Model Checking
&
Program Analysis

Markus Müller-Olm
Dortmund University

Overview

- Introduction
- Model Checking
- Flow Analysis
- Some Links between MC and FA
- Conclusion

Apology for not giving proper credit to other researchers' work in this tutorial !

Purposes of Automatic Analysis

- Optimizing compilation
- Validation/Verification
  - Type checking
  - Functional correctness
  - Security properties
  - . . .
- Debugging

Fundamental Limit

Rice's Theorem [Rice,1953]:

> All interesting semantic questions about programs from a universal programming language are undecidable.

Example: Detection of Constants

read(new);
π;
write(new)

?

read(new);
π;
write(k)

write(new) can be replaced by write(k) for some constant k

$\Leftrightarrow$

π terminates

Hence: Constant Detection is undecidable

Two Solutions

Weaker formalisms

- analyze abstract models of systems
- e.g.: automata, labelled transition systems,...

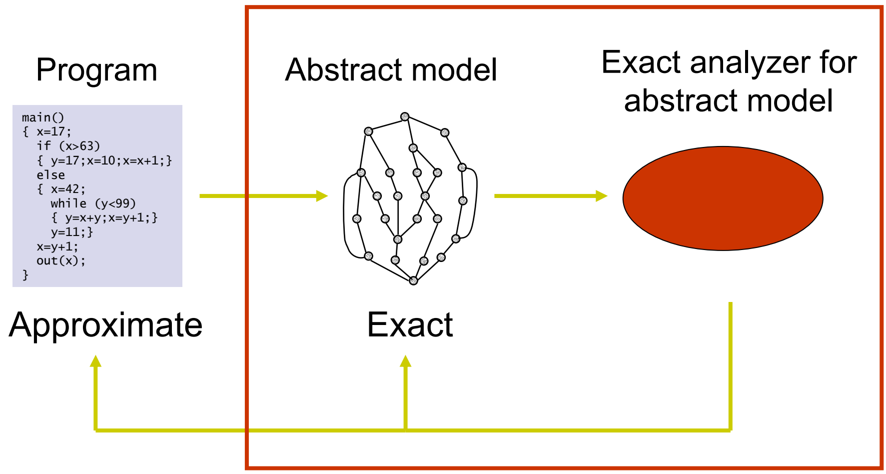
Approximate analyses

- yield sound but, in general, incomplete results
- e.g.: detects some instead of all constants

Model checking

Flow analysis
Abstract interpretation
Type checking

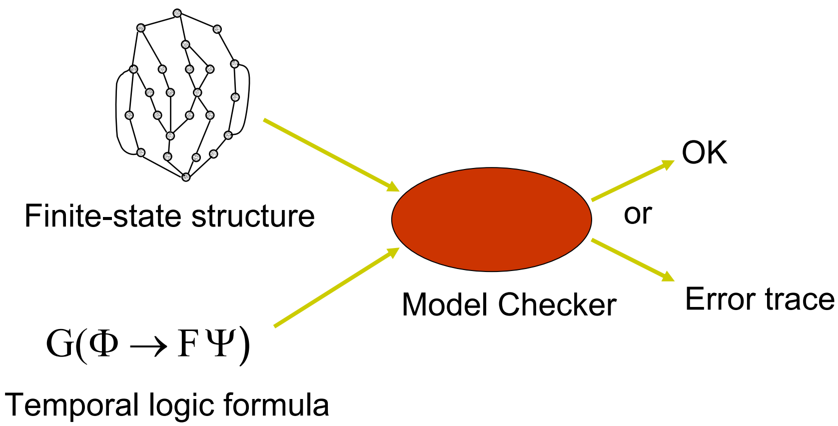
Weaker Formalisms



Overview

- Introduction
- **Model Checking**
- Flow Analysis
- Some Links between MC and FA
- Conclusion

Model Checking



What is a Model Checker?

an automatic procedure for deciding

$$M \models \phi$$

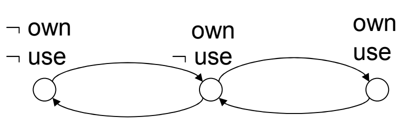
where

- M is a model structure
- ϕ is a (temporal-logic) formula
- $\models$ means satisfaction

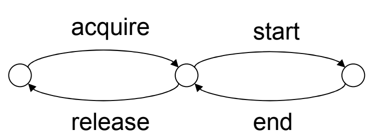
Model Structures

$$M \models \phi$$

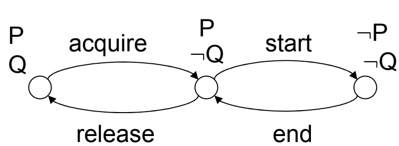
Model Structures



Kripke structure

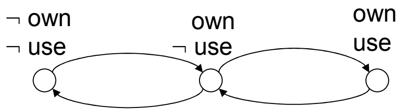


Labeled Transition System



Kripke Transition System

Kripke Structures



$K = (S, R, I)$, where

S	states
$R \subseteq S \times S$	transition relation, total
$I : S \rightarrow 2^{AP}$	interpretation
AP	atomic propositions

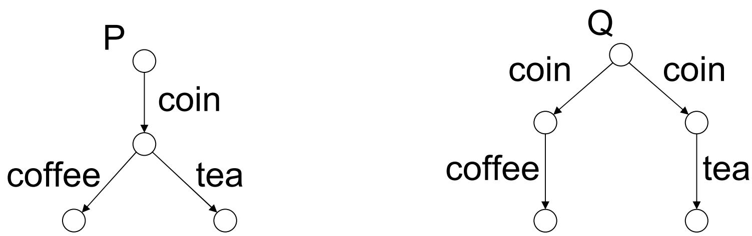
Temporal Logics

$$M \models \phi$$

Temporal Logics

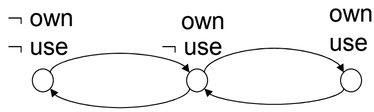
- Linear-Time Logics
  - formulas specify properties of program paths
  - state satisfies property if all paths starting in this state do
- Branching-Time Logic
  - can specify properties sensitive to branching
  - has (or can simulate) path quantifiers A and E

Branching vs. Linear-Time Logics



- LT logics cannot distinguish P and Q:
  - P and Q have same execution paths
- BT logics can:
  - P $\models$ [coin] <tea> true but Q $\not\models$ [coin] <tea> true

A Linear-Time Logic: PLTL



1. own the resource before using it

$\neg use\ U\ own$

2. release the resource in finite time

$own \Rightarrow F(\neg own)$

PLTL formulas

$$\phi ::= p \,|\, \neg\phi \,|\, \phi_1 \vee \phi_2 \,|\, X\phi \,|\, \phi_1 U \phi_2$$

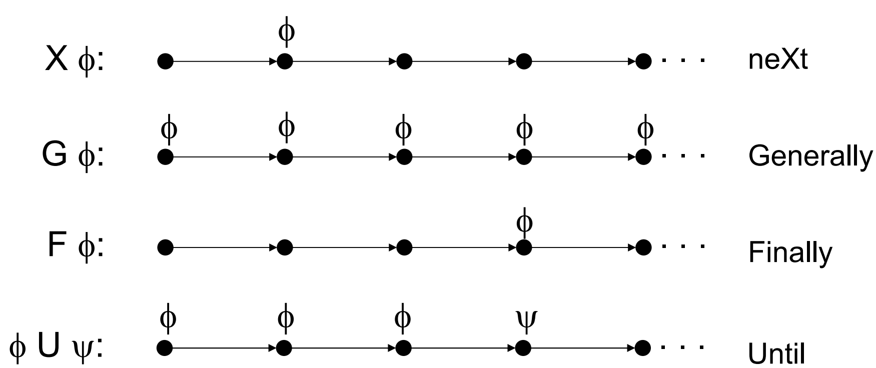
Abbreviations

$$F\,\phi := true\,U\,\phi$$

$$G\,\phi := \neg F(\neg\phi)$$

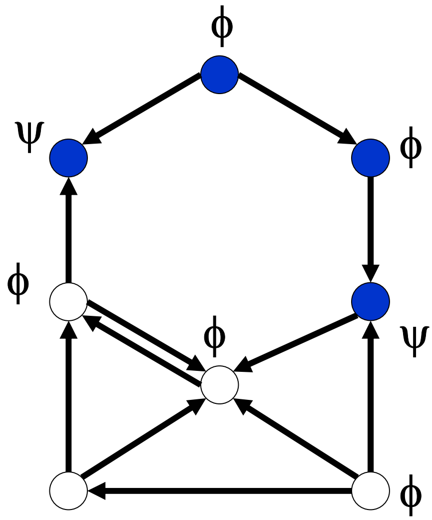
$$\phi\, WU\, \psi := \phi\, U\, \psi \vee G\,\phi$$

Linear-Time Modalities



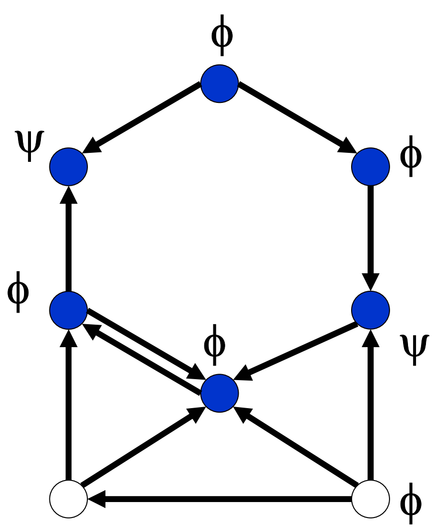
ϕ WU ψ: ϕ U ψ or G ϕ Weak Until

Until



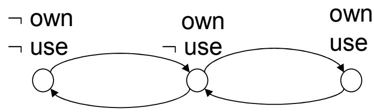
- Linear-time: blue nodes satisfy
  $v \models \phi \; U \; \psi$
- Branching-time: blue nodes satisfy
  $v \models A(\phi \; U \; \psi)$

Weak Until



- Linear-time: blue nodes satisfy
  $v \models \phi \; WU \; \psi$
- Branching-time: blue nodes satisfy
  $v \models A \, (\phi \; WU \; \psi)$

A Branching-Time Logic: CTL



1. own the resource before using it

   $A(\neg use\ U\ own)$

2. release the resource in finite time

   $own \Rightarrow AF(\neg own)$

State formulas ϕ

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid E\psi \mid A\psi$$

Path formulas ψ

$$\psi ::= X\,\phi \mid \phi_1\,U\,\phi_2 \mid G\,\phi \mid \mathrm{F}\,\phi$$

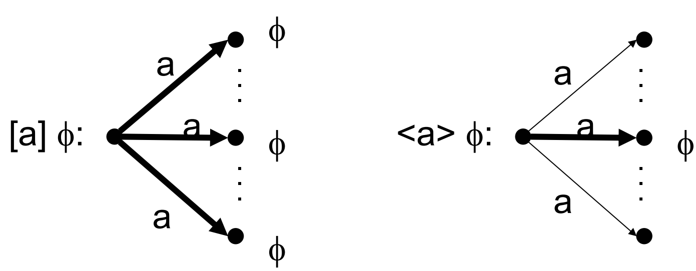
Duality

- Path quantifiers are duals:
  - $\neg A\ \phi = E \neg\phi$
  - $\neg E\ \phi = A \neg\phi$
- Until and weak until are almost duals (on paths):
  - $\neg(\phi\ U\ \psi) = \neg\psi\ WU\ (\neg\phi \wedge \neg\psi)$
  - $\neg(\phi\ WU\ \psi) = \neg\psi\ U\ (\neg\phi \wedge \neg\psi)$

Modal mu-Calculus

- a branching-time logic with **local** modalities…
  - $\Box\,\phi$: **all** successor states satisfy ϕ
  - $\Diamond\,\phi$: **some** successor state satisfies ϕ
- … and fixpoint formulae.
  - μ X. ϕ(X) : minimum fixpoint formula
  - ν X. ϕ(X) : maximum fixpoint formula
- Fixpoint formulae can be nested
  - Alternation: $\mu X.(\nu Y.X \wedge Y)$

Local Modalities for Labelled Edges



Computing Fixpoint Formulae

- On finite structures, meaning of fixpoint formulae can be computed by computing Kleene chains until stabilization:
  - μ : $\bot$,f($\bot$),f(f($\bot$),f(f(f($\bot$))), . . .
  - ν : $\top$,f($\top$),f(f($\top$)),f(f(f($\top$))), . . .

  f is a function on state sets derived from the body of the fixpoint formula.

CTL and Modal mu-Calculus

- CTL-formulae can inductively be transformed to modal mu-calculus formulae
- Global modalities can be expressed by fixpoint formulae, e.g.:

$$A(\phi\ U\ \psi) = \mu\mathrm{X}.(\psi \vee (\phi \wedge \Box \mathrm{X} \wedge \Diamond\, \mathrm{true}))$$
$$E(\phi\ U\ \psi) = \mu\mathrm{X}.(\psi \vee (\phi \wedge \Diamond\, \mathrm{X}))$$

$$A(\phi\ WU\ \psi) = \nu\mathrm{X}.(\psi \vee (\phi \wedge \Box \mathrm{X}))$$
$$E(\phi\ WU\ \psi) = \nu\mathrm{X}.(\psi \vee (\phi \wedge (\Diamond\, \mathrm{X} \vee \Box\, \mathrm{false})))$$

Model Checking Approaches

$$M \models \phi$$

Global vs. Local Model Checking

- Global model checking problem
  - Given: finite model structure M, formula ϕ
  - Determine: $\{s \mid s \models \phi\}$
- Local model checking problem
  - Given: finite model structure M, formula ϕ, and state s in M
  - Determine, whether $s \models \phi$ or not

Model Checking Approaches

- Iterative model checking
- Automata-theoretic model checking
- Tableau-based model checking

Iterative Model Checking

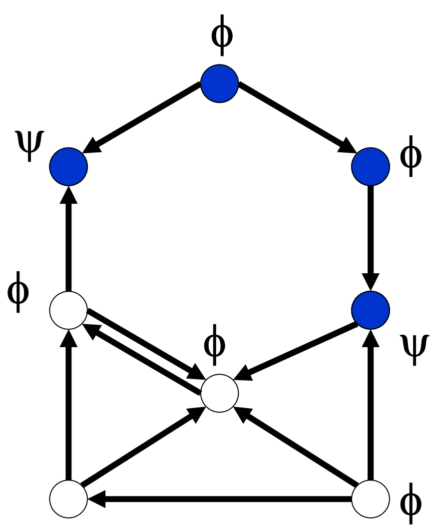
- Good for global model-checking of branching-time logics
- Idea: Compute semantics of formula on given model by structural induction on the formula
  - Reduce modalities to their fixpoint definition
  - Compute the fixpoints by Kleene chains
  - Alternating fixpoints lead to backtracking

(→ proceedings)

Iterative Model Checking of CTL

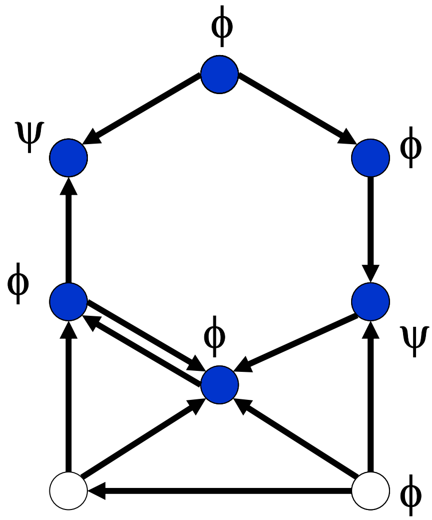
- Annotate each state with those subformulas ψ of ϕ with $s \models \psi$.
- Use structural induction on ϕ.
- Use backwards propagation to compute modalities $A(\phi\ U\ \psi)$ and $A(\phi\ WU\ \psi)$.

Model Checking $A(\phi U \psi)$



1. Mark all nodes v with $v \models \psi$
2. Mark all unmarked nodes w with $w \models \phi$ and **all** successors marked
3. Iterate 2. until stabilization

Model Checking A(ϕWUψ)



1. Mark all nodes with $v \models \phi$ or $v \models \psi$
2. Unmark all nodes v with $v \not\models \psi$ and some unmarked successor
3. Iterate 2. until stabilization

Automata-theoretic MC

- Good for linear-time logics
- Idea: reduce model-checking to non-emptiness problem of an automaton

Automata-theoretic MC

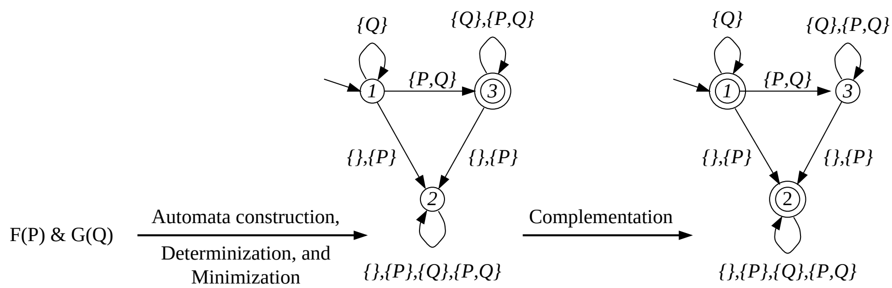
- Construct (Büchi-) automaton, A_ϕ, from ϕ
  - A_ϕ accepts paths satisfying ϕ
- Construct (Büchi-) automaton, A_M, from M
  - A_M accepts paths exhibited by M

$$\begin{aligned} M \models \phi \quad & \text{iff} \quad L(A_M) \subseteq L(A_\phi) \\ & \text{iff} \quad L(A_M) \cap \overline{L(A_\phi)} = \varnothing \\ & \text{iff} \quad L(A_M \times \overline{A_\phi}) = \varnothing \end{aligned}$$

Automata-theoretic MC

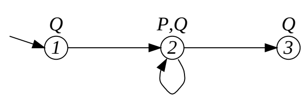
- Construct (Büchi-) automaton, A_ϕ, from ϕ
  - A_ϕ accepts paths satisfying ϕ
- Construct (Büchi-) automaton, A_M, from M
  - A_M accepts paths exhibited by M
- Compute automaton $A_M \times \overline{A}_\phi$
  - Complementation & product construction
- Decide $L(A_M \times \overline{A}_\phi) = \varnothing$ by reachability analysis

Automaton for F(P)∧G(Q)
(Finite Maximal Paths Interpretation)

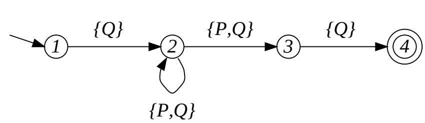


A Successful Model-Check for formula F(P)∧G(Q)

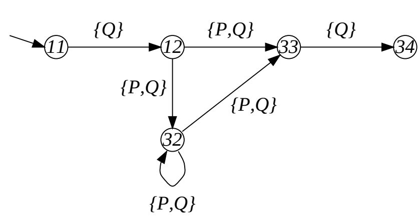
Kripke structure:



Corresponding automaton:

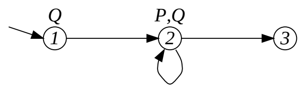


Product automaton:

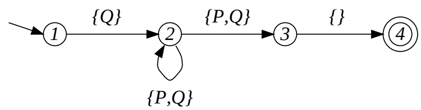


A Failing Model-Check for formula $F(P) \wedge G(Q)$

Kripke structure:



Corresponding automaton:



Product automaton:

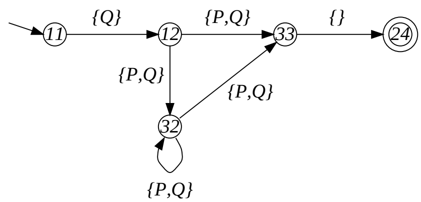


Tableau-Based Model-Checking

- Idea: solve local model checking problem by sub-goaling
  - Try to construct a proof tree that witnesses $s \models \phi$
  - If no proof tree can be found, then $s \not\models \phi$
  - For proof tree construction, tableau rules are given that reduce goals to sub-goals

Some Tableau Rules

AND $\dfrac{s \vdash \phi_1 \wedge \phi_2}{s \vdash \phi_1 \qquad s \vdash \phi_2}$

OR1 $\dfrac{s \vdash \phi_1 \vee \phi_2}{s \vdash \phi_1}$ OR2 $\dfrac{s \vdash \phi_1 \vee \phi_2}{s \vdash \phi_2}$

BOX $\dfrac{s \vdash [\mathrm{a}]\phi}{s_1 \vdash \phi, \ldots, s_n \vdash \phi}$ if $\{s_1, \ldots, s_n\} = \{t \mid s \xrightarrow{a} t\}$

DIAMOND $\dfrac{s \vdash \langle \mathrm{a} \rangle \phi}{t \vdash \phi}$ if $s \xrightarrow{a} t$

Tableau Rules for Fixpoints

- Fixpoint formulas are analyzed with unfolding rules, e.g.,

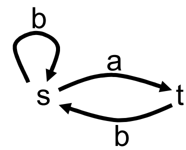
$$\mu\text{-UNFOLD} \quad \frac{s \vdash \mu X.\phi(X)}{s \vdash \phi(\mu X.\phi(X))}$$

- If fixpoint formula sequent re-appears later:
  - μ : unsuccessful leaf
  - ν : successful leaf
- Special care needed for nested fixpoints ($\rightarrow$ proceedings)

A Model Check by Tableau

$$\phi = \nu X.(\psi \wedge [b]X)$$

$$\psi = \mu X.\langle a \rangle \mathrm{true} \vee \langle b \rangle X$$



$$s \vdash \phi$$

$$s \vdash \psi \wedge [b]\phi$$

$$s \vdash \psi \qquad\qquad s \vdash [b]\phi$$

$$s \vdash \langle a \rangle \mathrm{true} \vee \langle b \rangle \psi \qquad\qquad s \vdash \phi$$

$$s \vdash \langle a \rangle \mathrm{true}$$

$$t \vdash \mathrm{true} \ \checkmark$$

ν ⇒ successful !

Typical Profile of Model Checking Techniques

	Branching-time	Linear-time	Global	Local
Iterative	X		X	
Automata-theoretic		X	X	X
Tableau methods	X	X		X

State-Explosion Problem

- State-Explosion
  - number of states grows exponentially in number of components or variables
- Techniques for fighting state-explosion
  - symbolic model-checking
  - incremental construction of state space
  - abstraction
  - symmetry reductions
  - partial-order methods
  - compositional methods
  - …

Other Classes of Systems

- Infinite-state systems
- Timed systems
- Hybrid systems
- Probabilistic systems
- . . .

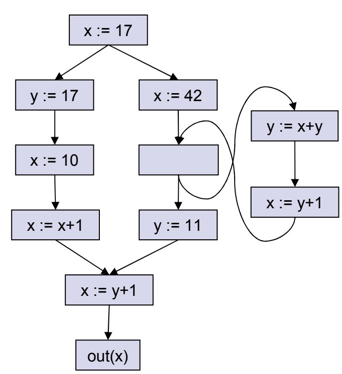
⇨ Learn more about such systems this week !

Overview

- Introduction
- Model Checking
- **Flow Analysis**
- Some Links between MC and FA
- Conclusion

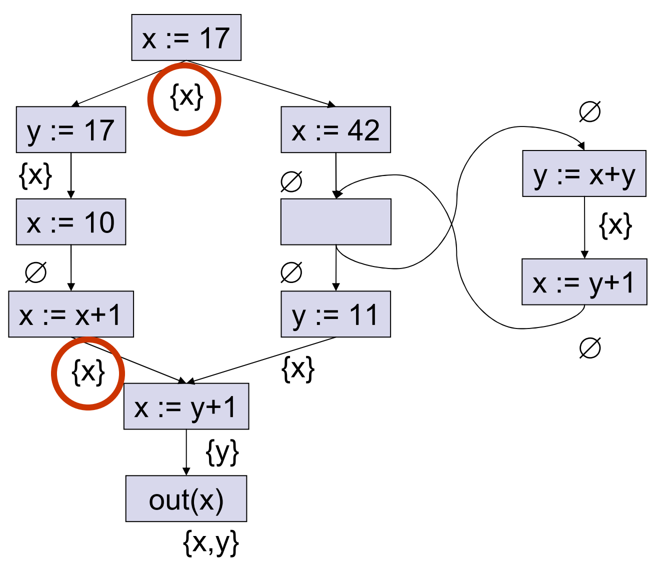
From Programs to Flow Graphs

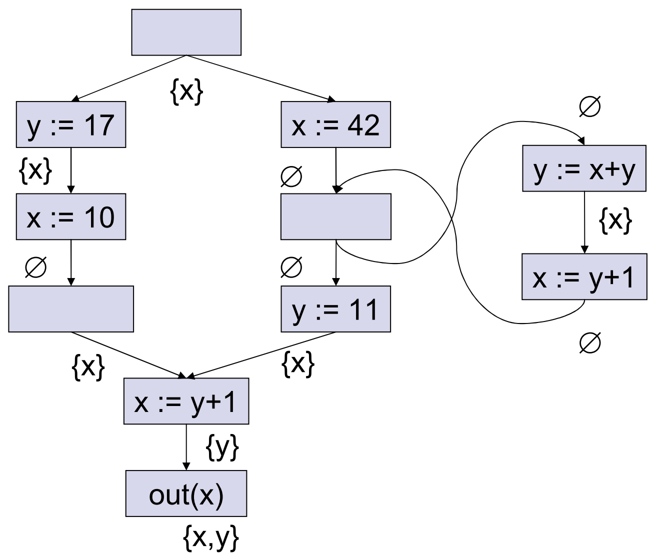
```
main()
{ x=17;
  if (x>63)
  { y=17;x=10;x=x+1;}
  else
  { x=42;
    while (y<99)
    { y=x+y;x=y+1;}
    y=11;}
  x=y+1;
  out(x);
}
```



Dead Code Elimination

- Goal:
  - find and eliminate assignments that compute values which are never used
- Fundamental problem:
  - undecidability
  - hence: use approximate algorithm; ignore guards
- Technique:
  - propagate set of non-needed variables backwards through the flow graph

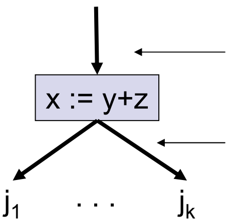




Remarks

- Forward vs. backward analyses
- Bitvector analyses
  - backward: live/dead variables, very busy expressions,
  - forward: reaching definitions, available expressions
- Computation strategies

Flow Equations for Dead Code



$$\text{DeadIn}[i] = (\text{DeadOut}[i] \cup \{x\} \setminus \{y, z\}$$

$$\text{DeadOut}[i] = \bigcap_{j \in \text{Succ}[i]} \text{DeadIn}[i]$$

General equations:

$$\text{DeadIn}[i] = (\text{DeadOut}[i] \cup \text{Mod}[i]) \setminus \text{Use}[i]$$

$$\text{DeadOut}[i] = \bigcap_{j \in \text{Succ}[i]} \text{DeadIn}[i]$$

Equations may be combined…

$$\text{DeadOut}[i] = \bigcap_{j \in \text{Succ}[i]} (\text{DeadOut}[j] \cup \text{Mod}[j]) \setminus \text{Use}[j]$$

…or replaced by inequations:

$$\text{DeadOut}[i] \subseteq (\text{DeadOut}[j] \cup \text{Mod}[j]) \setminus \text{Use}[j]\ , \quad \text{for } j \in \text{Succ}[i]$$

Data-Flow Frameworks

- Correctness
  - generic properties of frameworks can be studied and proved
- Implementation
  - efficient, generic implementations can be constructed

Data-Flow Frameworks

- a complete lattice $(D, \sqsubseteq)$ of data-flow facts
- a space F of transfer functions f: D $\rightarrow$ D
  - Id$\in$F
  - closed under composition
- initial value init$\in$D
- a control-flow graph with set of entry/exit nodes
- mapping assigning transfer functions f_i to flow graph nodes i

Framework for dead variables

lattice D	2^{Var}
$\sqsubseteq$	$\subseteq$
$\sqcap$	$\cap$
$\top$	Var
control flow	program graph
initial value	$\varnothing$
function space	$\{f : D \rightarrow D \mid \exists d_1, d_2 : f(d) = (d \cup d_1) \setminus d_2\}$
f_i	$f_i(d) = (d \cup \mathrm{Mod}[i]) \setminus \mathrm{Use}[i]$

What Data Flow Algorithms Compute

- Forward Analysis
  - Computes maximal solution w.r.t. (D,$\sqsubseteq$) of

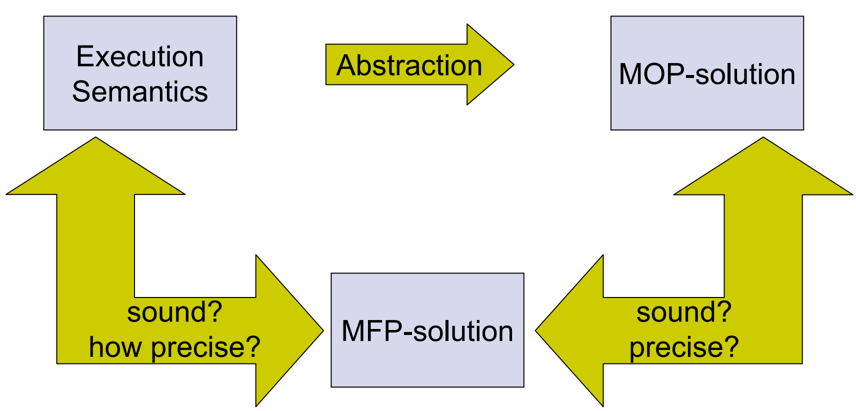
$$\mathrm{In}[i] = \begin{cases} init & i \in \mathrm{Entry} \\ \sqcap_{j \in \mathrm{Pred}(i)} f_j(\mathrm{In}[j]) & \text{otherwise} \end{cases}$$

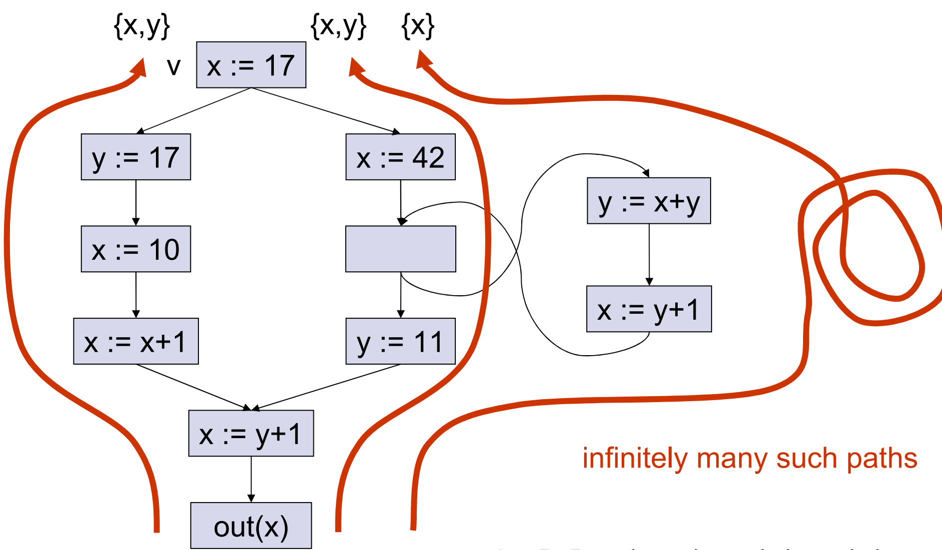
- Backward Analysis
  - Computes maximal solution of

$$\mathrm{Out}[i] = \begin{cases} init & i \in \mathrm{Exit} \\ \sqcap_{j \in \mathrm{Succ}(i)} f_j(\mathrm{Out}[j]) & \text{otherwise} \end{cases}$$

- This is called the maximal fixpoint solution MFP[i]

Assessing Data Flow Frameworks





$$\mathrm{MOP}[v] = \{x, y\} \cap \{x\} = \{x\}$$

Meet-Over-All-Paths Solution

- Forward Analysis

$$\mathrm{MOP}[i] := \bigsqcap_{p \in \mathrm{Paths}[entry, i)} \mathrm{F}_p(init)$$

- Backward Analysis

$$\mathrm{MOP}[i] := \bigsqcap_{p \in \mathrm{Paths}(i, exit]} \mathrm{F}_p(init)$$

Coincidence Theorem

Definition:

A framework is distributive if
$f(\sqcap X) = \sqcap\{f(x) | x \in X\}$ for all $X \subseteq D$, $f \in F$.

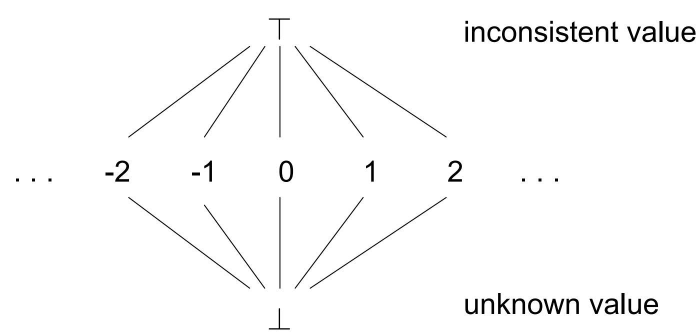
Theorem:

In any distributive framework,
MOP[i]=MFP[i] for all program points i.

Theorem:

All bitvector frameworks are distributive.

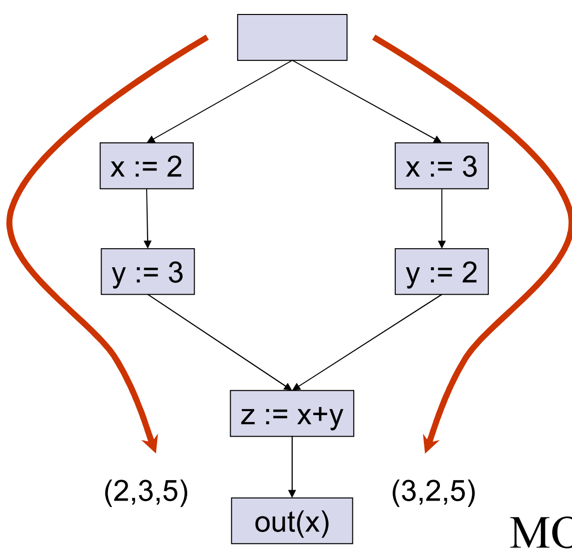
Lattice for Constant Propagation



Constant Propagation Framework

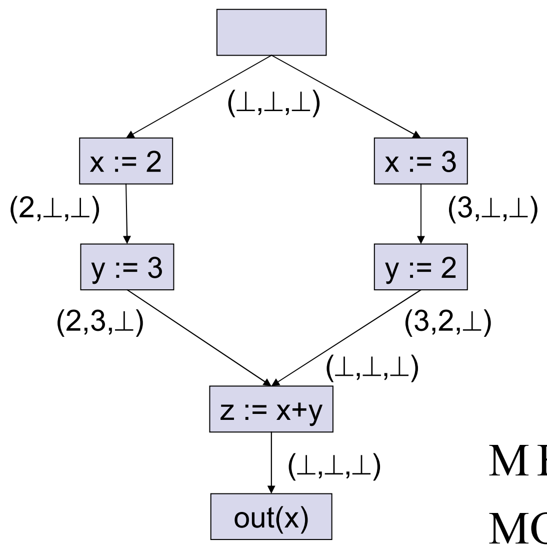
lattice D	$\mathrm{Var} \to (\mathbb{Z} \cup \{\bot, \top\}) \;=\; \mathrm{Var} \to \mathrm{ConstVal}$
$\sqsubseteq$	$\rho \sqsubseteq \rho' :\Leftrightarrow \forall x : \rho(x) \sqsubseteq \rho'(x)$
$\sqcap$	pointwise meet
$\top$	$\top(x) = \top$ f.a. $x \in \mathrm{Var}$
control flow	program graph
initial value	$\bot$
function space	$\{f : D \to D \mid f \text{ monotone}\}$
f_i	$f_i(d) = \begin{cases} d[x \mapsto [\![e]\!]^{CP}(d)] & \text{if } i \text{ annotated with } x := e \\ d & \text{otherwise} \end{cases}$

$(\rho(x), \rho(y), \rho(z))$



$\mathrm{MOP}[v] = (\bot, \bot, 5)$

$(\rho(x), \rho(y), \rho(z))$



$$MFP[v] = (\bot, \bot, \bot)$$

$$MOP[v] = (\bot, \bot, 5)$$

Correctness Theorem

Definition:

A framework is monotone if $f(x) \sqsubseteq f(y)$ for all $x, y \in D$, $x \sqsubseteq y$.

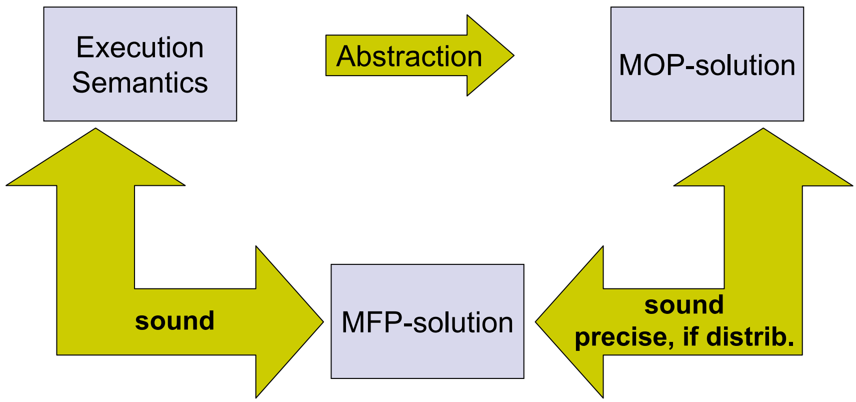
Theorem:

In any monotone framework,
$MFP[i] \sqsubseteq MOP[i]$ for all program points i.

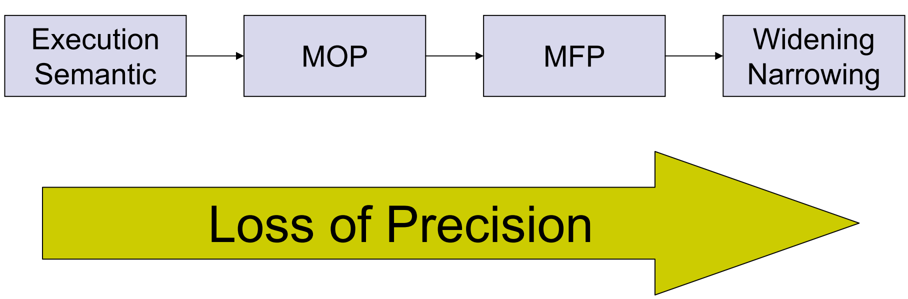
Remark:

Any "reasonable" framework is monotone.

Assessing Data Flow Frameworks



Where Flow Analysis Looses Precision



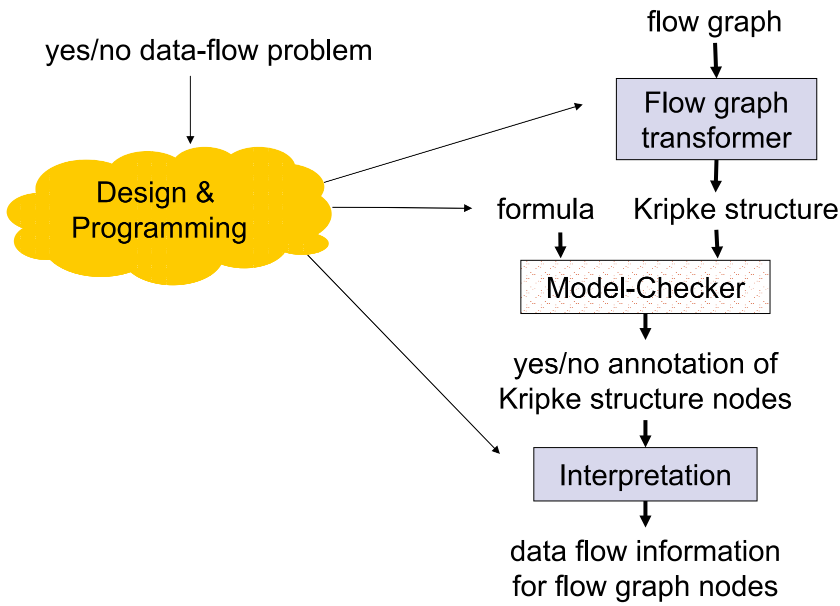
Overview

- Introduction
- Model Checking
- Flow Analysis
- **Some Links between MC and FA**
- Conclusion

Some Links between MC and FA

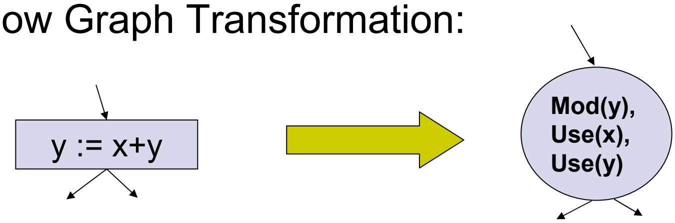
- Flow Analysis via Model Checking
- Model Checking via Flow Analysis
- Synergy of MC and FA

Flow Analysis via Model Checking



Dead-Code Elimination

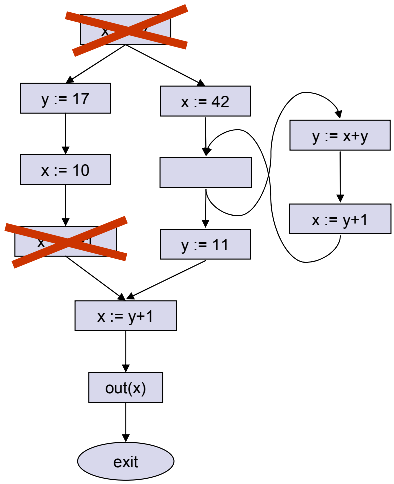
- Flow Graph Transformation:

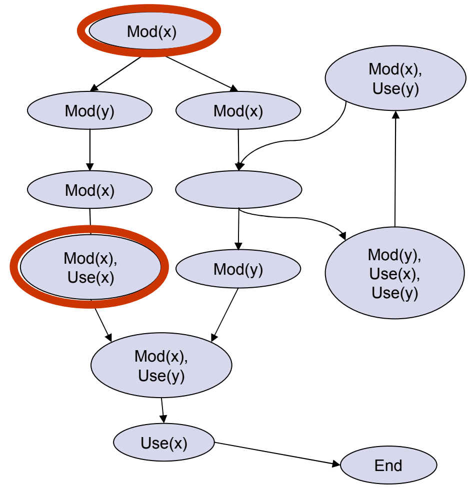


- Formula specifying "x is dead":

$$\text{Dead}_x = \text{X}\ (\neg \text{use}(x)\ \text{WU}\ (\text{mod}(x) \wedge \neg \text{use}(x)))$$

- Note: more direct specification than in the data flow framework





Check: $\text{Dead}_{x} = X\ (\neg \text{use}(x)\ \text{WU}\ (\text{mod}(x) \wedge \neg \text{use}(x)))$

Model Checking via Flow Analysis

- Evaluation of (CTL) modalities can be seen as a data flow analysis
- CTL model-checking is iterated flow analysis

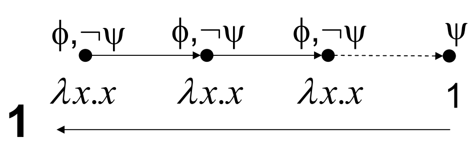
Framework for E(ϕ U ψ)

0
|
1

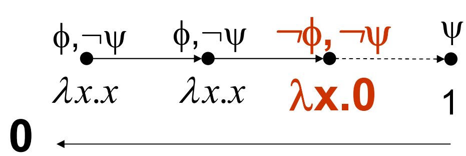
lattice D	$\mathbb{B} = \{0,1\}$
$\sqsubseteq$	$1 \sqsubseteq 0$
$\sqcap$	$0 \sqcap 1 = 1$
$\top$	0
control flow	Kripke structure (S, R, I)
initial nodes	Nodes s with $s \vDash \psi$
initial value	1
function space	$\{\lambda x.0, \lambda x.1, \lambda x.x\}$
f_i	$f_i(d) = \begin{cases} \lambda x.x & \text{if } i \nvDash \psi, i \vDash \phi \\ \lambda x.0 & \text{if } i \nvDash \psi, i \nvDash \phi \end{cases}$

Why the Framework computes E(ϕ U ψ)

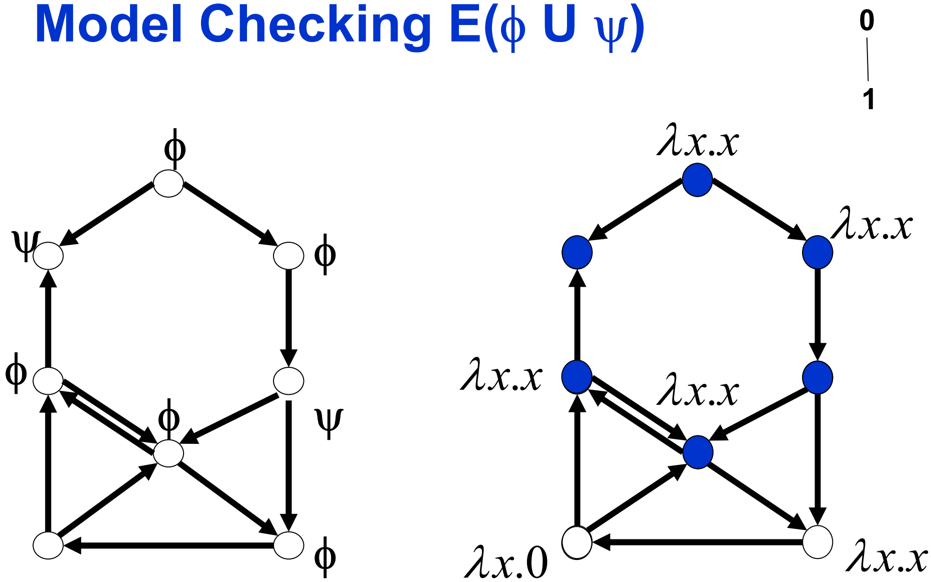
$$\mathrm{MFP}[v] = \mathrm{MOP}[v] = 1 \quad \text{iff} \quad s \vDash \mathrm{E}(\phi \mathrm{U} \psi)$$



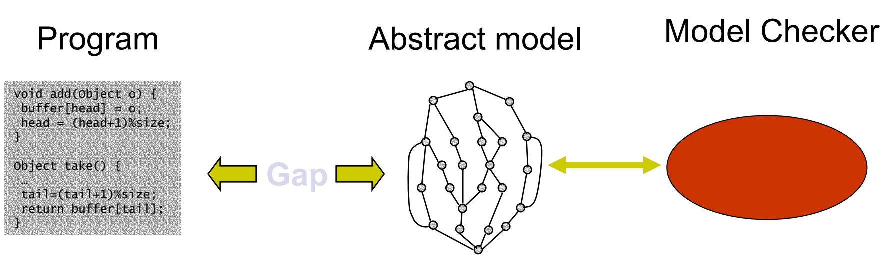
0
|
1



Model Checking E(ϕ U ψ)



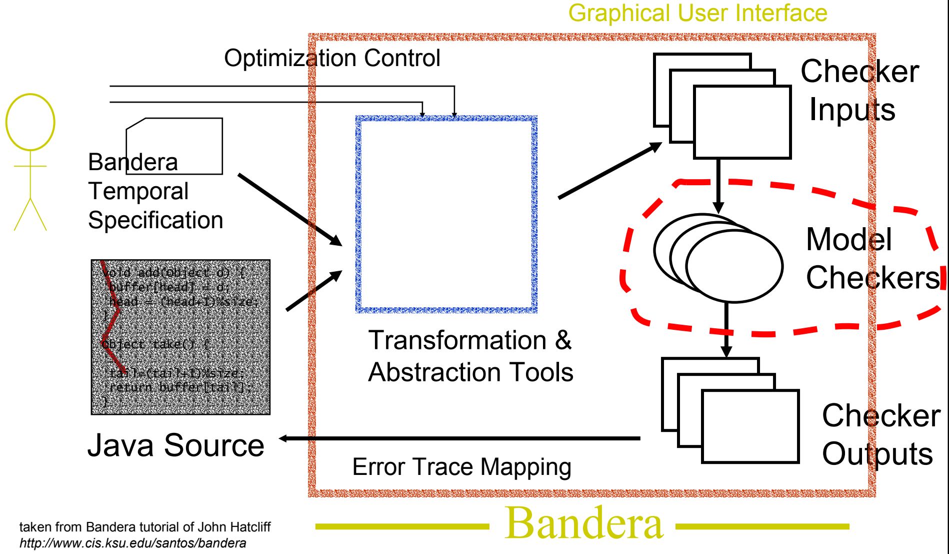
The Model-Extraction Problem in Software Model-Checking



Idea: Use techniques from abstract interpretation and program analysis to extract a finite-state model from program

Bandera: [Dwyer, Hatcliff, et. al.]

An open tool set for model-checking Java source code



taken from Bandera tutorial of John Hatcliff
http://www.cis.ksu.edu/santos/bandera

Conclusion

- Overview on fundamentals of
  - Model Checking
  - Flow Analysis
  - Some links

Just the Beginning. . .

- More complex properties
- Theory of Abstract Interpretation
- Interprocedural Flow Analysis
- Parallel Programs

Any Questions?