretation" of a product or representation of a product for the purpose of inferring properties or characteristics
- Testing
  - the (systematic) selection and subsequent "execution" of sample inputs from a product's input space in order to infer information about the product's behavior.
    - usually trying to uncover failures
    - the most common form of dynamic analysis
- Debugging -- the search for the cause of a failure and subsequent repair

Analysis



Validation and Verification: V&V

- Validation -- techniques for assessing the quality of a software product
- Verification -- the use of analytic inference to (formally) prove that a product is consistent with a specification of its intent
  - the specification could be a selected property of interest or it could be a specification of all expected behaviors and qualities
    - e.g., provide a user-friendly and efficient ATM system for remotely depositing funds into and withdrawing funds from a checking or saving account
    - e.g., all deposit transactions for an individual will be completed before any withdrawal transaction will be initiated
  - a form of validation
  - usually achieved via some form of static analysis

Correctness

- a product is functionally correct if it satisfies all the functional requirement specifications
  - correctness is a mathematical property
  - requires a specification of intent
  - specifications are rarely complete
- a product is behaviorally correct if it satisfies all the specified behavioral requirements
  - difficult to prove poorly-quantified qualities such as user-friendly

Reliability

- measures the dependability of a product
  - the **probability** that a product will perform as expected
  - sometimes stated as a property of time
    e.g., mean time to failure
- Reliability vs. Correctness
  - reliability is relative, while correctness is absolute
  - given a "correct" specification, a correct product is reliable, but not necessarily vice versa

Robustness

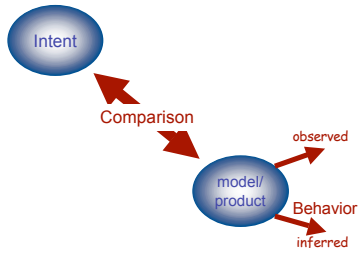
- behaves "reasonably" even in circumstances that were not expected
  - making a system robust more then doubles development costs
  - a system that is correct may not be robust, and vice versa

Formal models

- Analysis is usually done on a model of an artifact
  - textual representation of the artifact is translated into a model that is more amenable to analysis then the original representation
  - the translation may require syntactic and semantic analysis so that the model is as accurate as possible
    - e.g., x:= y + foo.bar
  - model must be appropriate for the intended analysis
- graphs are the most common forms of models used
  - e.g., abstract syntax graphs, control flow graphs, call graphs, reachability graphs, Petri nets, program dependence graphs

Modeling intent & artifacts

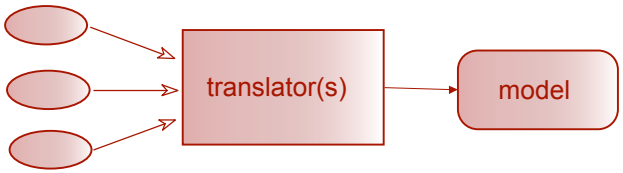
- natural language
- structured natural language
- pictorial notation
  - Charts, Diagrams, Box-and-Arrow Charts
  - Graphs
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
- data models
- formal language(s)
  - state-oriented
  - function-oriented
  - object-oriented



Ideally want general models

- different languages
  - e.g., Ada, C++, Java
- different levels of abstraction/detail
  - e.g., detailed design, arch. design
- different kinds of artifacts
  - e.g., code, designs, requirements

translate textual representations

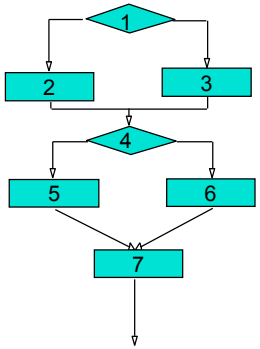


Static analysis

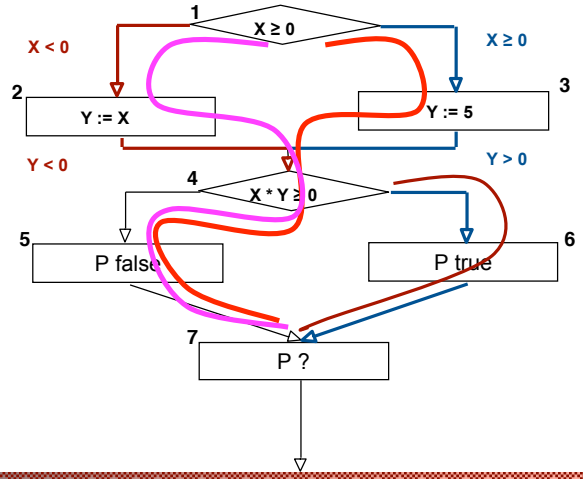
- typically conservative
  - never declare a property to be valid if it is not
  - usually achieve this by using representations that over-estimate actual behavior
  - the representation depends on the analysis
- AST is a conservative representation for
  - determining all the operators in a program
  - determining all the locations where X is defined
- CFG is a conservative representation for
  - Determining how many loops are in the program
  - determining how deeply nested each loop is

Conservative analysis in CFG

- For all execution sequences, is P true?
  - if P is true for all paths, then P is true
  - if P is true for some paths, then P may be true or false
    - Paths where P is not true may not be feasible
- For some execution sequence, is P true?
  - if P is true for some path, P may be true or false
    - the path where P is true may or may not be feasible
- Conservative analysis would only say P is true if is known to be true for all paths

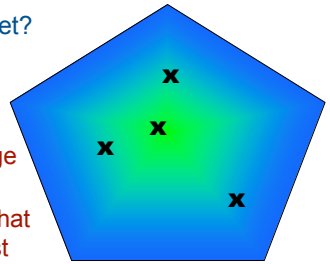


Example with an infeasible path



Dynamic analysis techniques

- draw inferences from a sample of the problem domain
- how do we choose that subset?
- Fault detection may depend upon
  - Specific combinations of statements, not just coverage of those statements
  - Astutely selected test data that reveals the fault, not just test data that executes the path



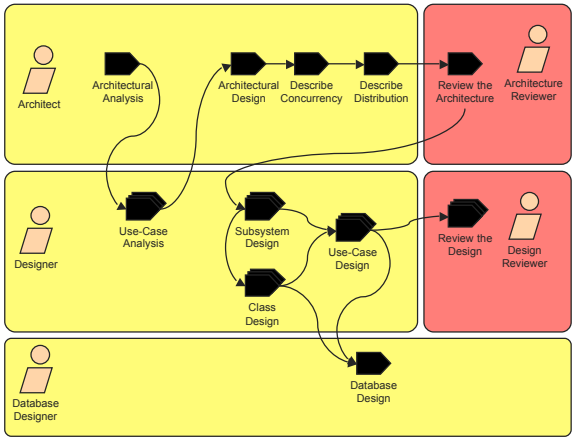
Approaches

- Static Analysis
  - Inspections
  - Software metrics
  - Symbolic execution
  - Dependence Analysis
  - Data flow analysis
  - Software Verification
- Dynamic Analysis
  - Assertions
  - Error seeding, mutation testing
  - Coverage criteria
  - Fault-based testing
  - Specification-based testing
  - Object-oriented testing
  - Regression testing

Reviews, Inspections, and Walkthroughs

- Manual static analysis methods
- Most can be applied at any step in the lifecycle
- Have been shown to improve reliability, but
  - often the first thing dropped when time is tight
  - labor intensive
  - often done informally, no data/history, not repeatable

Reviews in the RUP



Reviews, Inspections, and Walkthroughs

- Formal reviews
  - author or one reviewer leads a presentation of the product
  - review is driven by presentation, issues raised
- Walkthroughs
  - usually informal reviews of source code
  - step-by-step, line-by-line review
- Inspections
  - list of criteria drive review
  - properties not limited to error correction
  - historical context

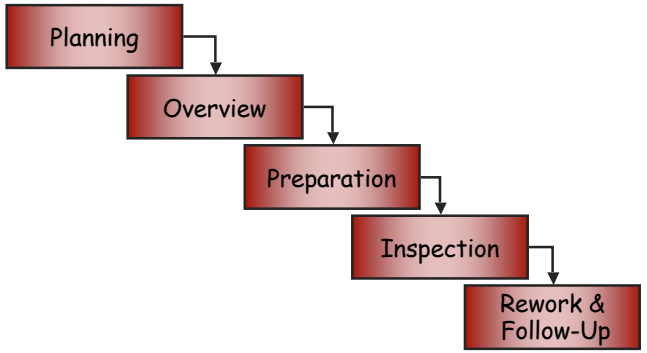
Review methods

- Fagan inspections
  - formal, multi-stage process
  - significant background & preparation
  - led by moderator
- Active design reviews
  - also called "phased inspections"
  - several brief reviews rather than one large review
  - guided by questions from the author
- Cleanroom
  - more than reviews, but reviews important component
  - we'll come back to this
- N-fold
  - parallel reviews controlled by moderator
  - focuses on user requirements

Fagan Inspections (3-5 participants)

- Moderator
  - Responsible for organizing, scheduling, distributing materials, and leading the session
- Author
  - Responsible for explaining the product
- Scribe
  - Responsible for recording bugs found
- Planner or designer
  - Author from a previous step in the software lifecycle
- User representative
  - To relate the product to what the user wants
- Peers of the author
  - Perhaps more experienced, perhaps less
- Apprentice
  - An observer who is there mostly to learn

Fagan Inspection Process (5 steps)



Fagan Inspection Process

- Planning ← moderator
  - Gather materials and insure that they meet entry criteria
  - Arrange for participants,
    - assign them roles,
    - insure their training
  - Arrange meeting
- Overview
  - explain content to the inspectors ← author(s)
- Preparation
  - Participants study material
- Inspection
  - Find/Report faults (Do not discuss alternative solutions)
- Rework
  - Author fixes all faults
- Follow-Up
  - Team certifies faults fixed and no new faults introduced

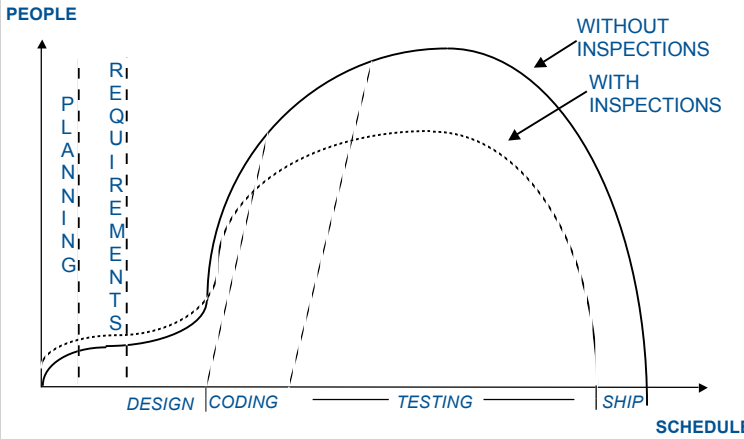
Fagan Inspection

- General guidelines
  - Distribute material ahead of time
  - Use a written checklist of what should be considered
    - e.g., functional testing guidelines
  - Criticize product, not the author

Experimental Results

- using software inspections has repeatedly been shown to be cost effective
- increases front-end costs
  - ~15% increase to development cost
- decreases overall cost
- IBM study
  - doubled number of lines of code produced per person
    - some of this due to inspection process
  - reduced faults by 2/3
  - found 60-90% of the faults
  - found faults close to when they are introduced
    - helps reduce cost

People Resource vs. Schedule



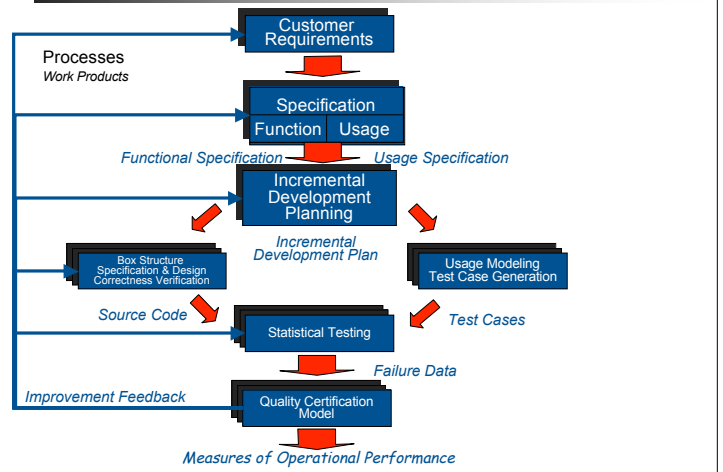
Why are inspections effective

- knowing the product will be scrutinized causes developers to produce a better product
- having others scrutinize a product increases the probability that faults will be found
- walkthroughs and reviews are not as formal as inspections, but appear to also be effective
  - hard to get empirical results

What are the deficiencies?

- focus on error detection
  - what about other "ilities" -- maintenance, portability, etc.
- not applied consistently & rigorously
  - inspection shows statistical improvement, but cannot ensure quality
  - inspection should have the same results without regard to the product to which it is applied or the inspection team
- range of errors not addressed
  - team expertise limited
  - one property may have many error modalities
- human intensive and often makes ineffective use of human resources
  - e.g., skilled software engineer reviewing coding standards, comments spelling, etc.
- no automated support
  - again inefficient of human resources
- aspects of review not used appropriately
  - e.g., in Fagan process, overview often covers what should be described if documentation is adequate

Cleanroom

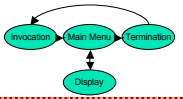


Cleanroom

```
[f]
do
    [g]
    [h]
od

For all
inputs, does
[g] followed
by [h] do
[f]?
```

- Verification as Review Process
  - team verification of correctness takes the place of individual unit testing; correctness is established by **group consensus** if it is obvious
  - by formal proof techniques if it is not.
- benefits
  - intellectual control of the process
  - motivates developers to deliver error-free code
  - verification is a form of peer review
  - each person assumes responsibility for and derives a sense of ownership in the evolving product
- every person must agree that the work is correct before it is accepted -> successes are ultimately team successes, and failures are team failures.
- Markov Analysis
- Factors
  - number of statistically typical (i.e., likely) usage paths through the software
- Steps
  - focus verification efforts,
  - identify the likelihood of given events,
  - project the test schedule, and
  - ascertain the (affordable) upper bound on inferences about reliability
- Stopping Criterion for Testing
  - goals (e.g., target level of estimated reliability) are achieved
  - or quality standards (e.g., errors/KLOC) are violated



Generation of Test Cases

- usage model->test cases
  - may be automatically generated.
- each test case is a random walk through the usage model
  - invocation->termination
- test cases constitute a "script" for use in testing
  - may be applied by human testers, or used as input to an automated test tool.
- Stopping Criterion for Testing
  - goals (e.g., target level of estimated reliability) are achieved
  - or quality standards (e.g., errors/KLOC) are violated
- Statistical Hypothesis Testing

Confidence level (%)

Reliability level (r) \ %	90	95	99	99.9
0.9	22	29	44	66
0.95	45	59	90	135
0.99	230	299	459	688
0.999	2302	2993	4603	6905

Software Metrics

- measures that predict qualities about software
- can be applied to any of the products (e.g., design, code, test cases) or to the process (e.g., Capability Maturity Model)
- Qualities measured by software metrics
  - performance
  - user-friendliness
  - resources
    - memory/storage
    - development costs
    - maintenance cost
  - quality
    - maintainabity
    - reliability
    - completeness
    - consistency
    - complexity

Function Points

- proposed by Albrecht in 1979
  - Originally applied to code
- UFP =
  number of inputs x w1 +
  number of outputs x w2 +
  number of user inquiries x w3
  number of files x w4 +
  number of external references x w5
- function points = UFP* TCA = UFP* (.65 + 0.01 * SUM(Fi))
  - where the degree of influence, DI= SUM(Fi) is the sum of complexity adjustment values, Fi
- metrics:
  - productivity: FP/person-month
  - quality: defects/FP
  - cost: $/FP

- weights:

	Simple	Average	Complex
w1	3	4	6
w2	3	5	7
w3	3	4	6
w4	7	10	15
w5	5	7	10

More Quality Metrics

- Modularity
  - cohesion metric
    - applied to unit design
    - the relationship among the elements of a module
    - best cohesion level is functional, and the worst is coincidental.
  - Cruickshank and Gaffney Cohesion Strength
    Strength = $\sqrt{(X^2 + Y^2)}$
    - where:
    - X = reciprocal of the number of assignment statements in the module
    - Y = number of unique function outputs divided by number of unique function inputs

More Quality Metrics

- Modularity
  - coupling
    - applied to system and unit designs
    - measure of the degree to which modules share data
    - data coupling (the sharing of data via parameter lists) is the best type of coupling, while common coupling (the sharing of data via global or common areas) is the worst.
    - a lower coupling value is better.
  - Cruickshank and Gaffney Coupling:
    - M_j = sum of the number of input and output items shared between components i & j
    - Z_i = average number of input and output items shared over m components with component i
    - n = number of components in the software product

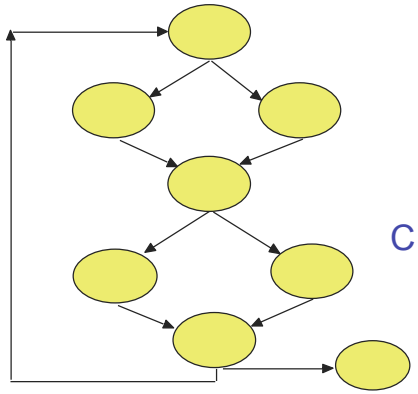
$$Coupling = \frac{\sum_{i=1}^{n} Z_i}{n}$$

$$where: \quad Z_i = \frac{\sum_{j=1}^{m} M_i}{m}$$

McCabe's cyclomatic complexity

- Complexity measured by control flow information
  - based on a control flow graph where e is number of edges, n is number of nodes, p is number of connected components
- McCabe's Cyclomatic Complexity:
  - v = e - n + 2
    - where:
    - v = complexity of the graph
    - e = number of edges (program flows between nodes)
    - n = number of nodes (sequential groups of program statements)
  - if a strongly connected graph is constructed (one in which there is an edge between the exit node and entry node), the calculation is
  - v = e - n + 1

Example



n = 8
e = 10
p = 1

C = 10 - 8 + 2 = 4

Software Science

- Halstead applied information theory to computer science
- metrics
  - η_1 **number of distinct operators**
  - η_2 **number of distinct operands**
  - $\mathcal{N}_1$ **total number of occurrences of operators**
  - $\mathcal{N}_2$ **total number of occurrences of operands**
- program level estimator

$$\mathcal{D} = 1 / \mathcal{L} = (\eta_1 / 2)(\mathcal{N}_2 / \eta_2)$$

$$\mathcal{L} = 1 / \mathcal{D} = (2 / \eta_1)(\eta_2 / \mathcal{N}_2)$$

difficulty increases as operators are introduced (η_1 /2 increases) and as operands are used repetitively ($\mathcal{N}_2$ / η_2 increases)

- programming time

$$\mathcal{T} = \mathcal{E} / \mathcal{S}$$

where $\mathcal{S}$ is the "Stroud number"

$5 \le \mathcal{S} \le 20$, usually 18

Software Science (continued)

- language level

$$\lambda = \mathcal{L} \times \mathcal{V}^* = \mathcal{L}^2 \mathcal{V}^*$$

$\lambda_{PL/1} = 1.53$, $\lambda_{Algol} = 1.21$,
$\lambda_{Fortran} = 1.14$, $\lambda_{CDC\ assmblr} = 0.88$

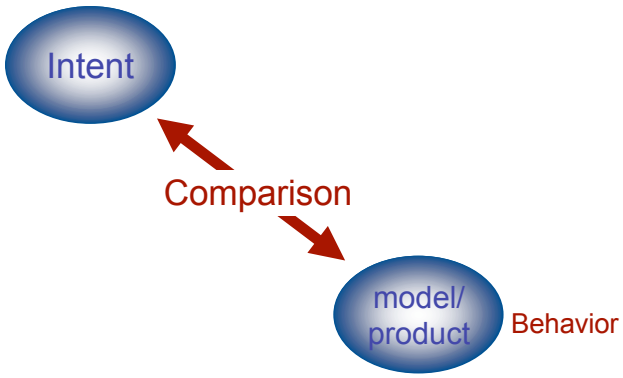
- predicted effort

$$\mathcal{E} = \mathcal{V}^{*3} / \lambda^2$$

Quality Metrics for Code

- Understandability
  - size metrics
    - lines of code
    - function points
    - function count
  - traceability metrics
    - number of comment lines per total source lines of code
    - percent comment lines of total lines
    - correctness of comments
- Predicting quality
  - LOC X domain seems to be the most reliable predictor

Analysis



Basic Verification Strategy

- analyze a system for desired properties, i.e., compare behavior to intent
  - intent
    - can be expressed as properties of a model
    - can be expressed as formulas in mathematical logic
  - behavior
    - can be observed as software executes
    - can be inferred from a model
    - can be expressed as formulas in mathematical logic
  - different representations support different sorts of inferences

Compare behavior to intent

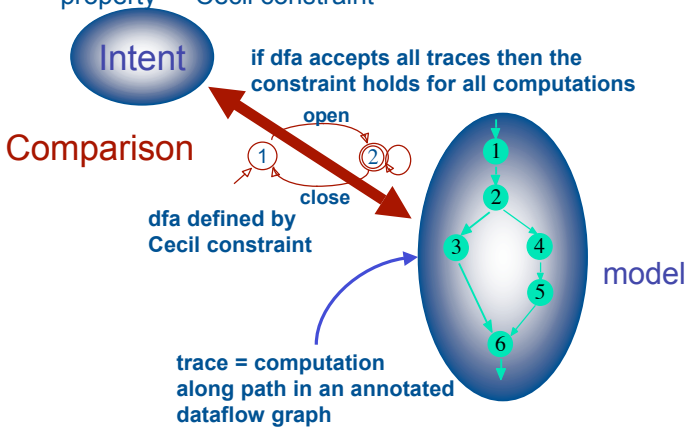
- comparison can be informal
  - done by human eye, e.g., inspection
- can be done by computers
  - comparing text strings
- can be done by model-checkers
  - such as formal machines (e.g., fsa's)
- can be done by rigorous mathematical reasoning

Example: Dataflow Analysis

- intent:
  - stated as a property
  - captured as an event sequence
- behavior:
  - model represents some execution characteristics
  - inferred from a model: (e.g., annotated flow graph)
  - inferences based upon:
    - semantics of flow graph
    - semantics captured by annotations
- comparison:
  - done by a fsa (e.g., a property automaton)

Data FlowAnalysis

"property" = Cecil constraint



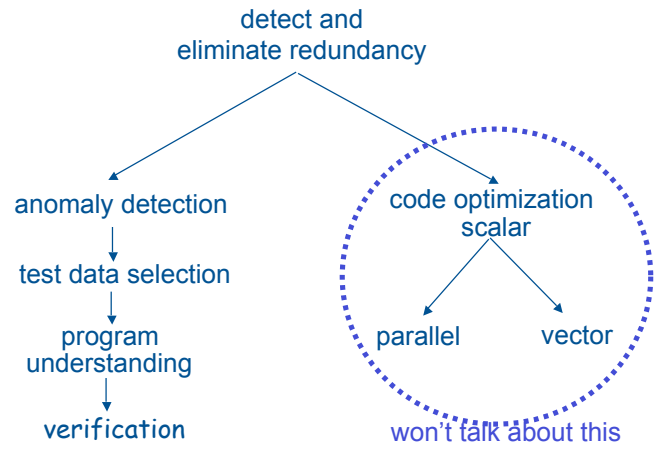
Data Flow Analysis (DFA)

- Uses an annotated control flow graph model of the program
  - Compute facts for each node
  - Use the flow in the graph to compute facts about the whole program
- DFA used extensively in program optimization, e.g.,
  - determine if a definition is dead (and can be removed)
  - determine if a variable always has a constant value
  - determine if an assertion is always true and can be removed
- Some Dataflow systems
  - DAVE system demonstrated the ability to find def/ref anomalies
  - Cecil/Cesar system demonstrated the ability to prove general user-specified properties
  - FLAVERS demonstrated applicability to concurrent system

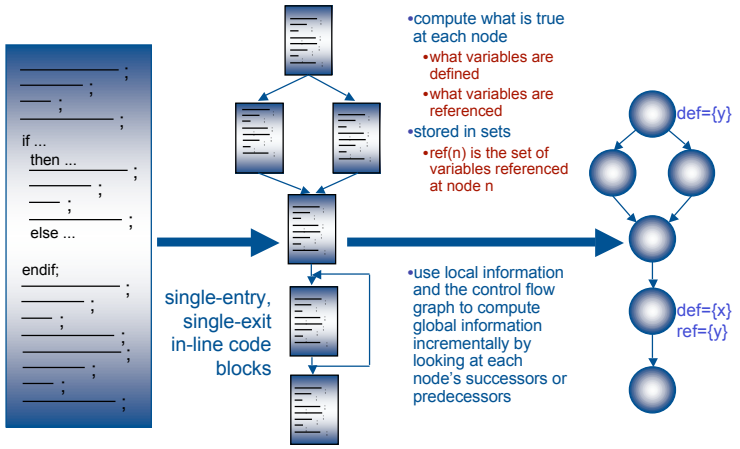
Data flow analysis

- computes information that is true at each node in the CFG, e.g.,
  - what variables are defined
  - what variables are referenced
- usually stored in sets
  - ref(n) is the set of variables referenced at node n
- uses this local information and the control flow graph to compute global information about the whole program
  - done incrementally by looking at each node's successors or predecessors

Data Flow Analysis



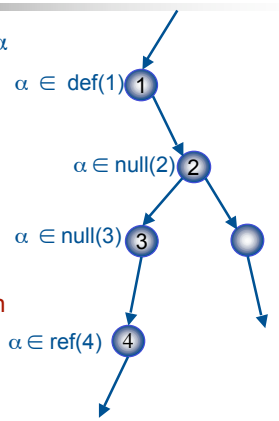
DFA



- compute what is true at each node
  - what variables are defined
  - what variables are referenced
- stored in sets
  - ref(n) is the set of variables referenced at node n
- use local information and the control flow graph to compute global information incrementally by looking at each node's successors or predecessors

Def-ref path expressions

- for a path P and a variable α can write a path expression describing the sequence of set memberships encountered for a, where
  - $\alpha \in$ def (n) or
  - $\alpha \in$ ref(n) or
  - $\alpha \in$ null (n)
  - for each node n on the path
- write (and simplify) a path expression
  - $P(n_1, n_1, \ldots, n_1; \alpha)$



$P(1,2,3,4; \alpha) = d11r = dr$

Anomalous pairs of ref/defs

d - defined, r - referenced, u - undefined

dd	bug?	du	bug?
dr	normal	ud	normal
uu	harmless?	rr	normal
rd	normal	ru	normal
ur	bug		

unreferenced definition

undefined reference

Consider unreferenced definition

- Want to know if a def is not going to be referenced
  - dd or du
- At the point of a definition of a, want to know if there is some path where a is defined or undefined before being used
  - May be indicative of a problem if the path is executable
  - Usually just a programming convenience and not a problem
- At the point of a definition of a, want to know if on all paths a is defined or undefined before being used
  - May be indicative of a problem
  - Or could just be wasteful

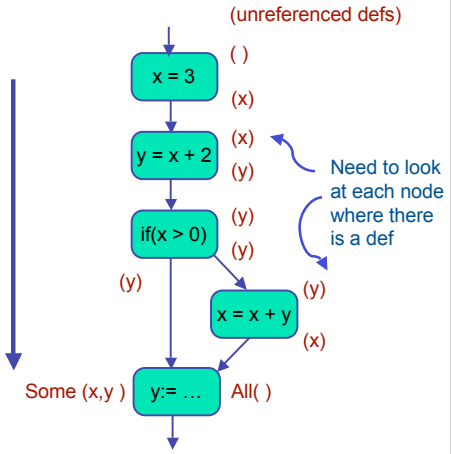
global dataflow analysis

- classes
  - forward flow problems (e.g., available expressions)
    - what definitions can affect computations at a given point in a program
  - backward flow problems (e.g., live variables)
    - what uses that follow a given point in the program can be affected by computations up to that point
- paths
  - any path
  - all path

Unreferenced definitions

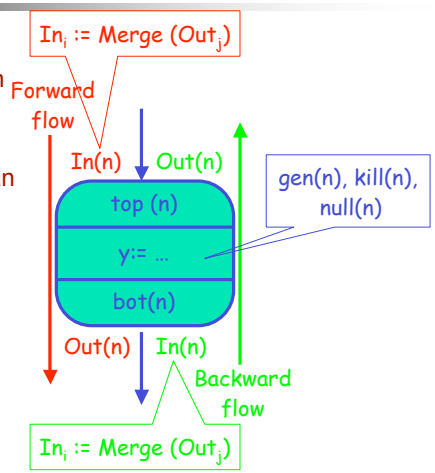
```
int x,y;
...
x := 3;
y := x + 2;
if x > 0 then
  x := x + y;
end if;
y:= …
```

Forward flow,
all paths problem



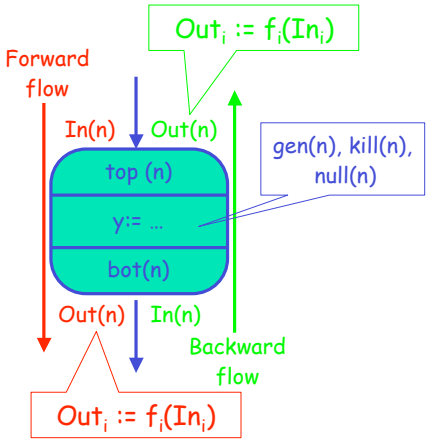
General Approach

- Initial values
  - for each node define gen and kill information
- Input Equations
  - for each node we have an equation of the form:
    In_i := Merge (Out_j)
  - "Merge" operation over the "predecessors" of n_i



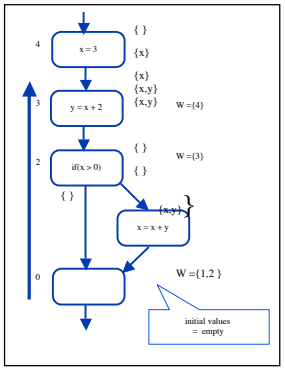
General Approach

- Transfer Equations
  - for each node we have an equation of the form: $Out_i := f_i(In_i)$
  - Transfer functions usually depend on Gen/Kill information that is computed for each node
  - Usually:
    Out := (In - kill)U gen
- We can view the set of variables, transfer functions, and flow graph as a system of equations



worklist algorithm

1. Start at initial node (entry for forward; exit for reverse), label IN_0 with pertinent "facts" (initial values)
2. Compute $OUT_0 = F(IN_0)$ (label OUT_0 with the computed facts)
3. Propagate OUT_0 to IN_i (label edge $N_0 \Rightarrow N_i$ with OUT_0) where N_i are successor nodes (forward) or predecessor nodes (reverse) of N_0
4. Compute $OUT_i = F(IN_i)$, place all N_i on a "worklist" W, and for all N_i label OUT_i with the computed facts.
5. While W is not empty,
   1. pick N_i from W and propagate OUT_i to IN_k (label edges $N_i \Rightarrow N_k$ with OUT_i) where N_k are successor nodes (forward) or predecessor nodes (reverse) for N_i ; delete N_i from W
   2. Compute $OUT_i = F(IN_k)$ for all N_k where IN_k=MERGE all input edge labels (MERGE = ∪ for "some paths" and ∩ for "all paths"), label OUT_k with the computed facts); and if for N_k, OUT_k changes put N_k, on W'
6. If W' is not empty, then W=W' and go to 5

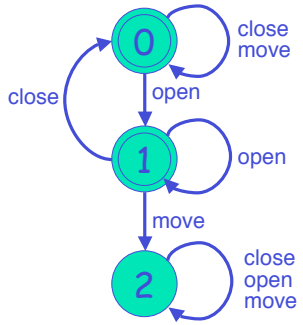


Using Quantified Regular Expressions

- Alphabet, quantification, regular expression
- For the events {open, close, move}

  show that for all paths:

  ((close v move)*,
  (open⁺ v open⁺,close))*



Cecil: Olender and Osterweil

- Instead of implicitly defined facts, let the user define application-specific facts
  - Represented as a Deterministic Finite State Automaton (DFSA) or as a Quantified Regular Expression (QRE)
  - Events
    - Recognizable events
    - Method calls
      - Can reason about sequences of method calls
      - E.g.,Push must be called before Pop
    - Thread interactions
      - Join or Fork
    - Arbitrary operations
      - a+b
    - Need to be able to treat events as indivisible actions
      - E.g., can treat pop and push as atomic as long as they do not contain any events of concern
- Propagate the states in the DFSA that can reach each node in the program

State Propagation

- States of the property are propagated through the CFG
- The property is proved if only accepting (non-accepting) states are contained in the final node of the CFG
- Cecil DFSA ->

  lattice ($\mathcal{P}$(S), $\subset$, $\cup$)

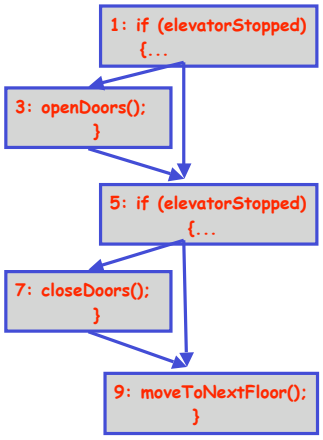
  function space

  $\delta : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$

  facts at nodes are elements of $\mathcal{P}$(S)
- propagate until convergence and check if terminal node in an accepting state of DFSA

Elevator Controller

```
    void main()
    {
    ...
1:  if (elevatorStopped)
    {...
3:          openDoors();
    }
    ...
5:  if (elevatorStopped)
    {...
7:          closeDoors();
    }
9:  moveToNextFloor();
    }
```



- States of the property are propagated through the CFG
- For an **all** property: the property is proved if only accepting states are contained in the final node of the CFG
- For a **none** property: the property is proved if only non-accepting states are contained in the final node of the CFG

State propagation