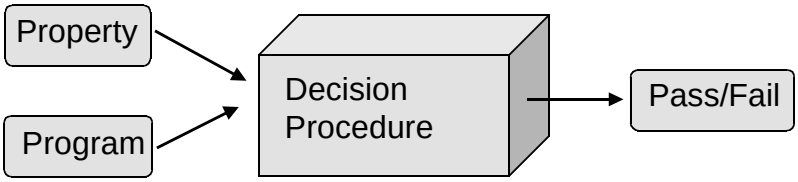
Software Test & Analysis (a.k.a. Verification)

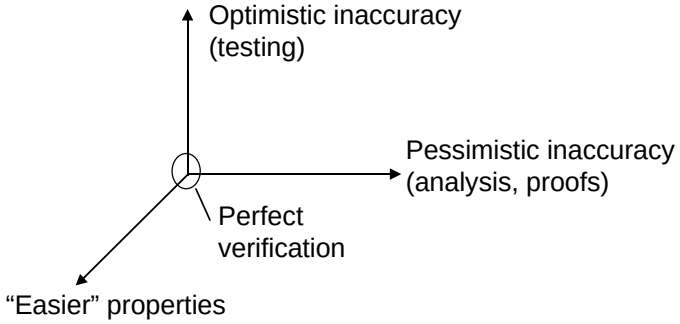
You can't always get what you want,
but if you try sometime,
you just might find,
you get what you need.
–Rolling Stones

You can't always get what you want



- Correctness properties are undecidable
  - the halting problem can be embedded in almost every property of interest

Getting what you need ...



- We must make the problem of verification "easier" by permitting some kind of inaccuracy

Static Analysis

Software Inspections

Formal Verification

Flow Analysis

. . .

"Static" analysis: without execution

- An over-broad category that encompasses nearly everything except testing
- General themes:
  - Early analysis: design and specs as well as code
  - Focused, efficient analysis; not comprehensive
  - Thorough (often conservative) analysis
  - Information gathering for other test & analysis

Software Inspection:
Low tech but effective

- Fagan Code Inspections
  - One of many "walk-through" and inspection techniques; among the most successful
    - More formal and well-defined than "structured walk-throughs" etc.
  - Has been extended to designs, requirements, etc. with similar organizing principles
  - A completely manual technique for finding and correcting errors

See also:
www.ics.Hawaii.edu/~johnson/FTR/Bib/bib-master.html

Software Inspection Roles

- Moderator:
  - Typically borrowed from another project. Chairs meeting, chooses participants, controls process
- Readers, Testers:
  - Read code to group, look for flaws
- Author:
  - Passive participant; answer questions when asked

Software Inspection Process

- Planning
  - Moderator checks entry criteria, choose participants, schedule meeting
- Overview
  - Provide background education, assign roles
- Inspection (see ahead)
- Rework
  - author addresses defects found in inspection
- Follow-up
  - Possible re-inspection

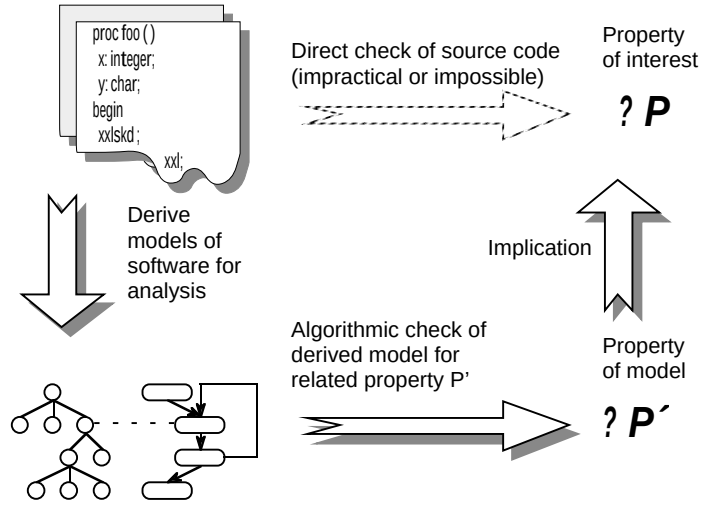
In the Meeting

- Goal: Find as many faults as possible
  - max 2 x 2 hour sessions per day
  - approx. 150 source lines/hour
- Approach: Line-by-line paraphrasing
  - Reconstruct intent of code from source
  - May also “hand test”
- Find and log defects, but don’t fix them
  - Moderator responsible for staying on track

Why does inspection work?

- The evidence says it is cost-effective. Why?
  - Detailed, formal process, with record keeping
  - Check-lists; self-improving process
  - Social aspects of process, esp. for author
  - Consideration of whole input space
  - Applies to incomplete programs
- Limitations
  - Scale: Inherently a unit-level technique
  - Non-incremental; what about evolution?

Analysis of Models



Classic data flow analyses to find program errors

- Uninitialized variable
  - "May" result from classic "avail" analysis
    - but conservative analysis can be annoying
    - "Must" version is also possible (how?)
- Dead assignment (no possible use)
  - Classic "live variables" analysis
  - In FORTRAN, Awk, BASIC, PERL, etc., usually indicates a misspelled variable
  - less useful in languages requiring declarations

Example analysis: DAVE

- Data flow analysis tools for error detection [Fosdick & Osterweil 1976]
- Classical data flow analysis algorithms
- Detected FORTRAN coding anomalies
  - Uninitialized variables
  - Dead stores (indication of wrong variable)
- "Must" and "May" results
  - overly conservative: too many "may" results

Precision & Safety

- An analysis is conservative (safe) if it doesn't miss errors
- An analysis is precise to the extent that it doesn't report spurious errors
  - An overly conservative, imprecise analysis may be useless.
  - A well-defined but overly strict property may be preferable to spurious error reports

Why is analysis imprecise?

- Not all program paths are executable
  - The same infeasible path problem as test coverage; perfectly precise analysis is impossible
- Precision is costly
  - Static analyses "summarize" results to obtain results in practical time (often $O(n^3)$ in theory, O(n) in practice). Precise results often require exponential time and space.

Aspect analysis

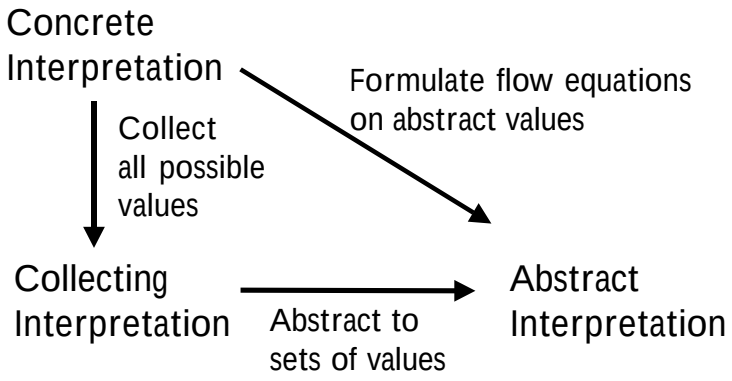
[Jackson 93]

- Classical data dependence analysis
  - with three differences
    - User-specified dependence properties
    - Compositional: Specs can be used in lieu of code
    - Dependence between abstract components (finer than dependence between concrete objects)
- Reports *missing* dependencies
- Reports only *must* results

Non-standard analyses

- Flow analysis doesn't *have* to be about data flow
  - the formal requirements don't say anything about data flow; they just describe a set of equations about approximate values
- Sometimes we can abstract in different ways from program execution
- Sometimes we can use the same methods for other systems of equations

Collection & Abstraction



How to cook an analysis

- Choose a "collecting" interpretation
  - execution in which a location "collects" every value
  - usually infinite
- Abstract to a finite-height lattice
  - with appropriate transfer functions
  - often (but not always) subsets of values

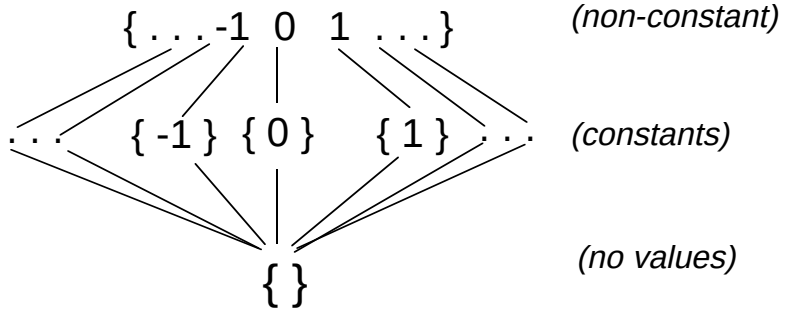
Java stack typing

- Java is compiled to op-codes for a virtual machine; the op-codes manipulate a stack of intermediate values
- For safety and efficiency, Java types the stack:
  - At every point in the program, the height of the stack is known
    - No stack overflow/underflow checks needed
  - The type of the object at the top of the stack is known

Measuring the stack

- At procedure entry, stack height is zero
  - this could be generalized to relative height
- Each stack operation has a predictable effect
  - e.g, ADDI reduces stack height by 1
- But what about control flow (if, while)?
  - you should be able to concoct a lattice of values for this; recall constant propogation

Lattice of stack heights



- Exactly as for constant propogation
- What are the flow equations?

Flow equations for stack height

- For a stack operation,
  out(b) = f (in (b)), where f is change in height
- For control flow join,
  out(b) = Merge(in(b))
  where Merge(x,x) = x
  Merge(x,y) = { -infinity .. infinity } if x≠ y

Treating control flow join as pseudo-node

- For other operations, out(b) = in(b)

Extend stack height analysis to stack type analysis

- In place of heights, propagate vectors of types
  - not as expensive as it sounds, since height should always be a constant
- Extend stack operations and Merge(x,y) in the obvious way
  - ADDI takes ...., i,i to ...., i
- ?? is a vector of unknown height, unknown type; Merge(??,x) = ?? for every value x

What is static analysis good for?

- Not a replacement for testing
  - focused, (mostly) automated analysis for limited classes of faults
- More thorough than testing (within scope)
  - conservative analyses are tantamount to formal verification
- Also augments testing, e.g., dependence analysis for data flow testing

Summary: Analysis & Test

- No panaceas:
  You can't always get what you want
- Multiple approaches
  - Complementary: Different strengths and weaknesses
  - Automate what you can
  - Analyze what is hard to test
  - Use testing to complete the analysis, and add a "reality check" for analysis of models