
Static Analysis

Analysis of Models

Data flow analyses

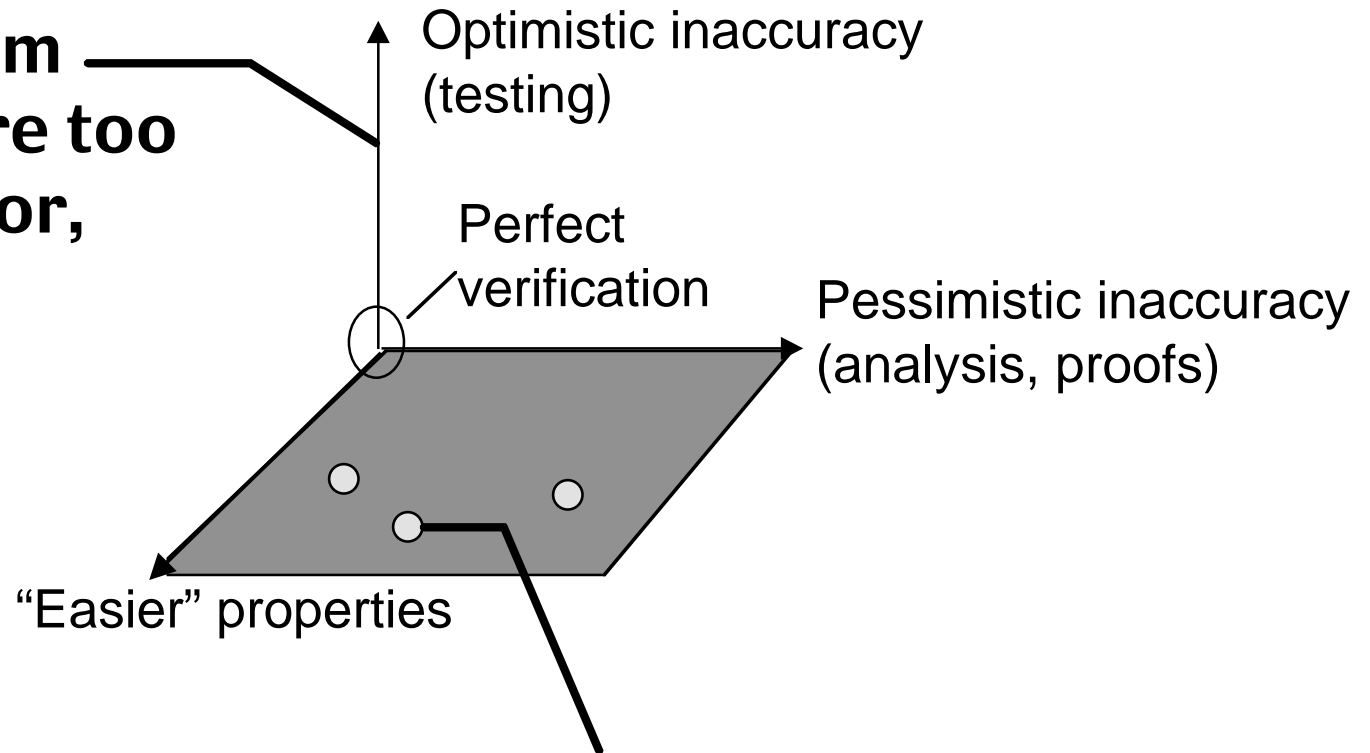
Finite-state verification

Static Analysis

- **Static = without program execution**
 - **An over-broad term, includes almost everything except conventional testing**
- **Why?**
 - **Because dynamic testing requires running code; analysis can be applied earlier in development**
 - **Because some kinds of defects are hard to find by testing (e.g., timing-dependent errors)**
 - **Because testing and analysis are complementary; each is best at finding different faults**

Analysis where testing fails ...

Some program properties are too hard to test for, e.g., race conditions



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

Static Analysis: Low and High Tech

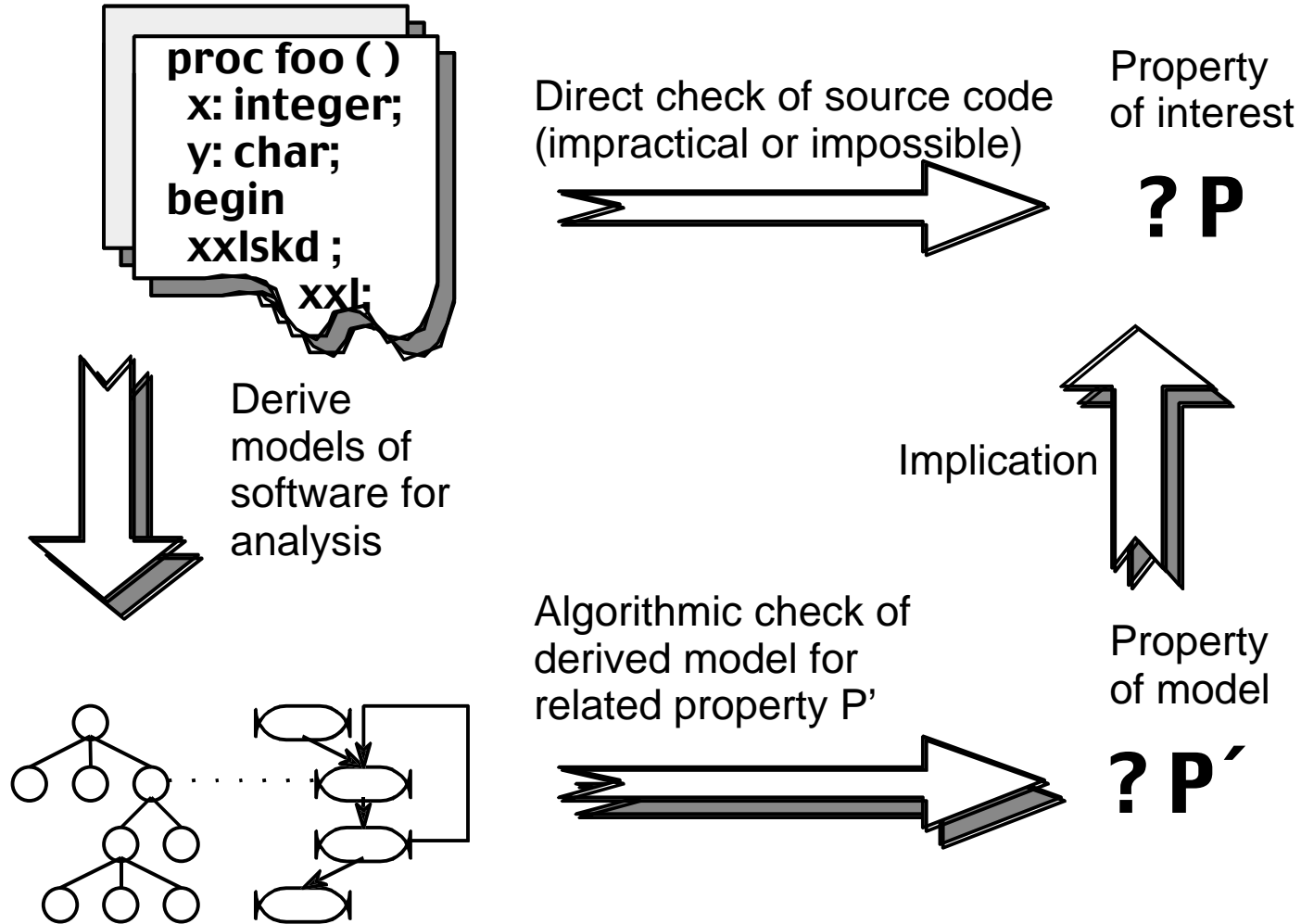
- **Low-tech static analysis:**
 - Software inspection
 - Simple syntactic standards and manual checks
- **High-tech static analysis**
 - Enforced syntactic checks
 - Well-formedness checks in specifications, designs, and code (e.g., matching connectors in design diagrams)
 - Automated program analyses
 - Often based on data flow analysis
 - Finite-state verification and other “high-power” analyses of models

Example:

Uninitialized pointer

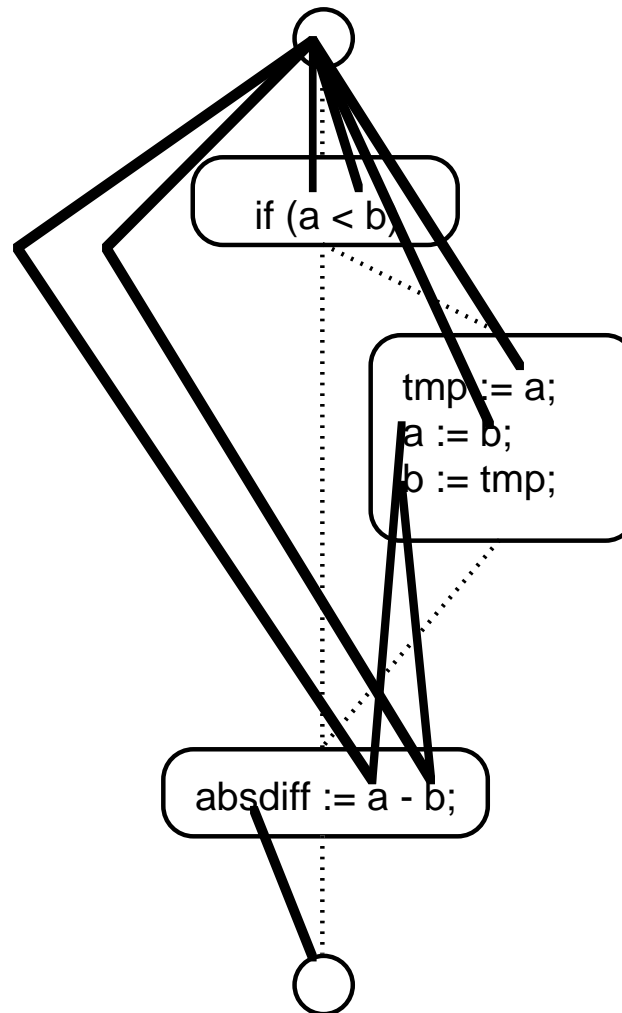
- **Dynamic test: easy to test on a particular execution, but impossible to know if we've covered every possible case**
- **Static (data flow) check: three possible outcomes, one is “maybe” (depending on execution path)**
 - exact static check is not possible in general
- **Easier (stronger) property: enforce initialization idiom, ban explicit deallocation (as in Java)**

Analysis of Models



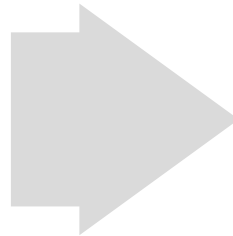
Data flow analysis

```
function absdiff (a, b: integer)
  return integer is
    tmp: integer;
begin
  if (a < b) then
    tmp := a;
    a := b;
    b := tmp;
  end if;
  return ( a - b );
end absdiff;
```



Data Flow as Abstraction

```
23  if (a < b) {  
24      tmp = a;  
25      a = b;  
26      b = tmp;  
    }
```



```
23  ref={a,b} def={ }  
24  ref={a}  def={tmp}  
25  ref={b}  def={a}  
26  ref={tmp} def={b}  
    }
```

- **Abstract from (location, computation) to (location, reference & definition)**
- **Compute approximations to possible usages**
- **Recall: Why must this analysis be approximate?**

Example analysis: DAVE

Fosdick & Osterweil, "Data Flow Analysis in Software Reliability." ACM Surveys, Sept 1976.

- **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

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)

Classic data flow analyses to find program errors

- **Available variable analysis (forward flow)**
 - a definition is “avail” if it can affect computation at the current location
 - *uninitialized variable* indicates an error
- **Live variable analysis (backward flow)**
 - a variable is “live” if its current value may be used later
 - *dead assignment* probably indicates an error

Quick Question ...

- **Would the DAVE analyses be as useful for Java or Pascal as for FORTRAN?**
 - Why or why not?
- **Would the DAVE analyses be more or less useful in TCL, Perl, Awk, Bash, or your favorite scripting language? Why?**

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**
- **Static flow analysis considers all (syntactic) program paths; it can be conservative or precise, but not both**
 - **An overly conservative, imprecise analysis may be useless.**
 - **A well-defined but overly strict property may be preferable to spurious error reports**

Conservative Analysis

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

Aspect analysis

D. Jackson, "Abstract Analysis with Aspect."
Proc. ISSTA, June 1993.

- **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**

Slicing: Data flow in debugging

- A *slice* is the set of program statements that “affected” a given value
- Originally, static slice defined by Weiser
 - All the statements that “could affect” a value
 - Essentially a reaching definitions analysis, with control flow
- Dynamic slice: for a single execution
 - All the statements that “did affect” a value
 - Subset of static slice

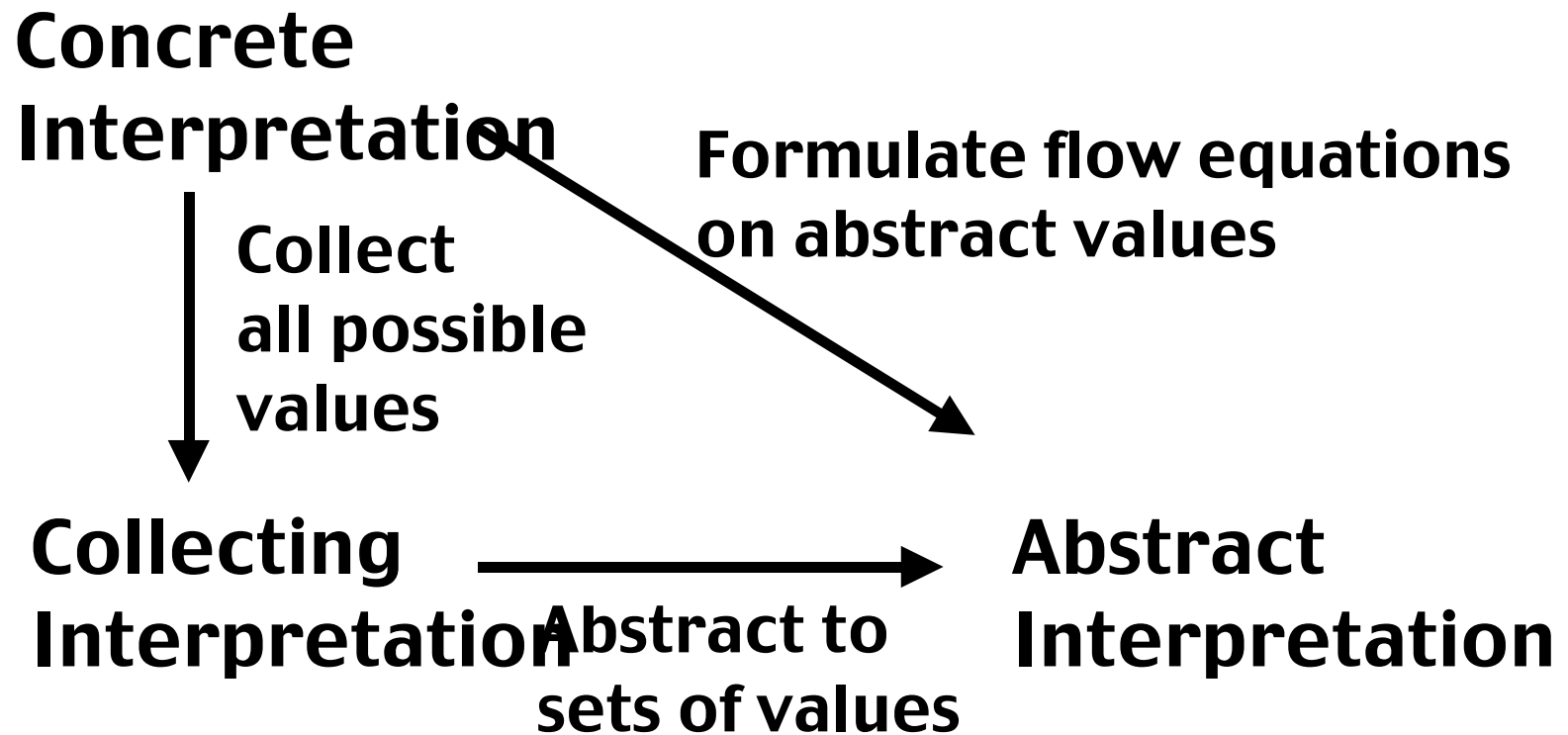
Non-standard analyses

- **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:

Cecil/Cesar

Olender & Osterweil, "Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation." IEEE TSE 16(3), March 1990.

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

FLAVERS: *Flow Analysis & Verification*

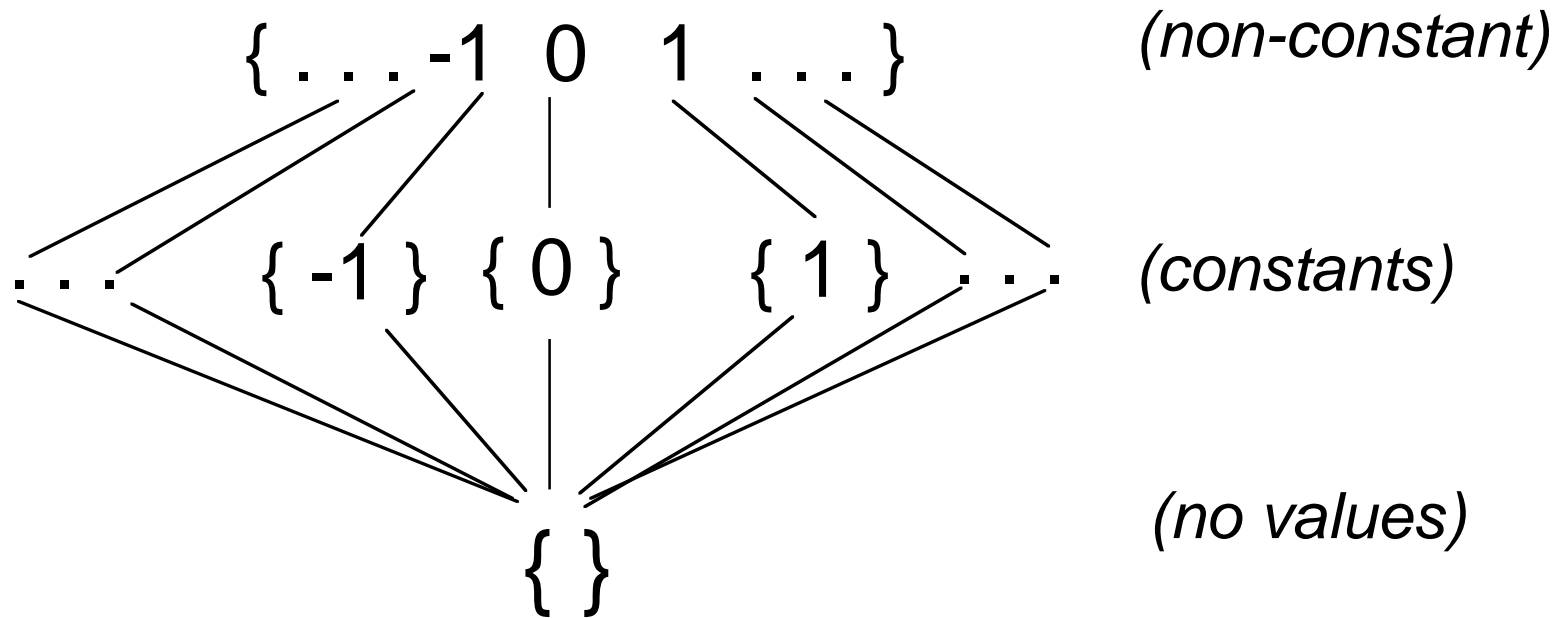
Dwyer & Clarke, "A Flexible Architecture for Building Data Flow Analyzers." Proc. ICSE, March 1996.

- **Reasons about a model of program executions**
 - e.g., Ada programs annotated with events of interest
- **Verifies the presence or absence of a property, specified as a sequence of events**
- **Cost-effective, incremental technology**
 - based on standard data flow analysis algorithms
 - analysis can be made more accurate through program model refinement or analysis feasibility constraints
- **Toolkit for prototyping of flow analyzers**
 - reduces the cost of developing flow analyzers to support user-defined properties, refinements, or constraints

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**

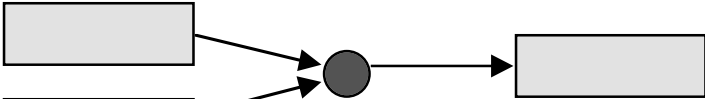
Lattice of stack heights



- **Exactly as for constant propagation**
- **What are the flow equations?**

Flow equations for stack height

- For a stack operation,
 $\text{out}(b) = f(\text{in}(b))$, where f is change in height

- For control flow join,
 $\text{out}(b) = \text{Merge}(\text{in}(b))$
where $\text{Merge}(x,x) = x$
- 
- Treating control flow join as pseudo-node ●*

$$\text{Merge}(x,y) = \{ -\text{infinity} .. \text{infinity} \} \text{ if } x \neq y$$

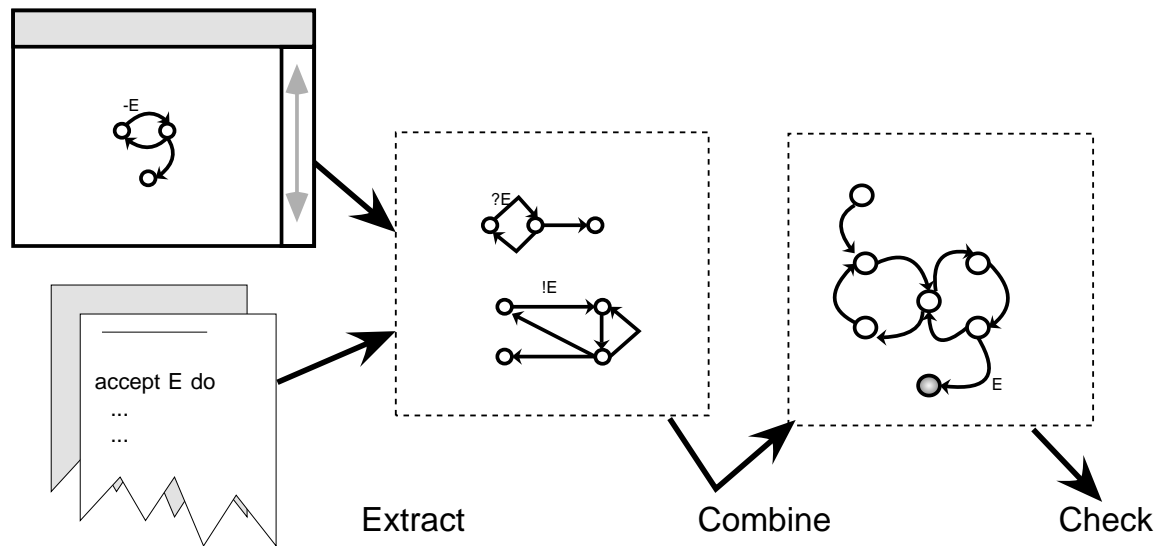
- For other operations, $\text{out}(b) = \text{in}(b)$

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**

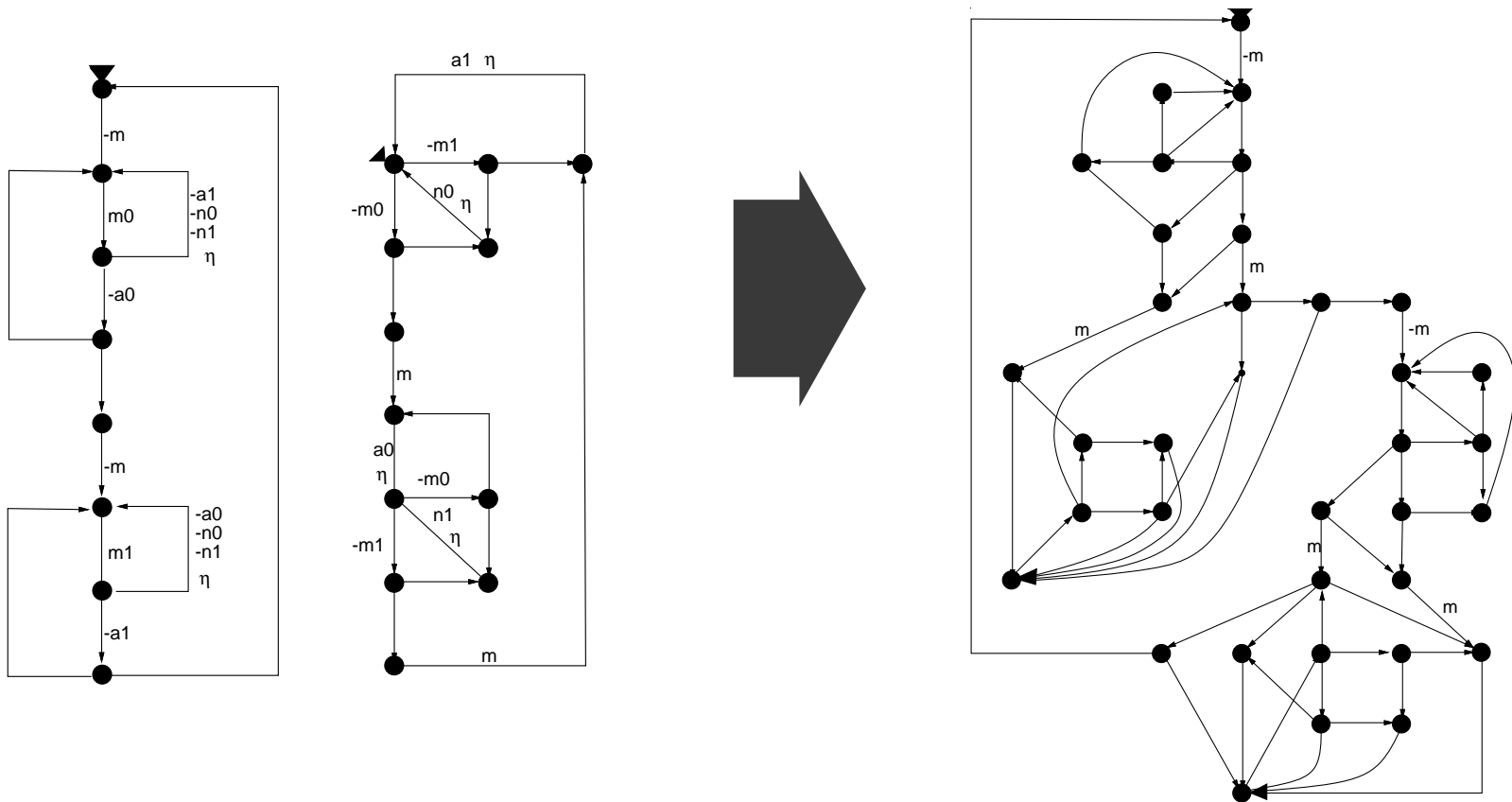


Automated Finite-State Verification

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**

State-space exploration example: Alternating Bit Protocol



State explosion problem

- **Size of composite state graph is product of individual state graphs.**
 - **OK for a simple two-party protocol, but impossibly expensive for systems with many processes**
- **Brute force state enumeration is limited to a few processes**

*State explosion is one face of a (provably) hard problem.
The same fundamental limits appear in different form for
non-enumerative analyses*

Attacks on state explosion

- **Symbolic representations**
 - **OBDDs (symbolic model checking)**
 - Very successful for hardware; little success for software
 - **Linear inequality systems (constrained expressions)**
 - Necessary conditions: occasionally imprecise
 - Appears to scale well for regularly structured systems
- **Flow analysis**
 - **The only guaranteed polynomial approach (so far)**
 - In practice, competitive with symbolic representations
- **Compositional analysis**
 - **Divide-and-conquer approach effective (only) for a class of well-structured systems**

Symbolic Model Checking

Atlee & Gannon, "State-Based Model Checking of Event-Driven System Requirements." IEEE TSE 19(1), Jan 1993.

- **The finite-state model (graph) could be very large**

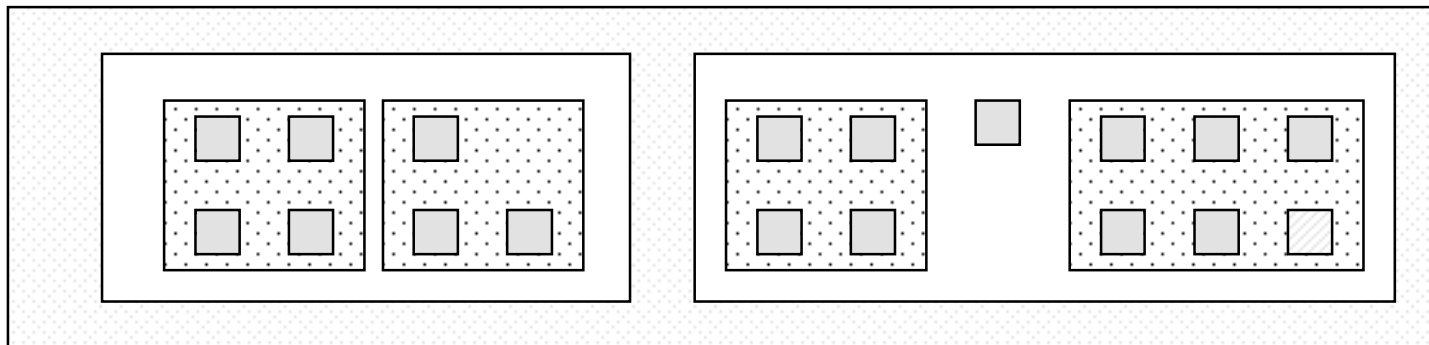
Q: Can we do better than explicitly enumerating states?

A: Not always, but sometimes symbolic representations help

- **Example tool: SMV [Clarke et al]**
 - can sometimes evaluate enormous state spaces
 - most successful for hardware verification
 - has been used to verify software requirements models

Incrementality and Compositionality

- **Incrementality:** Reanalysis must “reuse” results of previous analysis & test
 - Analyze (and/or test) changed portion of system and reuse analysis results from other portions
- **Compositionality:** System analysis is composed hierarchically from subsystems
 - Process algebra seems to be the most promising approach
- **Example tools:** fc2tools , aldeberan



Schedulability Analysis

- **Several tools apply rate-monotonic analysis to determine worst-case schedulability**
 - **Simple formulas: A spreadsheet is enough**
- **Lessons:**
 - **Simplify the problem ~ constrain the domain**
 - **RMA analysis applicable (only) when applications obey strict structuring criteria (e.g., priority ceiling protocol)**
 - **Focus static analysis on what is hard to test**
 - **Especially rare occurrences, like worst-case response**

CATS/Pal: *Concurrency Analysis with Process Algebra*

Yeh & Young, “Compositional Reachability Analysis Using Process Algebra.” Proc. TAV4, Oct 1991.

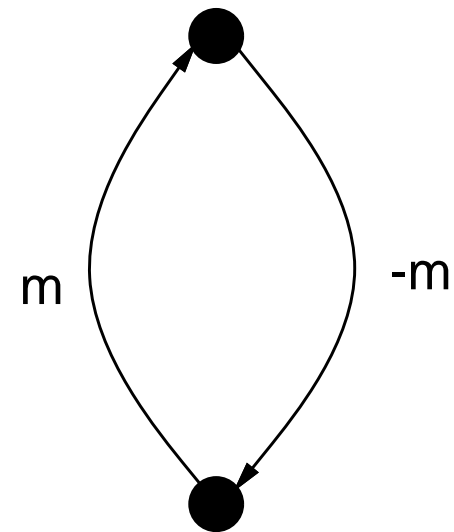
- **Reasons about concurrent structure of a software system**
 - e.g., well-structured, layered designs in Ada-PDL
- **Detects and diagnoses defects in task coordination**
 - potential deadlocks, race conditions
 - particularly difficult for designers and integrators
 - non-deterministic and time-sensitive, which can escape testing and appear in rare field conditions
- **Modular, efficient approach**
 - hierarchical state-space analysis
 - process algebra used to reduce billions of behaviors to a manageable number

Process Algebra

- **Processes (state machines) are “terms” in an algebra, with a congruence based on observable behavior**
 - CCS (Milner), CSP (Hoare), ACP (Bergstra & Klop), ...
- **Parallel composition is associative and commutative, i.e., $a || (b || c) = b || (a || c)$**
- **Events can be “hidden” to model modularity**
 - Restriction and abstraction distribute through composition (under suitable conditions)
- ***Provides basis for a compositional approach***

Alternating Bit Protocol: After reduction

- After restriction and abstraction, process graphs can be reduced to equivalent form with respect to a congruence relation (observation equivalence, testing equivalence, ...)



... but radical reductions in process graph size occur only when the system to be analyzed is “well-structured”

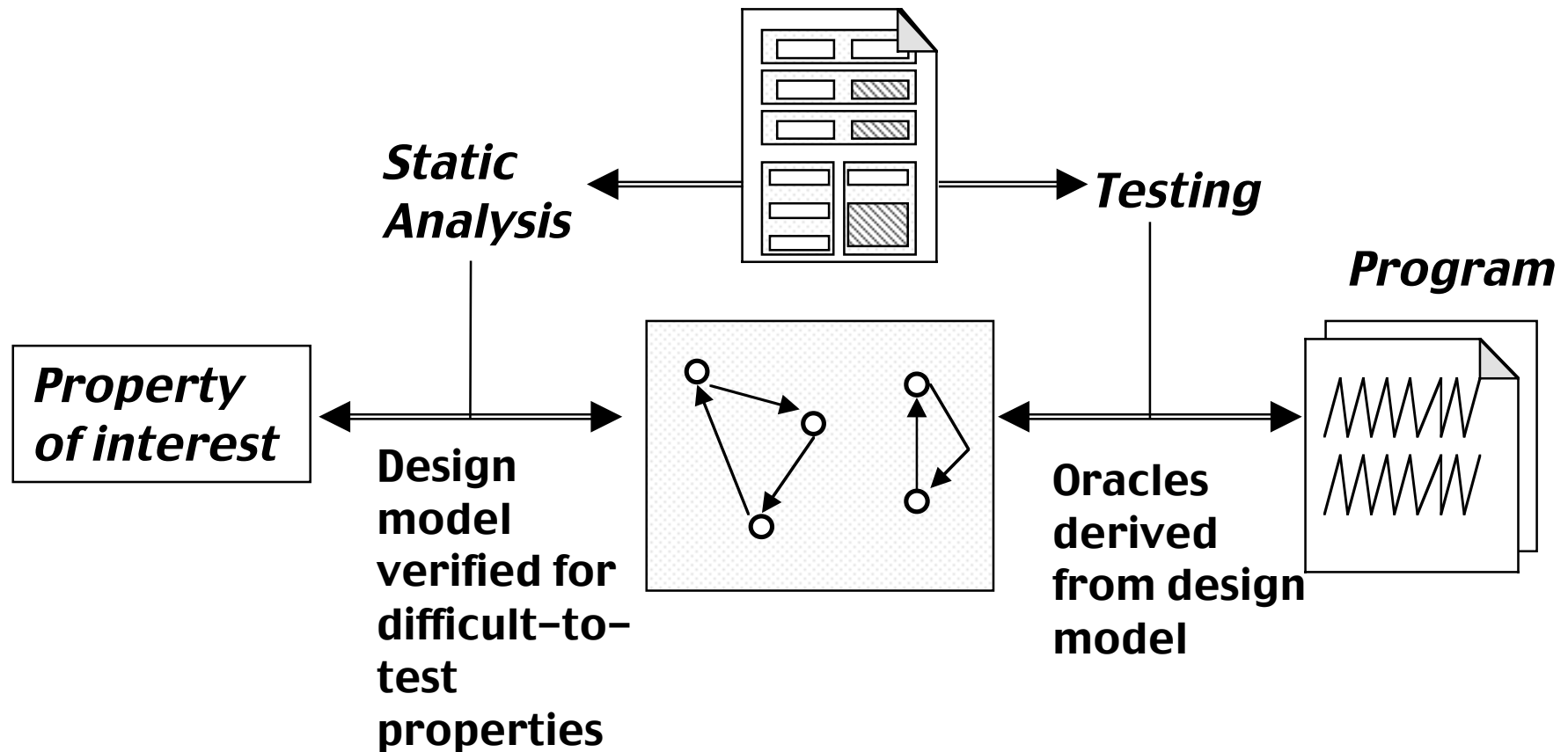
What is static analysis good for?

- **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**

Prospects

- **Some static analyses are usable now**
 - Rate-monotonic schedule analysis
 - Java byte-code verification
- **Others are nearing readiness**
 - Finite-state verification: Usable by experts
 - Data flow analysis: Mostly “roll your own”
- **Sometimes simple tools are worth building**
 - Enforcing and/or assuming organization standards to simplify analysis

Combining Analysis and Test



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