Outline

- **The Naive Approach**
  - “using SMV to model-check software”
- **Abstraction and Model-Checking**
- **Automatic Abstraction Refinement**
- **Yasm: Tool Demo**
- **Conclusion**

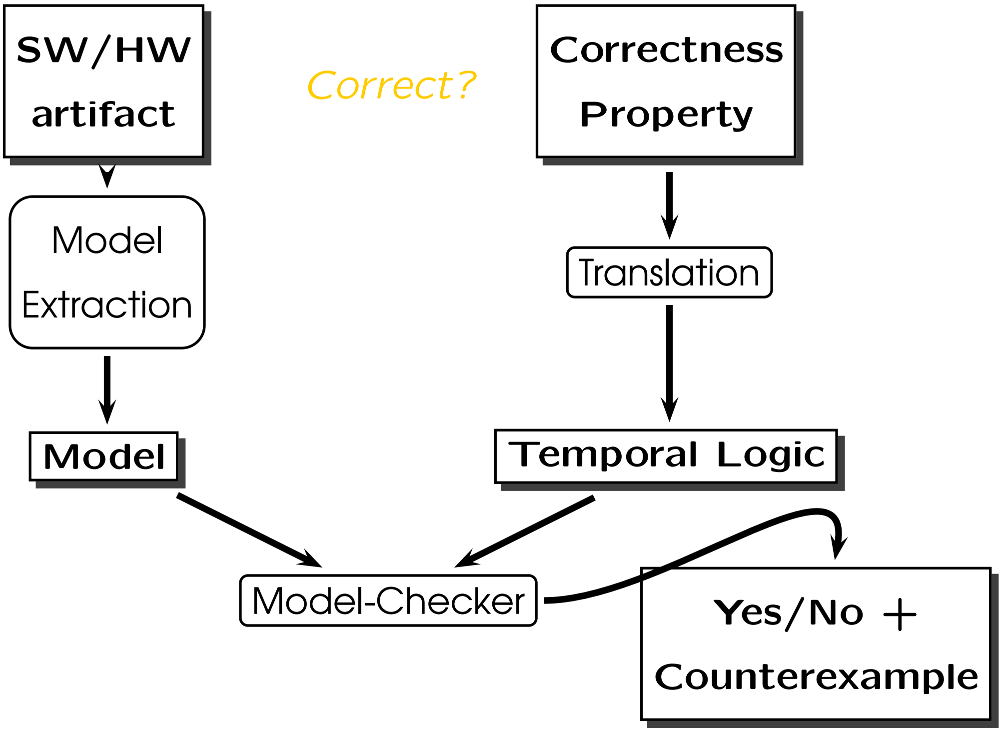
Software Model-Checking with YASM: A Tutorial

Arie Gurfinkel

{arie}@cs.toronto.edu.

University of Toronto

Overview of Model-Checking



SW/HW artifact

Correct?

Correctness Property

Model Extraction

Translation

Model

Temporal Logic

Model-Checker

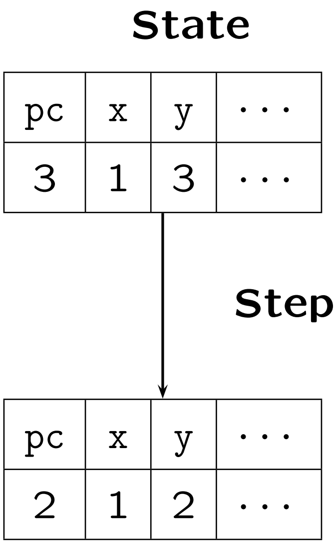
Yes/No + Counterexample

SoftMC: The Naive Approach

- **Input:**
  - Program:
    ```
    void main (void) {
    1: int x = 2;
       int y = 2;
    2: while (y <= 2)
    3:    {y = y - 1;}
    4: if (x == 2)
    5:    {P1:}
    6:}
    ```
  - Property: is line `P1` reachable?
- **Output: Yes/No**

From Programs to Models

```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:    {y = y - 1;}
4: if (x == 2)
5:    {P1:}
6:}
```



State

pc	x	y	...
3	1	3	...

Step

pc	x	y	...
2	1	2	...

From Programs to Models

```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:    {y = y - 1;}
4: if (x == 2)
5:    {P1:}
6:}
```

State

pc	x	y	...
3	1	3	...

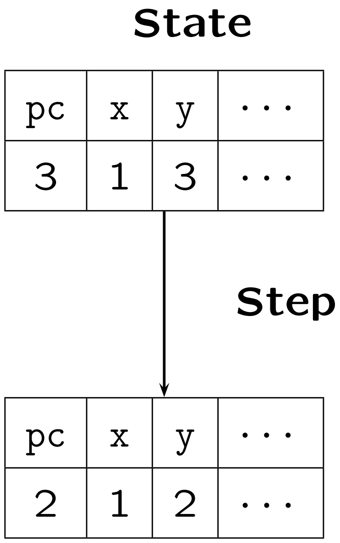
Representing Transition Relation

```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   {y = y - 1;}
4: if (x == 2)
5:   {P1:}
6:}
```

$$pc = 1 \land x' = 2 \land y' = 2 \land pc' = 2 \lor$$
$$pc = 2 \land y > 2 \land x' = x \land y' = y \land pc' = 4 \lor$$
$$pc = 2 \land y \leq 2 \land x' = x \land y' = y \land pc' = 3 \lor$$
$$pc = 3 \land x' = x \land y' = y - 1 \land pc' = 2 \lor$$
$$pc = 4 \land x = 2 \land x' = x \land y' = y \land pc' = 5 \lor$$
$$pc = 4 \land x \neq 2 \land x' = x \land y' = y \land pc' = 6 \lor$$
$$pc = 6 \land x = x' \land y = y' \land pc' = 6$$

From Programs to Models

```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   {y = y - 1;}
4: if (x == 2)
5:   {P1:}
6:}
```



State

pc	x	y	···
3	1	3	···

Step

pc	x	y	···
2	1	2	···

Property: $EF\ (pc = 5)$

Computing Reachability

Computation for $EF(pc = 5)$

Stage$_0$	Stage$_1$	Stage$_2$
$pc = 5$	Stage$_0$ $\vee$ $pre[P](pc = 5)$	Stage$_1$ $\vee$ $pre[P]($Stage$_1)$
$pc = 5$	$pc = 5 \vee$ $pc = 4 \wedge x = 2$	$pc = 5 \vee$ $pc = 4 \wedge x = 2 \vee$ $pc = 2 \wedge x = 2 \wedge y > 2$

Representing Transition Relation

```
void main (void) {
1: i
   i
2: w
3:
4: i
5:
6:}
```

$$
\begin{aligned}
& pc = 1 \wedge x' = 2 \wedge y' = 2 \wedge pc' = 2 \vee \\
& pc = 2 \wedge y > 2 \wedge x' = x \wedge y' = y \wedge pc' = 4 \vee \\
& pc = 2 \wedge y \leq 2 \wedge x' = x \wedge y' = y \wedge pc' = 3 \vee \\
& pc = 3 \wedge x' = x \wedge y' = y - 1 \wedge pc' = 2 \vee \\
& pc = 4 \wedge x = 2 \wedge x' = x \wedge y' = y \wedge pc' = 5 \vee \\
& pc = 4 \wedge x \neq 2 \wedge x' = x \wedge y' = y \wedge pc' = 6 \vee \\
& pc = 6 \wedge x = x' \wedge y = y' \wedge pc' = 6
\end{aligned}
$$

$\vee$
$\vee$
$\vee$
$\vee$

Partitioning Transitions by CFG

(pc, pc')	value
$(1,2)$	$x' = 2 \wedge y' = 2$
$(2,4)$	$y > 2 \wedge x' = x \wedge y' = y$
$(2,3)$	$y \leq 2 \wedge x' = x \wedge y' = y$
$(3,2)$	$x' = x \wedge y' = y - 1$
$(4,5)$	$x = 2 \wedge x' = x \wedge y' = y$
$(4,6)$	$x \neq 2 \wedge x' = x \wedge y' = y$
$(6,6)$	$x' = x \wedge y' = y$

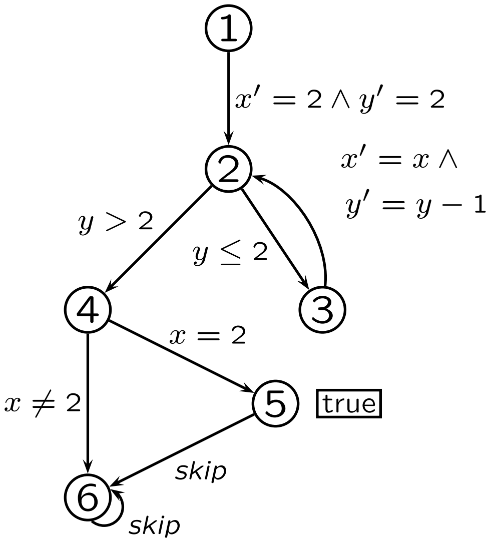
Partitioning State Sets by CFG

Stage_2
$pc = 5 \vee$ $pc = 4 \wedge x = 2 \vee$ $pc = 2 \wedge x = 2 \wedge y > 2$

pc	value
1	false
2	$x = 2 \wedge y > 2$
3	false
4	$x = 2$
5	true
6	false

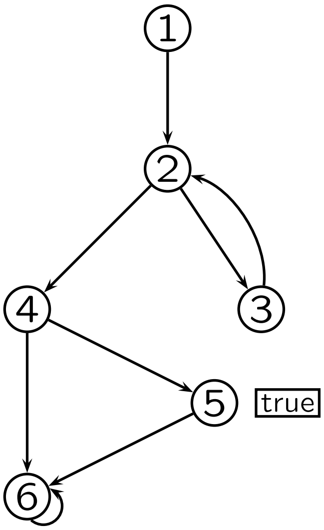
MC with Partitioned Representation

Stage$_0$



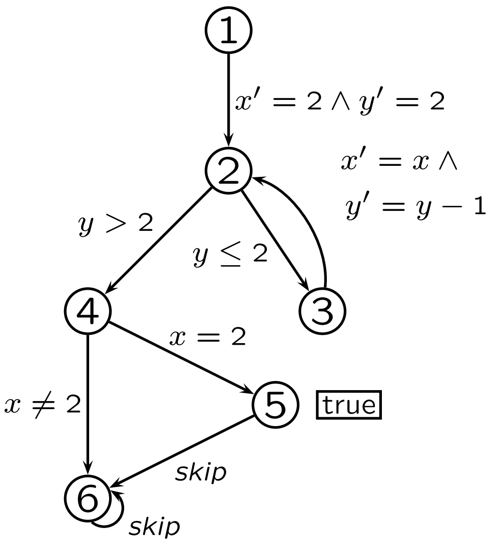
MC with Partitioned Representation

Stage$_0$

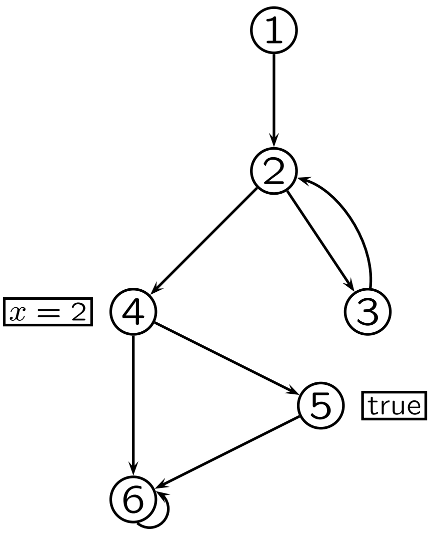


MC with Partitioned Representation

Stage$_0$

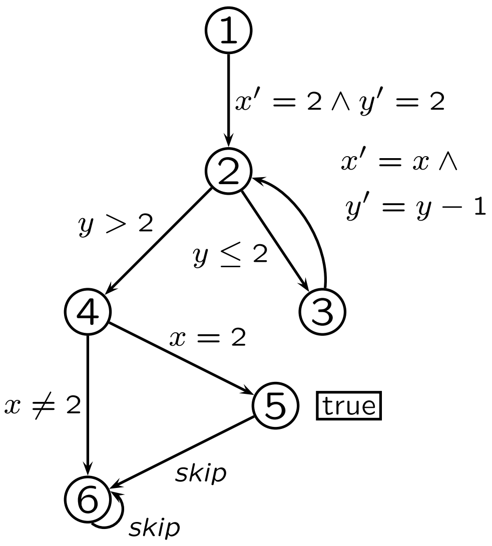


Stage$_1$

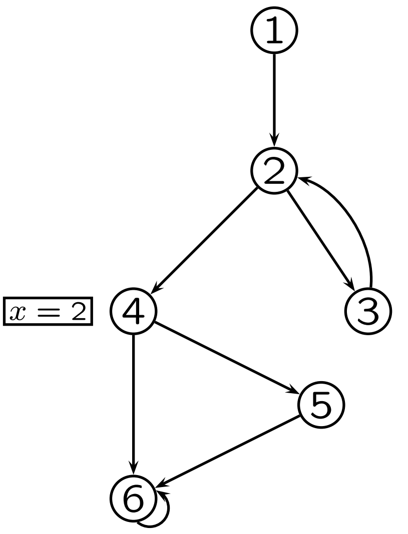


MC with Partitioned Representation

Stage$_0$



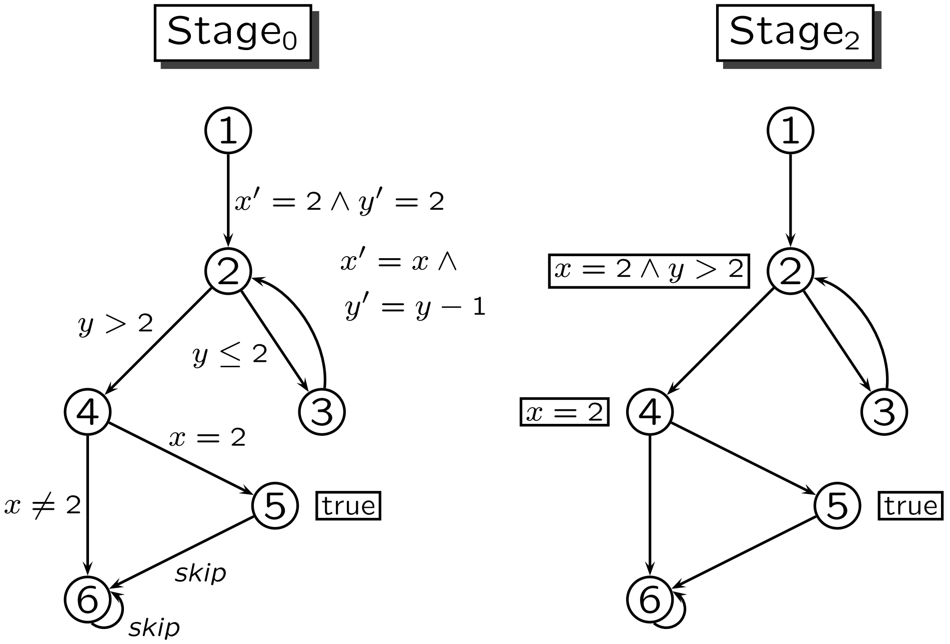
$pre[P]$(Stage$_0$)



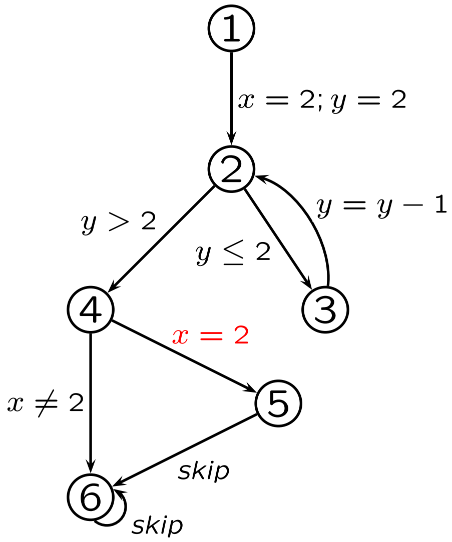
Will This Work?

- **Advantage:**
  - operate on one statement at a time
- **But . . . the number of states at each program point is too large**
  - program with n bits requires 2^n states
    - one int variable: 2^{32} states,
    - two int vars: 2^{64} states,
    - . . .
- **Do we always need that many states?!**

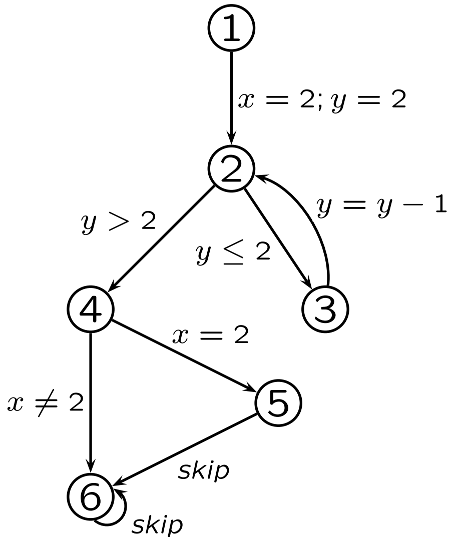
MC with Partitioned Representation



Deciding Reachability Manually



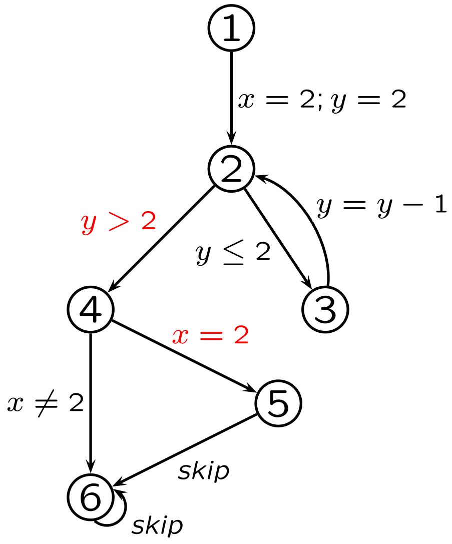
Deciding Reachability Manually



Predicate Abstraction

- **Input:**
  - a finite set of predicates Φ
  - a program P
- **Output: Predicate Program**
  - an approximate description of P using only the predicates from Φ

Deciding Reachability Manually



Approximating a Single Statement

- Describe the behavior of the statement using only a fixed set of predicates

approximating $\mathtt{y} = \mathtt{y} - 1$ using $x = 2$

$x = 2 \;\Rightarrow\; x' = 2 \qquad x \neq 2 \;\Rightarrow\; x' \neq 2$

approximating $\mathtt{y} = \mathtt{y} - 1$ using $y \leq 2$

$y \leq 2 \;\Rightarrow\; y' \leq 2 \qquad ? \;\Rightarrow\; y' > 2$

Predicate Abstraction

Original

```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:    {y = y - 1;}
4: if (x == 2)
5:    {P1:}
6:}
```

Abstract

```
void main (void) {
1: (x=2) := T,
   (y<=2) := T;
2: while (y<=2)
3:    {(y<=2) := (y<=2)? T : *;
        (x=2) := (x=2);}
4: if (x=2)
5:    {P1:}
6:}
```

Approximation: The Algorithm

- **Input:**
  - a set of predicates Φ
  - a program statement `foo`
  - a target predicate φ'
- **Output:**
  - a pair (pos, neg) of formulas over Φ, s.t.

$$\begin{aligned} pos \wedge \mathtt{foo} &\Rightarrow \varphi' \\ neg \wedge \mathtt{foo} &\Rightarrow \neg\varphi' \end{aligned}$$

Approximating a Single Statement

- **Describe the behavior of the statement using only a fixed set of predicates**

approximating $\mathtt{y} = \mathtt{y} - 1$ using $x = 2$

approximating $\mathtt{y} = \mathtt{y} - 1$ using $\{x = 2, y \leq 2\}$

$$\begin{aligned} x = 2 \wedge y \leq 2 &\Rightarrow x' = 2 \wedge y' \leq 2 \\ x \neq 2 \wedge y \leq 2 &\Rightarrow x' \neq 2 \wedge y' \leq 2 \\ ? \qquad\qquad &\Rightarrow x' = 2 \wedge y' > 2 \\ ? \qquad\qquad &\Rightarrow x' \neq 2 \wedge y' > 2 \end{aligned}$$

Example: Statement Abstraction

- **Input:**
  - $\Phi = \{y \leq 2, x = 2\}$
  - statement $y := y - 1$ (or $x' = x \wedge y' = y - 1$)
  - target predicate $\varphi' = (y' \leq 2)$
- **Output**
  - $\texttt{res} = (y \leq 2 \wedge x = 2) \vee (y \leq 2 \wedge x \neq 2)$

Approximation: The Algorithm

- **Input:**
  - a set of predicates Φ
  - a program stateme
  - a target predicate
- **Output:**
  - a pair (pos, neg) o

$$pos \wedge \texttt{foo} \quad \Rightarrow \quad \varphi'$$

$$neg \wedge \texttt{foo} \quad \Rightarrow \quad \neg$$

The Algorithm

```
Input: $\Phi$, $\varphi'$, foo

res := false
for all conj $\in$ conj($\Phi$) do
    if (conj $\wedge$ foo $\Rightarrow$ $\varphi'$) then
        res := res $\vee$ conj
    end if
end for
```

Example: Statement Abstraction

- **Input:**
  - $\Phi = \{y \leq 2, x = 2\}$
  - statement $y := y - 1$ (or $x' = x \wedge y' = y - 1$)
  - target predicate $\varphi' = (y' \leq 2)$
- **Output**
  - `res` $= (y \leq 2 \wedge x = 2) \vee (y \leq 2 \wedge x \neq 2)$
- **Complexity**
  - 2 execution per each predicate
  - $O(2^n)$ calls to decision procedure per execution

Example: Statement Abstraction

- **Input:**
  - $\Phi =$
  - staten
  - targe
- **Outpu**
  - `res` $= (y \leq 2 \wedge x = 2) \vee (y \leq 2 \wedge x \neq 2)$

$$y \leq 2 \wedge x = 2 \wedge$$
$$y' = y - 1 \wedge x' = x \;\Rightarrow\; y' \leq 2$$

$$y \leq 2 \wedge x \neq 2 \wedge$$
$$y' = y - 1 \wedge x' = x \;\Rightarrow\; y' \leq 2$$

Approximation: The End Result

Before	After
`foo`	$p_0 := (pos_0, neg_0),$ $p_1 := (pos_1, neg_1),$ $\vdots$ $p_n := (pos_n, neg_n)$
$y := y - 1$	$x = 2 := (x = 2, x \neq 2),$ $y \leq 2 := (y \leq 2, \text{false})$

Approximation: The End Result

Before	After
`foo`	$p_0 := (pos_0, neg_0),$ $p_1 := (pos_1, neg_1),$ $\vdots$ $p_n := (pos_n, neg_n)$

Predicate Program

Original

```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   {y = y - 1;}
4: if (x == 2)
5:   {P1:}
6:}
```

Abstract

$$\Phi = \{x = 2\}$$

```
void main (void) {
1: (x=2) := T;
2: while (*)
3:   {(x=2) := (x=2);}
4: if (x=2)
5:   {P1:}
6:}
```

Predicate Program

Original

```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   {y = y - 1;}
4: if (x == 2)
5:   {P1:}
6:}
```

Abstract

$$\Phi = \{\}$$

```
void main (void) {
1:   ;
2: while (*)
3:   { ; }
4: if (*)
5:   {P1:}
6:}
```

Handling Unknowns

- **Treat “unknown” as non-deterministic (over-approximation)**
  - suited for proving unreachability
- **Treat “unknown” as abort (under-approximation)**
  - suited for proving reachability
- **Fix the model-checker (Yasm)**
  - treat “unknown” as unknown
  - works for proving both reachability and unreachability

Predicate Program

Original

```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:    {y = y - 1;}
4: if (x == 2)
5:    {P1:}
6:}
```

Abstract

$$\Phi = \{x = 2, y \leq 2\}$$

```
void main (void) {
1: (x=2) := T,
   (y<=2) := T;
2: while (y<=2)
3:    {(y<=2) := (y<=2)? T : *;
       (x=2) := (x=2);}
4: if (x=2)
5:    {P1:}
6:}
```

Partial Pre-Image Computation

- **Partial Statement:** P

$$x = 2 \ := \ (x = 2, x \neq 2),$$

$$y \leq 2 \ := \ (y \leq 2, \text{false})$$

- **Set of States:** φ

true	false
$x = 2 \wedge y \leq 2$	$x \neq 2 \vee y > 2$

- **Result of pre-image computation:** $pre[P](\varphi)$

true	false	unknown
$x = 2 \wedge y \leq 2$	$x \neq 2$	$x = 2 \wedge y > 2$

Representing Unknowns

- **Representing approximate sets of states**
  - a set S such that

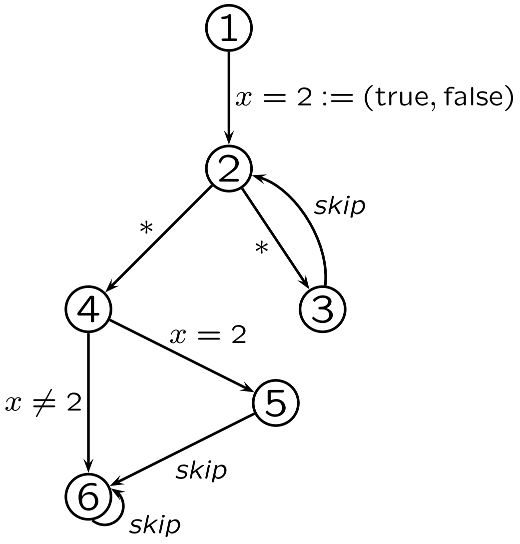
    $(x = 2 \wedge y \leq 2) \in S, \qquad (x \neq 2) \notin S,$

    $x = 2 \wedge y > 2$ unknown
  - represented by 2 formulas

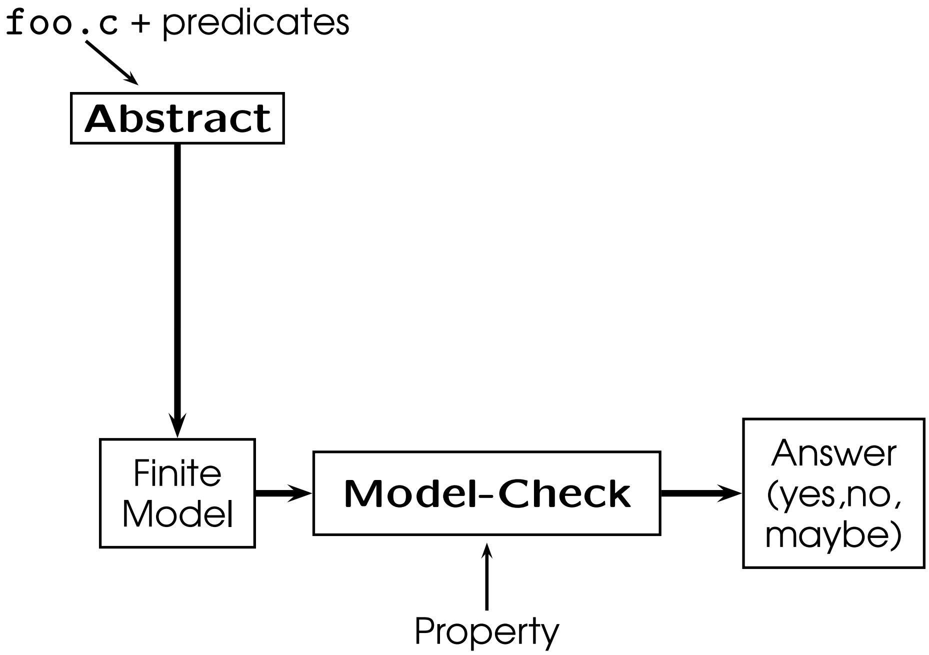
true	false
$x = 2 \wedge y > 2$	$x \neq 2$

- **Representing an approximate transition relation**
  - similar to sets of states

Where Do Predicates Come From?



Abstraction: Summary



`foo.c` + predicates

Abstract

Finite Model

Model-Check

Answer (yes,no, maybe)

Property

Predicate Program

Original

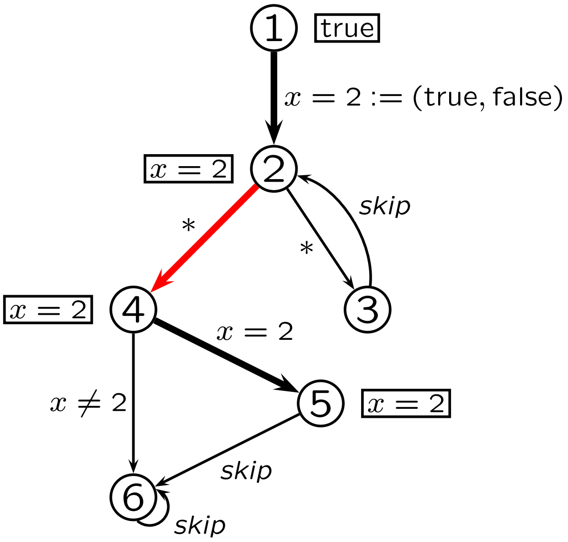
```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   {y = y - 1;}
4: if (x == 2)
5:   {P1:}
6:}
```

Abstract

$$\Phi = \{\}$$

```
void main (void) {
1:  ;
2: while (*)
3:   { ; }
4: if (*)
5:   {P1:}
6:}
```

Where Do Predicates Come From?



Predicate Program

Original

```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   {y = y - 1;}
4: if (x == 2)
5:   {P1:}
6:}
```

Abstract

$$\Phi = \{x = 2, y \leq 2\}$$

```
void main (void) {
1: (x=2) := T,
   (y<=2) := T;
2: while (y<=2)
3:   {(y<=2) := (y<=2)? T : *;
      (x=2) := (x=2);}
4: if (x=2)
5:   {P1:}
6:}
```

Predicate Program

Original

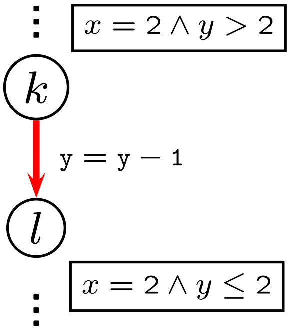
```
void main (void) {
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   {y = y - 1;}
4: if (x == 2)
5:   {P1:}
6:}
```

Abstract

$$\Phi = \{x = 2\}$$

```
void main (void) {
1: (x=2) := T;
2: while (*)
3:   {(x=2) := (x=2);}
4: if (x=2)
5:   {P1:}
6:}
```

Another Example



$x = 2 \land y > 2$

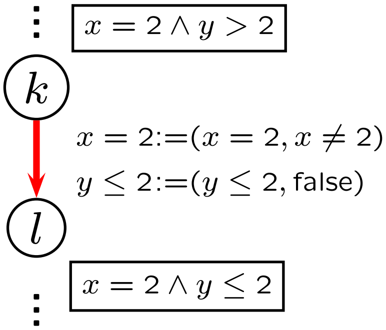
k

`y = y - 1`

l

$x = 2 \land y \leq 2$

Another Example



$x = 2 \land y > 2$

k

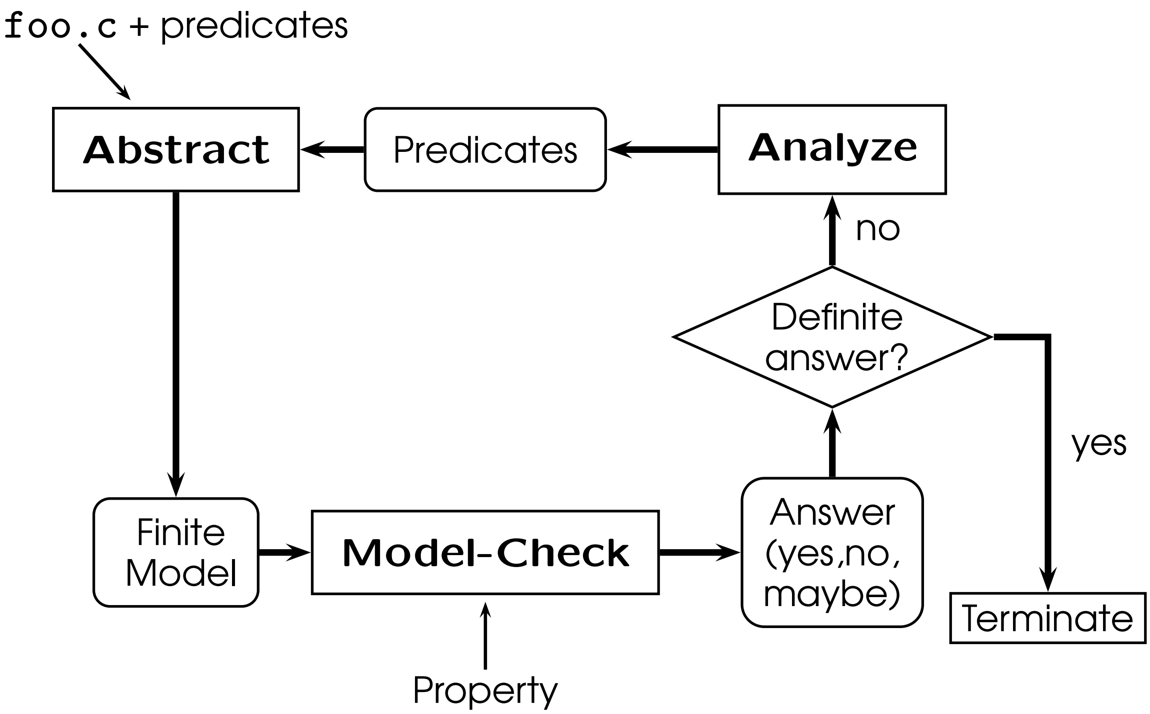
$x = 2 := (x = 2, x \neq 2)$

$y \leq 2 := (y \leq 2, \text{false})$

l

$x = 2 \land y \leq 2$

Abstraction Refinement Cycle



`foo.c` + predicates

Abstract

Predicates

Analyze

no

Definite answer?

yes

Finite Model

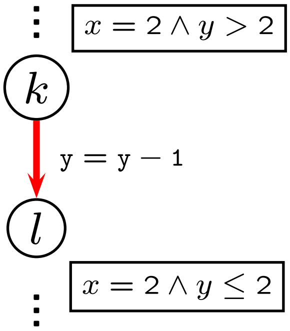
Model-Check

Answer (yes,no, maybe)

Terminate

Property

Another Example



- **The result is unknown because. . .**

  $$y > 2 \wedge (\mathrm{y} = \mathrm{y} - 1) \quad \not\Rightarrow \quad y' \leq 2$$

  $$y > 2 \wedge (\mathrm{y} = \mathrm{y} - 1) \quad \not\Rightarrow \quad y' > 2$$

- **Idea:**
  look at the precondition for $y' \leq 2$

  $$wp[\mathrm{y} = \mathrm{y} - 1](y \leq 2) \quad = \quad y \leq 3$$

- **Solution:**
  add $y \leq 3$ and repeat

Demo

YASM: The Tool

- **Abstract**
  - several abstraction strategies
  - uses CVCLite (Stanford + NYU)
- **Model-Check**
  - symbolic multi-valued model-checker
  - based on CUDD (Colorado)
- **Analyze**
  - based on proof-like counterexamples
  - several strategies to complement abstraction

YASM: Current State

- **Scalability**
  - size: about 5,000 LOC
  - time: from minutes to an hour
- **Major Bottleneck: The Front End**
  - limited support for arrays, structures, pointers, etc. . .
- **New version**
  - optimized for reachability properties
  - success with 50,000 LOC (`OpenSSH`)
  - supports recursive functions
  - more to come. . .

Other Software Model-Checkers

- **Symbolic (BDD)**
  - SLAM (Microsoft Research)
  - BLAST (Berkley)
- **Symbolic (SAT)**
  - MAGIC (CMU + SEI)
  - CBMC (CMU)
- **Explicit State**
  - VeriSoft (Lucent)
  - JavaPathFinder (NASA)
  - Zing (Microsoft Research)
  - Bogor (Kansas State)

Thank You

Current Research Directions

- **Abstract**
  - time vs precision trade-offs
  - abstraction reuse
  - combining with other abstract domains
- **Model-Check**
  - incremental analysis
  - concurrency
  - progress and termination analysis
- **Analyze**
  - new techniques for predicate discovery
  - improvements for liveness properties

Introduction

Static Analysis vs Software Model-Checking

	Static Analysis	**Model-Checking**
Goal	Software Analysis	
Properties	hard-coded	temporal logic
Deployment	compilers	testing
Focus	time	result

Software MC is a form of Static Analysis

Introduction

Static Analysis vs Software Model-Checking

	Static Analysis	**Model-Checking**
Goal	Software Analysis	
Properties	hard-coded	temporal logic
Deployment	compilers	testing
Focus	time	result