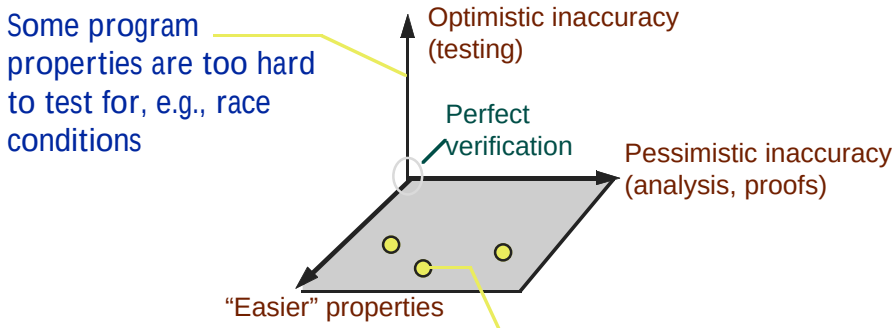
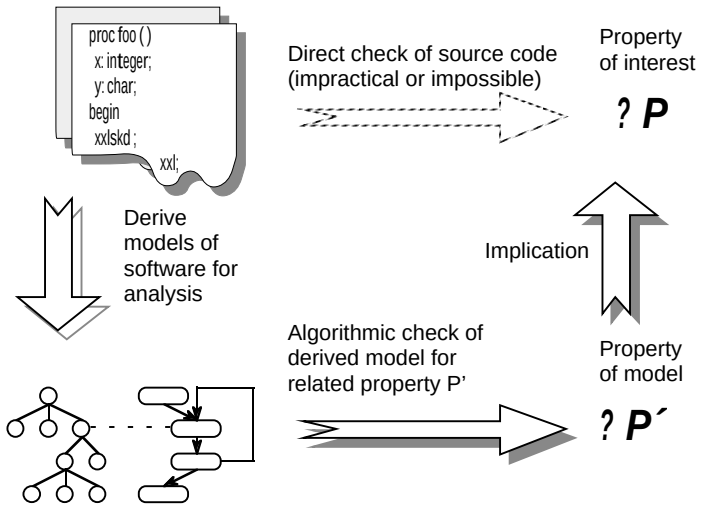
Analysis where testing fails ...



We can't prove program correctness, but we can prove (simple) properties of (simplified) models

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

The Classic Analyses

	Forward	Backward
Any path	**Reaching definitions** The assignments that produced current variable values	**Live variables** Variables whose current values may be used later
All paths	**Available expressions** Computed expressions whose values have not changed	**Very busy expressions** Expressions that are always evaluated (in a loop)

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.

Conservative Analysis

- Flow analysis considers all program paths
  - both directions at every branch
  - includes some unexecutable paths
- Flow analysis propogates estimates of actual values
- Correctness condition: Estimates are always *conservative*

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

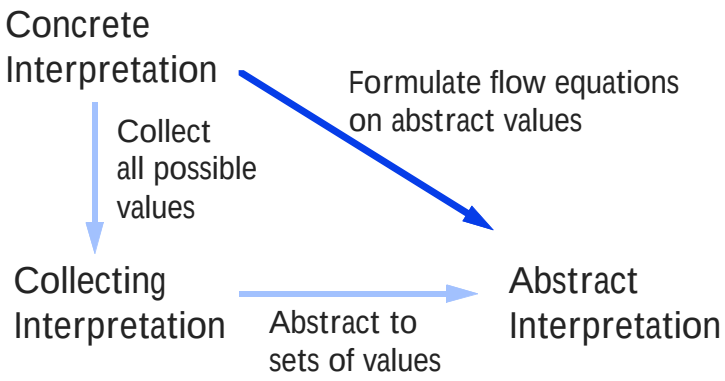
Non-standard analyses

- 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

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

Collection & Abstraction



Generalized flow analysis: Cesar
Olender & Osterweil 88

- Specify sequencing properties as regular expressions
  - symbols represent operations that can be identified in the program text
- Produce DFA (state-machine) accepter
- Propogate DFA states through program

Regular expressions as specs

- Alphabet is program events (identifiable in source code)
- Spec describes allowed pattern

```
(  (OpenR, Read*, Close)
 | (OpenW, (Read | Write)*, Close)
)*
```

Is the language of the program control flow graph contained in the language of the specification?

Path expression check

- Lattice: Sets of DFA nodes, top is all nodes
- Confluence is union
- Flow equations: extended transition relation
  - dfa nodes $\times$ symbol $\rightarrow$ nodes
  - monotonic: more dfa nodes in domain makes more dfa nodes in range
- Accept if *ONLY* accepting states at end

Semaphore Order Check

- In operating systems and other concurrent systems, a common discipline is to impose an order on semaphores
  - If A is ever locked when a lock is requested for B, then $A \leq B$; the relation $\leq$ must not contain cycles
- A partial order over semaphores (no cycles in $\leq$) is a sufficient condition to prevent deadlock

Recognizing semaphore order

- Assume we can recognize calls to P (lock) and V (unlock) for each semaphore
- Easy to formulate flow equations in a single procedure
- ... but we need a global analysis over the whole program or system

Semaphore analysis in the call graph

- First pass: Build up Gen/Kill sets of semaphores for each procedure in the call graph
  - Gen: All P operations
  - Kill: All V operations
- Second pass: Propogate sets of potentially locked semaphores

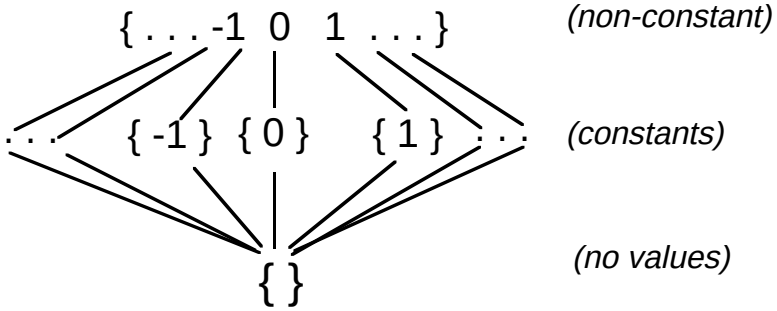
Java stack typing

- 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

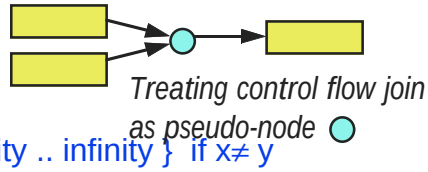
Lattice of stack heights



- 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



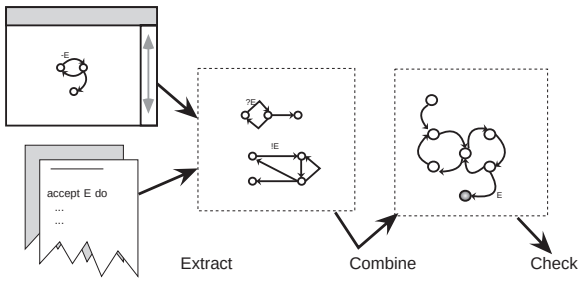
- For other operations, out(b) = in(b)

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

Analysis of Models (example): State-Space Exploration

- Concurrency (multi-threading, distributed programming, ...) makes testing harder
  - introduces non-determinism; time- and load-dependent bugs escape extensive testing
- Finite-state models can be exhaustively verified



Automated Finite-State Verification

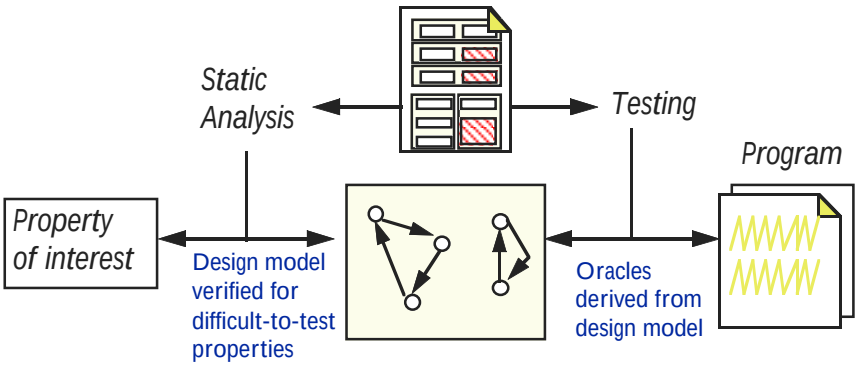
G. Holzmann, “The model checker SPIN.” IEEE TSE 23(5), May 1997

- Example tool SPIN (one of many)
  - verifies simple program-like design model
    - high-level design of process interaction, ignoring other aspects of computation (e.g., functional behavior)
  - used for protocols, OS scheduling, ...
    - useful despite limited capacity; best for verifying high-level design before coding
- Domain-specific analysis
  - limited “proof” of simple but critical properties in a limited domain

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

Combining Analysis and Test



Testing for conformance to a verified design model can be more effective than directly testing for a property of interest.