Last Lecture: Review

Testing, Quality Assurance, and Maintenance
Winter 2017

Prof. Arie Gurfinkel

UNIVERSITY OF WATERLOO

Syntax versus Semantics

TESTING AND VERIFICATION

Testing and Verification / Quality Assurance

Testing: Software validation the “old-fashioned” way

- create a test suite (a set of test cases)
- run and identify failures
- fix to address failures and repeat
- done when the test suite passes and achieves a desired criteria

Verification: formally prove that a computing system satisfies its specifications

- Rigor: well established mathematical foundations
- Exhaustiveness: considers all possible behaviors of the system, i.e., finds all errors
- Automation: uses computers to build reliable computers

“Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.”

Edsger W. Dijkstra

Very hard to test the portion inside the “if" statement!

```
input x
if (hash(x) == 10) {
    ...
}
```

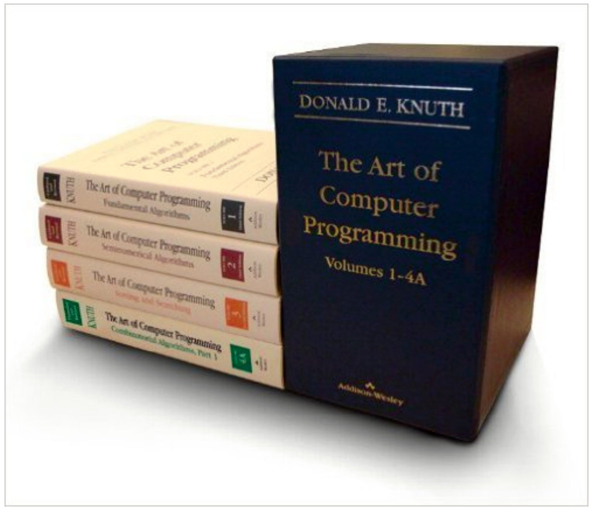
UNIVERSITY OF WATERLOO

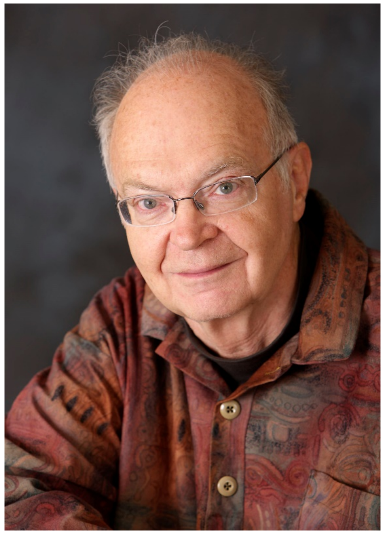
"Beware of bugs in the above code; I have only proved it correct, not tried it."

Donald Knuth

You can only verify what you have specified.

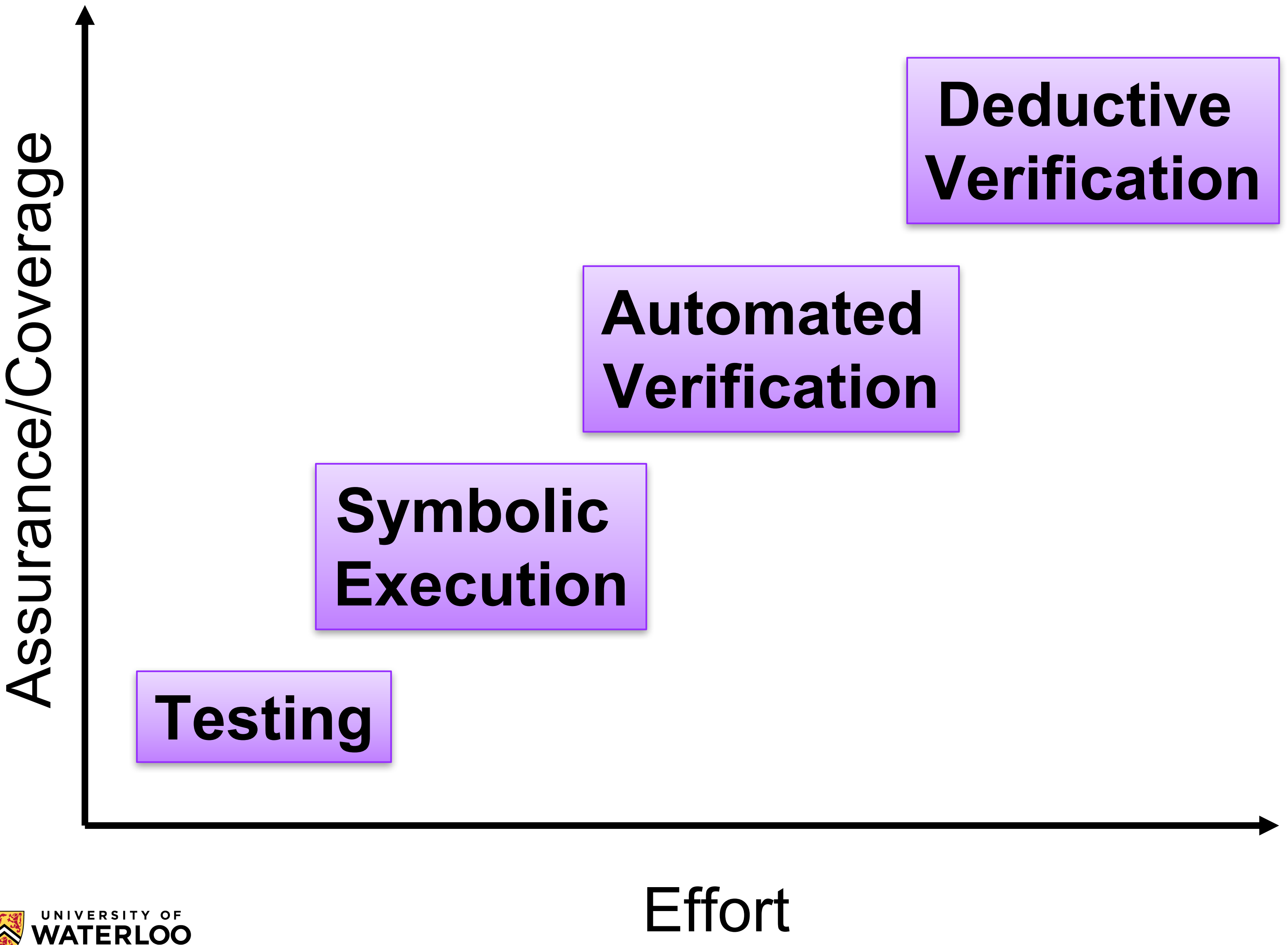
Testing is still important, but can we make it less impromptu?





UNIVERSITY OF WATERLOO

(User) Effort vs (Verification) Assurance



Undecidability

A problem is undecidable if there does not exists a Turing machine that can solve it

- i.e., not solvable by a computer program

The halting problem

- does a program P terminates on input I
- proved undecidable by Alan Turing in 1936
- https://en.wikipedia.org/wiki/Halting_problem

Rice’s Theorem

- for any non-trivial property of partial functions, no general and effective method can decide whether an algorithm computes a partial function with that property
- in practice, this means that there is no machine that can always decide whether the language of a given Turing machine has a particular nontrivial property
- https://en.wikipedia.org/wiki/Rice%27s_theorem

Topics Covered in the Course

Foundations

- syntax, semantics, abstract syntax trees, visitors, control flow graphs

Testing

- coverage: structural, dataflow, and logic

Symbolic Execution

- using SMT solvers, constraints, path conditions, exploration strategies
- building a (toy) symbolic execution engine

Deductive Verification

- Hoare Logic, weakest pre-condition calculus, verification condition generation
- verifying algorithm using Dafny, building a small verification engine

Automated Verification

- (basics of) software model checking

HOARE LOGIC

I THINK YOU SHOULD BE MORE SPECIFIC HERE IN STEP TWO
THEN A MIRACLE OCCURS . . .

Assertions for WHILE

{A} c {B} is a partial correctness assertion. It does not imply termination of c.

- If A holds in state q and there exists q' such that $q \rightarrow q'$
- then B holds in state q'

[A] c [B] is a total correctness assertion meaning that

- If A holds in state q
- then there exists q' such that $q \rightarrow q'$
  and B holds in state q'

Inference Rules for Hoare Triples

We write ⊢ {A} c {B} when we can derive the triple using inference rules

There is one inference rule for each command in the language

Plus, the *rule of consequence*

- e.g., strengthen pre-condition, weaken post-condition

$$\frac{\vdash A' \implies A \qquad \{A\}\,c\,\{B\} \qquad \vdash B \implies B'}{\{A'\}\,c\,\{B'\}}\ \textsc{Conseq}$$

Inference Rules for WHILE language

One rule for each syntactic construct:

$$\vdash \{A\}\ \texttt{skip}\ \{A\}$$

$$\frac{\vdash \{A\}\ s_1\ \{B\} \qquad \vdash \{B\}\ s_2\ \{C\}}{\vdash \{A\}\ s_1;\ s_2\ \{C\}}$$

$$\vdash \{A[e/x]\}\ x\texttt{:=}e\ \{A\}$$

$$\frac{\vdash \{A \land b\}\ s_1\ \{B\} \qquad \vdash \{A \land \neg b\}\ s_2\ \{B\}}{\vdash \{A\}\ \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{B\}}$$

$$\frac{\vdash \{I \land b\}\ s\ \{I\}}{\vdash \{I\}\ \texttt{while}\ b\ \texttt{do}\ s\ \{I \land \neg b\}}$$

UNIVERSITY OF WATERLOO

Inductive Loop Invariants

$$\frac{\vdash Pre \Rightarrow Inv \quad \vdash \{\, Inv \land b \,\}\; \mathtt{s}\; \{\, Inv \,\} \quad \vdash Inv \land \neg b \Rightarrow Post}{\{\, Pre \,\}\; \mathtt{while\ b\ do\ s}\; \{\, Post \,\}}$$

Inv is an inductive loop invariant if the following three conditions hold:

- (Initiation) Inv holds **initially** whenever the loop is reached. That is, it is true of the pre-condition *Pre*
- (Consecution) Inv is **preserved**: executing the loop body c from any state satisfying Inv and loop condition b ends in a state satisfying Inv
- (Safety) Inv is **strong enough**: Inv and the negation of loop condition b imply the desired post-condition *Post*

Example: a more interesting program

We want to derive that

$\{n \geq 0\}$

```
p := 0;
x := 0;
while x < n do
  x := x + 1;
  p := p + m
```

$\{p = n * m\}$

Example: a more interesting program

Only applicable rule (except for rule of consequence):

$$\frac{\vdash \{A\}\ c_1 \{C\} \quad \vdash \{C\}\ c_2\ \{B\}}{\vdash \{A\}\ c_1;\ c_2\ \{B\}}$$

$$\frac{\vdash \{n \geq 0\}\ \text{p:=0; x:=0}\ \{C\} \qquad \vdash \{C\}\ \text{while x < n do (x:=x+1; p:=p+m)}\ \{p = n * m\}}{\vdash \underbrace{\{n \geq 0\}}_{A}\ \underbrace{\text{p:=0; x:=0;}}_{c_1}\ \underbrace{\text{while x < n do (x:=x+1; p:=p+m)}}_{c_2}\ \underbrace{\{p = n * m\}}_{B}}$$

Example: a more interesting program

What is C? Look at the next possible matching rules for c_2!

Only applicable rule (except for rule of consequence):

$$\frac{\vdash \{\mathtt{I} \wedge b\}\ c\ \{\mathtt{I}\}}{\vdash \{\mathtt{I}\}\ \mathtt{while}\ b\ \mathtt{do}\ c\ \{\mathtt{I} \wedge \neg b\}}$$

We can match {I} with {C} but we cannot match {I ∧ ¬b} and {p = n * m} directly. Need to apply the rule of consequence first!

$$\frac{\vdash \{n \geq 0\}\ \text{p:=0; x:=0}\ \{C\} \qquad \vdash\{C\}\ \text{while x < n do (x:=x+1; p:=p+m)}\ \{p = n * m\}}{\vdash \underbrace{\{n \geq 0\}}_{A}\ \underbrace{\text{p:=0; x:=0;}}_{c_1}\ \underbrace{\text{while x < n do (x:=x+1; p:=p+m)}}_{c_2}\ \underbrace{\{p = n * m\}}_{B}}$$

Example: a more interesting program

What is C? Look at the next possible matching rules for c_2!

Only applicable rule (except for rule of consequence):

$$\frac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \underbrace{\{I\}}_{A}\ \underbrace{\texttt{while}\ b\ \texttt{do}\ c}_{c'}\ \underbrace{\{I \wedge \neg b\}}_{B}}$$

Rule of consequence:

$$\frac{\vdash A' \Rightarrow A \qquad \vdash \{A\}\ c'\ \{B\} \qquad \vdash B \Rightarrow B'}{\vdash \{A'\}\ c'\ \{B'\}}$$

$I = A = A' = C$

$$\frac{\vdash \{n \geq 0\}\ \text{p:=0; x:=0}\ \{C\} \qquad \vdash \overbrace{\{C\}}^{A'}\ \overbrace{\text{while x < n do (x:=x+1; p:=p+m)}}^{c'}\ \overbrace{\{p = n * m\}}^{B'}}{\vdash \{n \geq 0\}\ \text{p:=0; x:=0; while x < n do (x:=x+1; p:=p+m)}\ \{p = n * m\}}$$

Example: a more interesting program

What is I? Let's keep it as a placeholder for now!

Next applicable rule:

$$\frac{\vdash \{A\}\ c_1 \{C\} \qquad \vdash \{C\}\ c_2\ \{B\}}{\vdash \{A\}\ c_1;\ c_2\ \{B\}}$$

$$\frac{\vdash \{n \geq 0\}\ p:=0;\ x:=0\ \{I\} \qquad \dfrac{\dfrac{\vdash \underbrace{\{I \land x<n\}}_{A}\ \underbrace{x := x+1;}_{c_1}\ \underbrace{p:=p+m}_{c_2}\ \underbrace{\{I\}}_{B}}{\vdash \{I\}\ \text{while } x < n \text{ do } (x:=x+1;\ p:=p+m)\ \{I \land x \geq n\}} \qquad \vdash I \land x \geq n \Rightarrow p = n * m}{\vdash \{I\}\ \text{while } x < n \text{ do } (x:=x+1;\ p:=p+m)\ \{p = n * m\}}}{\vdash \{n \geq 0\}\ p:=0;\ x:=0;\ \text{while } x < n \text{ do } (x:=x+1;\ p:=p+m)\ \{p = n * m\}}$$

Example: a more interesting program

What is C? Look at the next possible matching rules for c_2!

Only applicable rule (except for rule of consequence):

$\vdash \{A[e/x]\}\ x{:=}e\ \{A\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \overbrace{\{I \land x<n\}}^{A}\ \overbrace{x := x+1}^{c_1}\ \{C\} \qquad \vdash \{C\}\ \overbrace{p{:=}p{+}m}^{c_2}\ \overbrace{\{I\}}^{B}}{\vdash \{I \land x<n\}\ x := x+1;\ p{:=}p{+}m\ \{I\}}}{\vdash \{I\}\ \text{while } x < n \text{ do } (x{:=}x{+}1;\ p{:=}p{+}m)\ \{I \land x \geq n\}} \quad \vdash I \land x \geq n \Rightarrow p = n * m}{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0\ \{I\} \quad \vdash \{I\}\ \text{while } x < n \text{ do } (x{:=}x{+}1;\ p{:=}p{+}m)\ \{p = n * m\}}}{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0;\ \text{while } x < n \text{ do } (x{:=}x{+}1;\ p{:=}p{+}m)\ \{p = n * m\}}$$

DYNAMIC SYMBOLIC EXECUTION

Concrete Operational Semantics of If-Stmt

$$\frac{\langle b, q\rangle \Downarrow \text{true} \qquad \langle s_1, q\rangle \Downarrow q'}{\langle \textbf{if}\, b\, \textbf{then}\ s_1\ \textbf{else}\ s_2, q\rangle \Downarrow q'}$$

$$\frac{\langle b, q\rangle \Downarrow \text{false} \qquad \langle s_2, q\rangle \Downarrow q'}{\langle \textbf{if}\, b\, \textbf{then}\ s_1\ \textbf{else}\ s_2, q\rangle \Downarrow q'}$$

If b is true, do s_1, otherwise, if b is false, do s_2

UNIVERSITY OF WATERLOO

https://ece.uwaterloo.ca/~agurfink/ece653/assets/pdf/opsymexec.pdf

Symbolic Semantics of If-Stmt

$$\frac{\langle b, q\rangle \Downarrow v \qquad pc \wedge v \text{ is SAT} \qquad \langle s_1, q, pc \wedge v\rangle \Downarrow q', pc'}{\langle \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, q, pc\rangle \Downarrow q', pc'}$$

$$\frac{\langle b, q\rangle \Downarrow v \qquad pc \wedge \neg v \text{ is SAT} \qquad \langle s_2, q, pc \wedge \neg v\rangle \Downarrow q', pc'}{\langle \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, q, pc\rangle \Downarrow q', pc'}$$

A state is a pair (q, pc), where q is a map from variables to values and pc is a FOL formula called *path condition*

If b=true is consistent with the current path condition then do s_1

If b=false is consistent with the current path condition then do s_2

Both rules can be applicable at the same time == many reachable states

UNIVERSITY OF WATERLOO

https://ece.uwaterloo.ca/~agurfink/ece653/assets/pdf/opsymexec.pdf

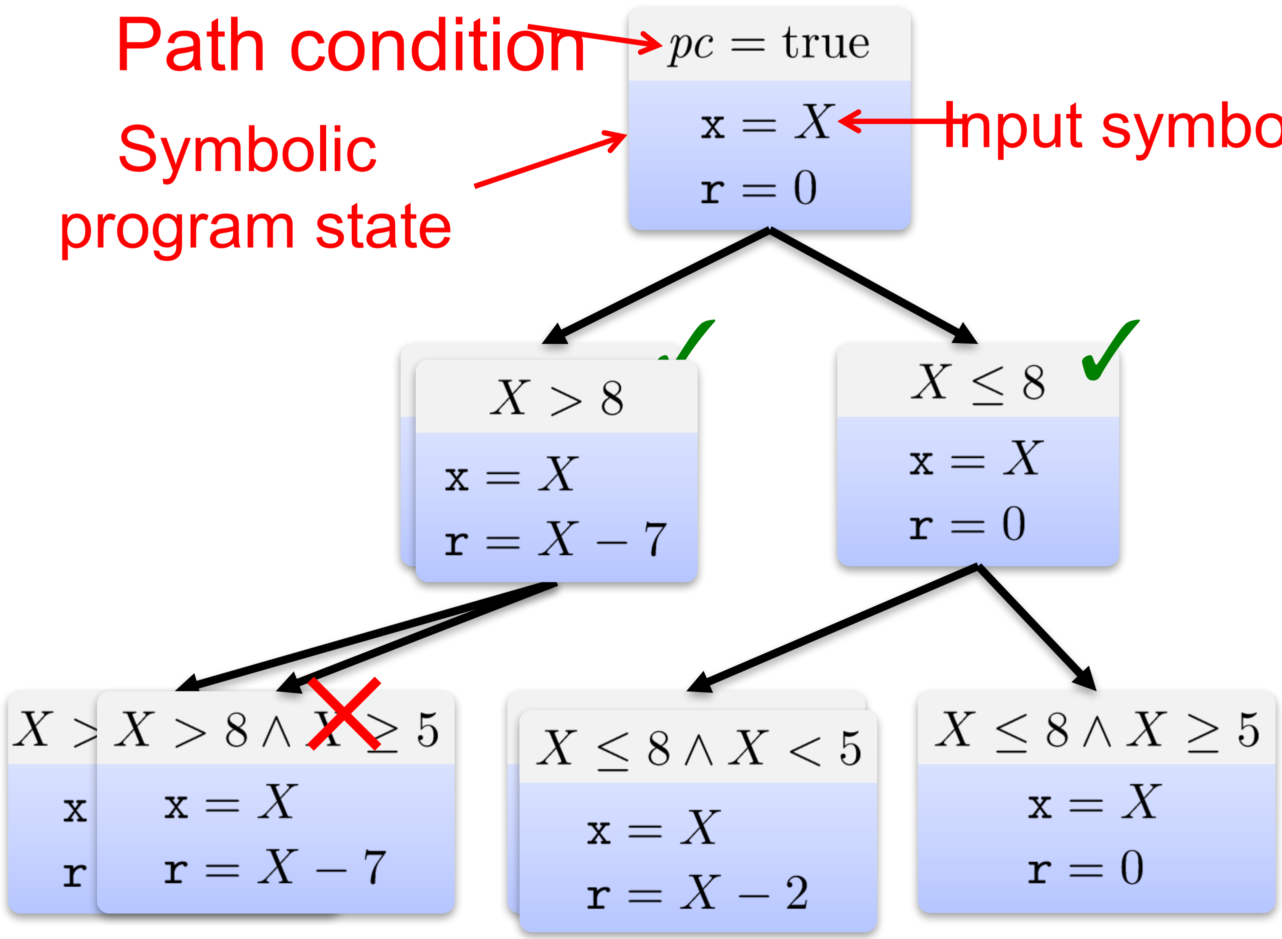
```
int proc(int x) {

  int r = 0

  if (x > 8) {
    r = x - 7
  }

  if (x < 5) {
    r = x - 2
  }

  return r
}
```



Satisfying assignments:

`X = 9`   `X = 4`   `X = 7`

Test cases:

`proc(9)`   `proc(4)`   `proc(7)`

Symbolic State

A ***symbolic state*** is a pair S = (Env, PC), where

- Env : L -> E is a mapping, called an ***environment***, from program variables to symbolic expressions (i.e., FOL terms)
- PC is a FOL formula called a ***path condition***

A concrete state M : L -> Z satisfies a symbolic state S = (Env, PC) iff

$$M \models (Env, PC) \text{ iff } \left((\bigwedge_{v \in L} M(v) = Env(v)) \wedge PC \text{ is SAT} \right)$$

Program semantics are extended to symbolic states

- each program statement updates symbolic variables and
- extends the path condition to reflect its operational semantics

UNIVERSITY OF WATERLOO

Example: Symbolic State Satisfiability

$$Env = \begin{cases} x \mapsto X \\ y \mapsto Y \end{cases} \qquad PC = X > 5 \wedge Y < 3$$

$$[x \mapsto 10, y \mapsto 1] \models ?S \qquad [x \mapsto 1, y \mapsto 10] \models ?S$$

$$Env = \begin{cases} x \mapsto X + Y \\ y \mapsto Y - X \end{cases} \qquad PC = 2 * X - Y > 0$$

$$[x \mapsto 10, y \mapsto 1] \models ?S \qquad [x \mapsto 1, y \mapsto 10] \models ?S$$

Flavors of Symbolic Execution Algorithms

Static symbolic execution

- Simulate execution on program source code
- Computes strongest post-conditions from entry point

Dynamic symbolic execution (DSE)

- Run / interpret the program with concrete state
- Symbolic state computed in parallel (“concolic”)
- Solver generates new concrete state

DSE-Flavors

- EXE-style [Cadar et al. ‘06] vs. DART [Godefroid et al. ‘05]

Many successful tools

- EXE → KLEE (Imperial), SPF (NASA), Cloud9, S2E (EPFL)
- DART → SAGE, PEX (Microsoft), CUTE (UIUC), CREST (Berkeley)

DSE Semantics of If-Stmt

$$\frac{\langle b, con(q)\rangle \Downarrow \text{true} \qquad \langle b, q\rangle \Downarrow v \qquad \langle s_1, \langle con(q), sym(q), pc(q) \wedge v\rangle\rangle \Downarrow q'}{\langle \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2, q\rangle \Downarrow q'}$$

A state is a triple (con(q), sym(q), pc), where

- con(q) is a map from variables to concrete values
- sym(q) is a map from variables to symbolic values
- pc is the path condition formula

Each statement is executed both concretely and symbolically

If b is true concretely, add v, the symbolic value of b, to the path condition and execute s_1

https://ece.uwaterloo.ca/~agurfink/ece653/assets/pdf/opsymexec.pdf

UNIVERSITY OF WATERLOO

DSE Semantics of If-Stmt

$$\frac{\begin{array}{c}\langle b, con(q)\rangle \Downarrow \text{true} \\ \langle b, q\rangle \Downarrow v \qquad pc(q) \wedge \neg v \text{ is SAT} \qquad c \models \langle sym(q), pc(q) \wedge \neg v\rangle \\ c \equiv_{con} con(q) \qquad \langle s_2, \langle c, sym(q), pc(q) \wedge \neg v\rangle\rangle \Downarrow q'\end{array}}{\langle \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2, q\rangle \Downarrow q'}$$

The complicated case

If b=true concretely but b=false is consistent with the path condition

- find a new concrete c state in which b=false
- create new DSE state using new concrete state and old symbolic state
- execute s_2 from the new DSE state

UNIVERSITY OF WATERLOO

https://ece.uwaterloo.ca/~agurfink/ece653/assets/pdf/opsymexec.pdf

EXE Algortihm

Program state is a tuple (ConcreteState, SymbolicState)

Initially all input variables are symbolic

At each execution step

- update the concrete state by executing a program instruction concretely
- update symbolic state executing symbolically
- if the last instruction was a branch
  - if PC and negation of branch condition is SAT
    - fork execution state
    - compute new concrete to match the new path condition

EXE

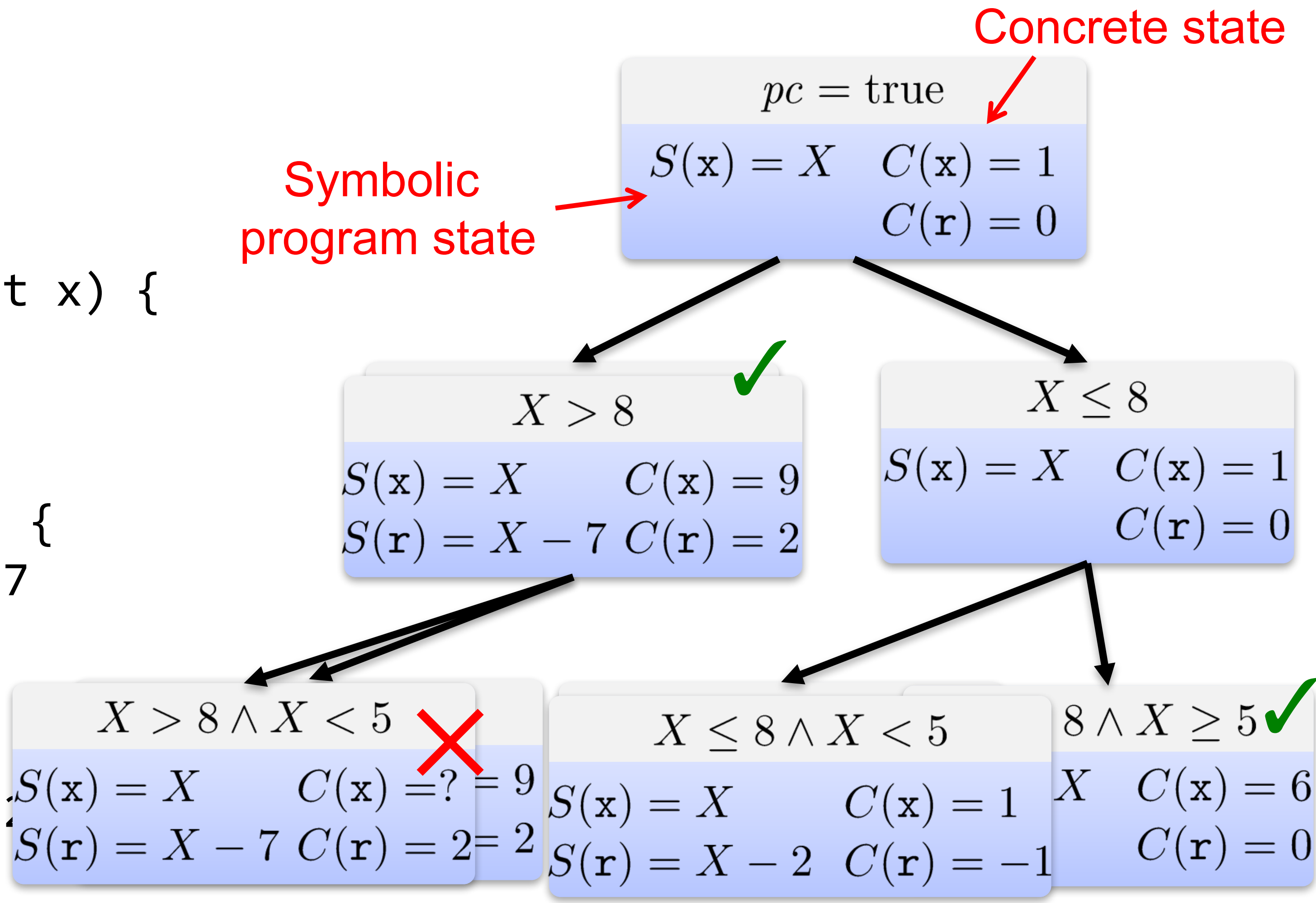
```
int proc(int x) {

  int r = 0

  if (x > 8) {
    r = x - 7
  }

  if (x < 5)
    r = x -
  }

  return r;
}
```



Concrete state
Symbolic program state
$pc = \text{true}$
$S(\texttt{x}) = X$ $C(\texttt{x}) = 1$ $C(\texttt{r}) = 0$
$X > 8$ ✓
$S(\texttt{x}) = X$ $C(\texttt{x}) = 9$ $S(\texttt{r}) = X - 7$ $C(\texttt{r}) = 2$
$X \leq 8$
$S(\texttt{x}) = X$ $C(\texttt{x}) = 1$ $C(\texttt{r}) = 0$
$X > 8 \wedge X < 5$ ✗
$S(\texttt{x}) = X$ $C(\texttt{x}) =?$ $S(\texttt{r}) = X - 7$ $C(\texttt{r}) = 2$
$X \leq 8 \wedge X < 5$
$S(\texttt{x}) = X$ $C(\texttt{x}) = 1$ $S(\texttt{r}) = X - 2$ $C(\texttt{r}) = -1$
$8 \wedge X \geq 5$ ✓
X $C(\texttt{x}) = 6$ $C(\texttt{r}) = 0$

Satisfying assignments:

`X = 9` `X = 1` `X = 6`

Test cases:

`proc(9)` `proc(1)` `proc(6)`

Concretization

Parts of input space can be kept concrete

- Reduces complexity
- Focuses search

Expressions can be concretized at runtime

- Avoid expressions outside of SMT solver theories (non-linear etc.)

Sound but incomplete

Semantics of Concretization in DSE

$$\frac{\langle x, sym(q)\rangle \Downarrow v \qquad \begin{array}{c} sym(x) \qquad \langle x, con(q)\rangle \Downarrow n \\ \langle s, \langle con(q), sym(q)[x := n], pc(q) \wedge v = n\rangle\rangle \Downarrow q' \end{array}}{\langle s, q\rangle \Downarrow q'}$$

Any symbolic variable can be set to its value in the concrete state as long as the path condition is updated to reflect the new assignment

UNIVERSITY OF WATERLOO

https://ece.uwaterloo.ca/~agurfink/ece653/assets/pdf/opsymexec.pdf

Implementing Concretization

Input

- concrete and symbolic states `C` and `S`
- a symbolic expression E to evaluate

Algorithm

- pick variables $x_1, \ldots, x_k$ in E to concretize
- replace x_i by $C(x_i)$ in E
- `v:=S.eval`(E); CC := $x_1=C(x_1) \wedge \ldots \wedge x_k=C(x_k)$
- add *concretization constraint* CC to the path condition
- return v

Concretization in EXE (Example)

$true$	
	$\boxed{C(X) = 5}$
$S(\texttt{m}) = X + 2$	$C(\texttt{m}) = 7$
$S(\texttt{size}) = Y$	$C(\texttt{size}) = 256$

```
if (m*m > size) {
  …
```

$$(X + 2)(X + 2) > Y$$

$$(X + 2)(X + 2) < 5$$

Concretization constraint

$$(5 + 2)(5 + 2) > Y$$

$$49 > Y$$

Solution diverges from expected path! (e.g., X = 2)

$49 > Y \boxed{\wedge X = 5}$	
	$C(X) = 5$
$S(\texttt{m}) = X + 2$	$C(\texttt{m}) = 7$
$S(\texttt{size}) = Y$	$\boxed{C(\texttt{size}) = 48}$

DART: Algorithm

```
Formula F := False
Loop
        Find program input i in solve(negate(F))  // stop if no such i can be found
        Execute P(i); record path condition C // in particular, C(i) holds
        F := F \/ C
End
```

Property: Every CFG path is explored only once

UNIVERSITY OF WATERLOO

DART

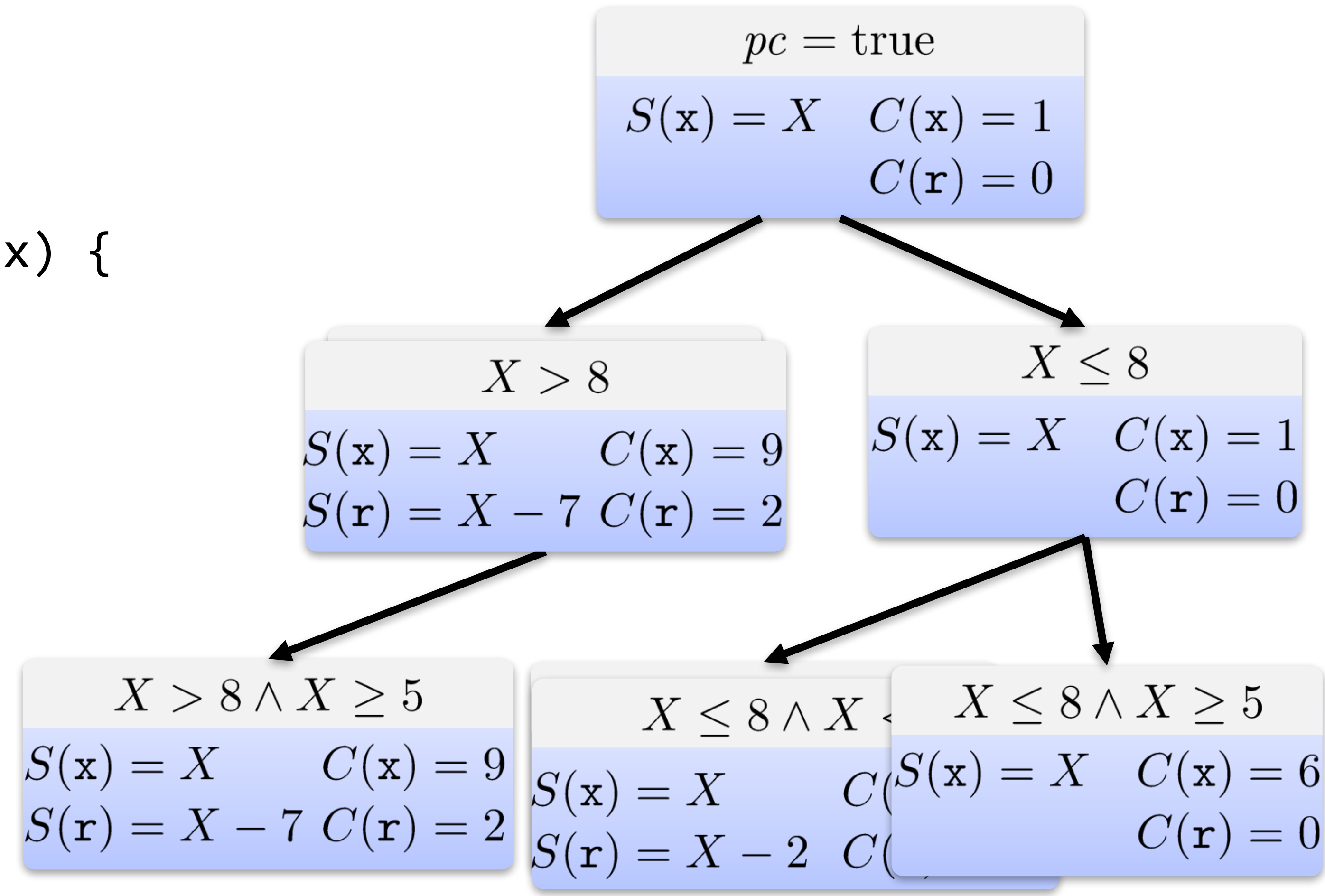
```
int proc(int x) {

  int r = 0

  if (x > 8) {
    r = x - 7
  }

  if (x < 5) {
    r = x - 2
  }

  return r;
}
```



$pc = \text{true}$
$S(\mathtt{x}) = X$ $C(\mathtt{x}) = 1$
$C(\mathtt{r}) = 0$
$X > 8$
$S(\mathtt{x}) = X$ $C(\mathtt{x}) = 9$
$S(\mathtt{r}) = X - 7$ $C(\mathtt{r}) = 2$
$X \leq 8$
$S(\mathtt{x}) = X$ $C(\mathtt{x}) = 1$
$C(\mathtt{r}) = 0$
$X > 8 \wedge X \geq 5$
$S(\mathtt{x}) = X$ $C(\mathtt{x}) = 9$
$S(\mathtt{r}) = X - 7$ $C(\mathtt{r}) = 2$
$X \leq 8 \wedge X$
$S(\mathtt{x}) = X$
$S(\mathtt{r}) = X - 2$
$X \leq 8 \wedge X \geq 5$
$S(\mathtt{x}) = X$ $C(\mathtt{x}) = 6$
$C(\mathtt{r}) = 0$

New path condition:

$\neg(X > 8) \wedge \neg(X \geq 5)$ ✓ ✗

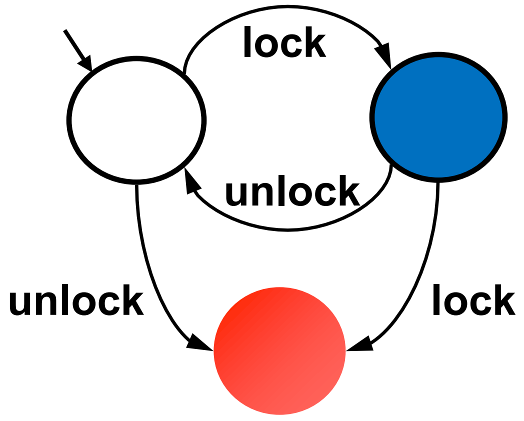
Test cases:

`proc(9)` `proc(1)` `proc(6)`

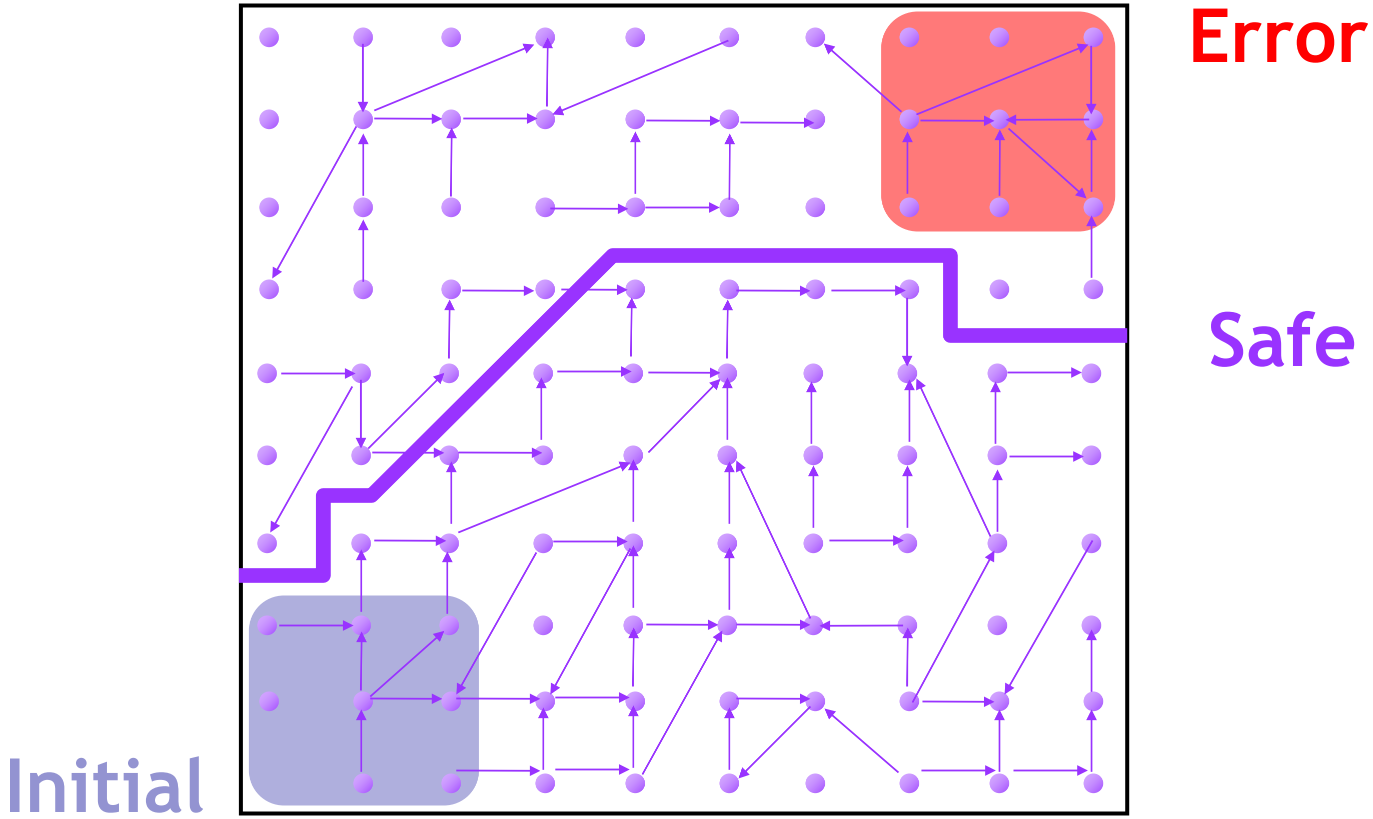
PREDICATE ABSTRACTION

Example Program

```
    example() {
1:    do {
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:        q->data = new;
          unlock();
          new ++;
        }
4:    } while(new != old);
5:    unlock();
      return;
    }
```



The Safety Verification Problem

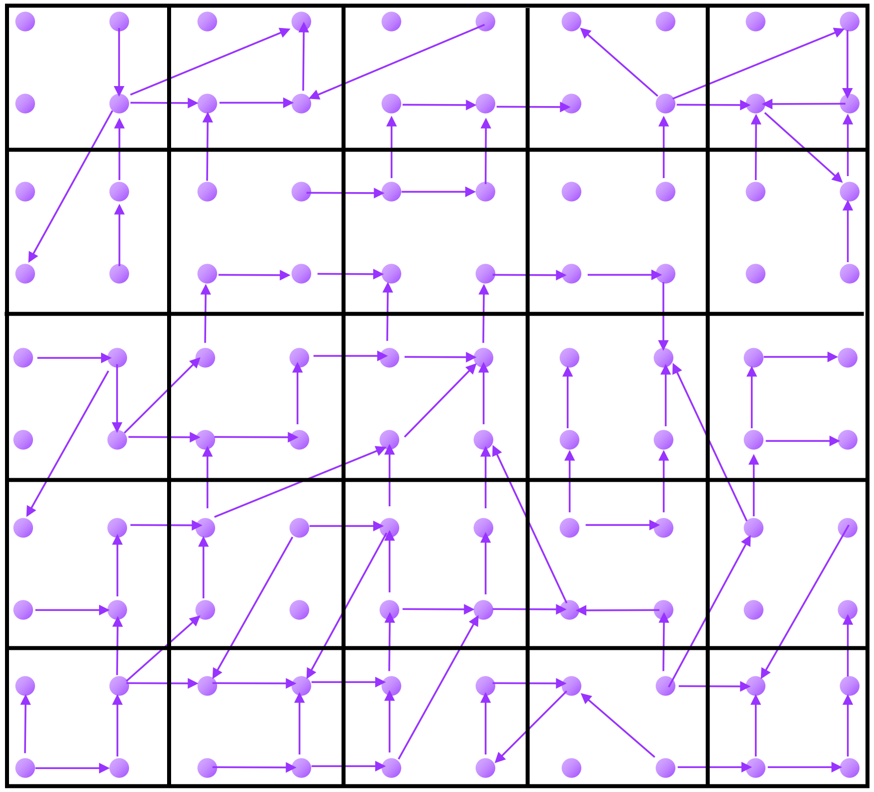


Is there a path from an initial to an error state?

Problem: Infinite state graph

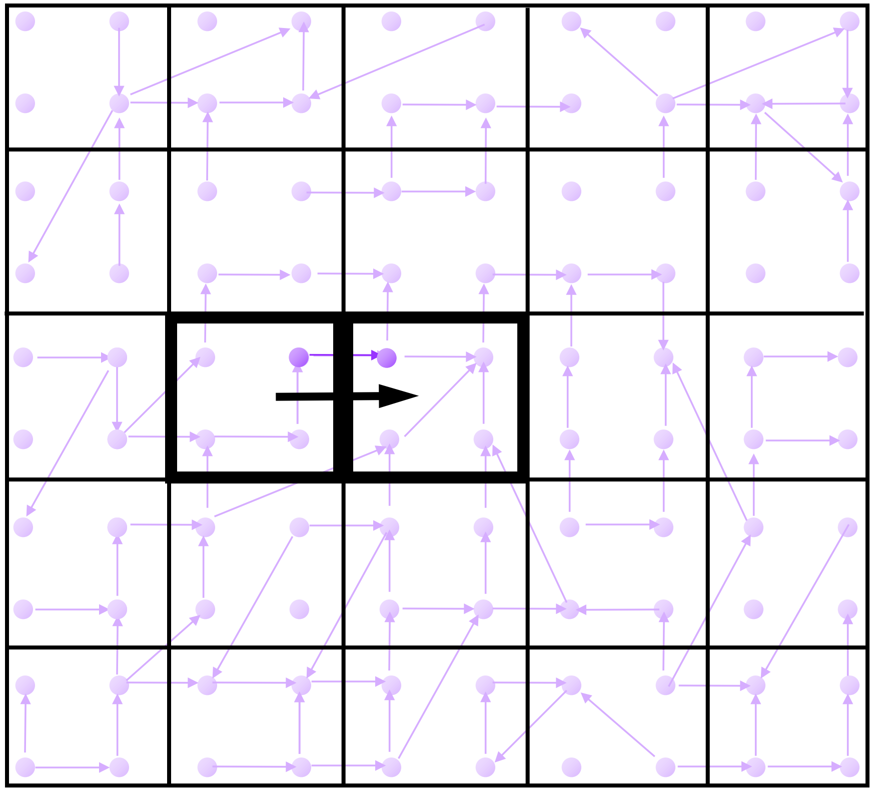
Solution: Set of states is a logical formula

Idea: Predicate Abstraction

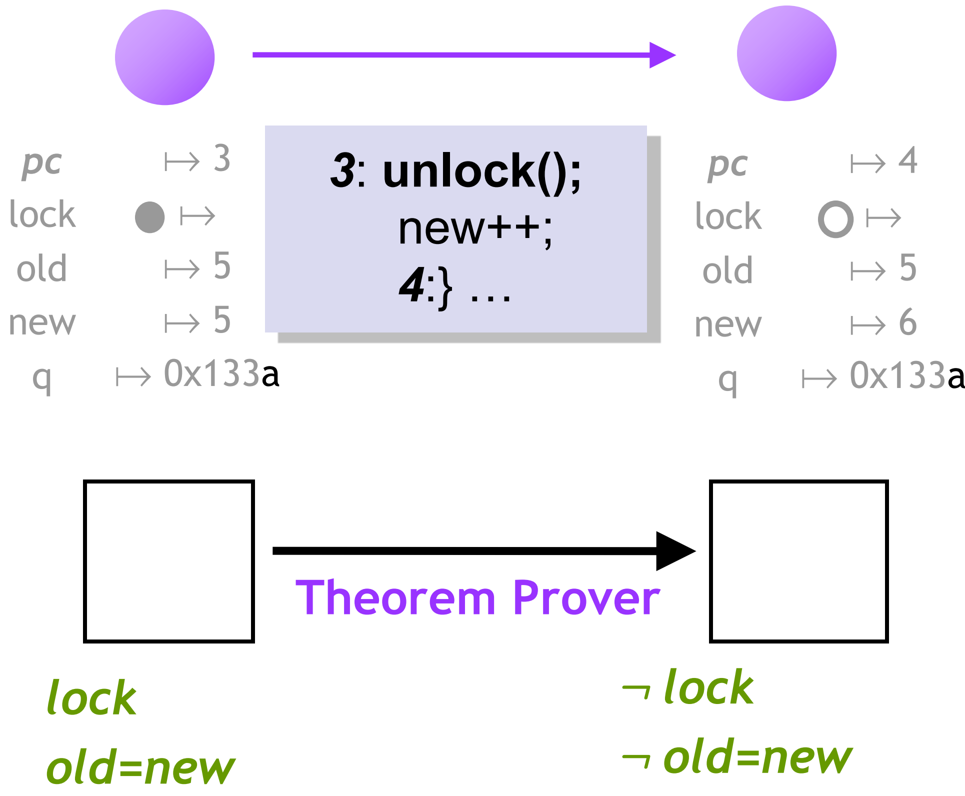


- Predicates on program state:
  `lock`
  `old = new`
- States satisfying same predicates are equivalent
  - Merged into one abstract state
- #abstract states is finite

Abstract States and Transitions



State

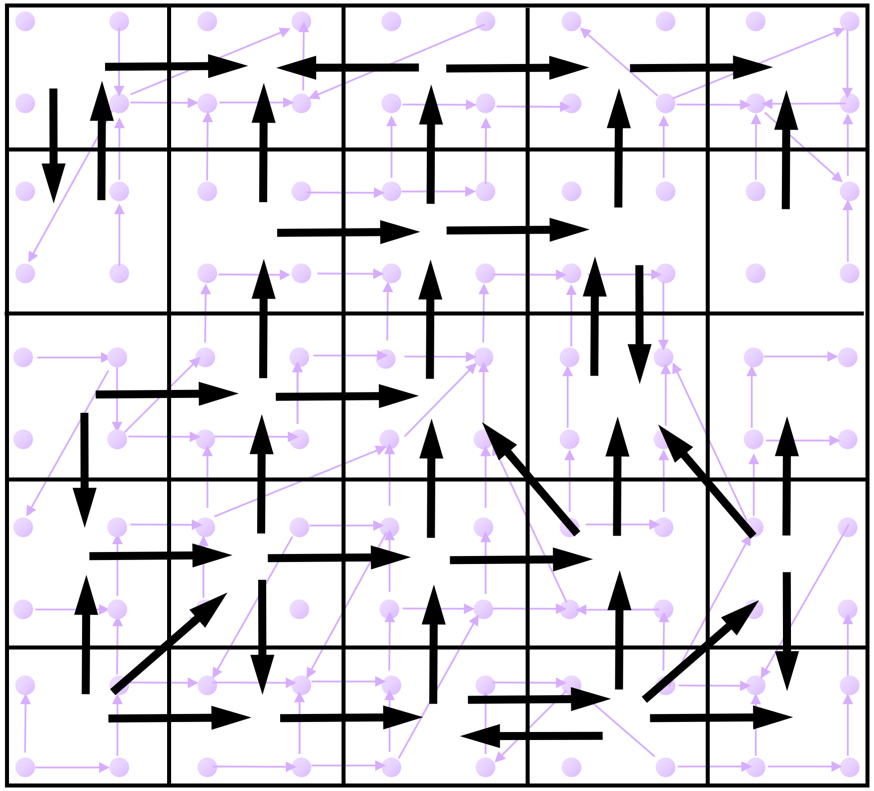


pc		↦ 3
lock	●	↦
old		↦ 5
new		↦ 5
q		↦ 0x133a

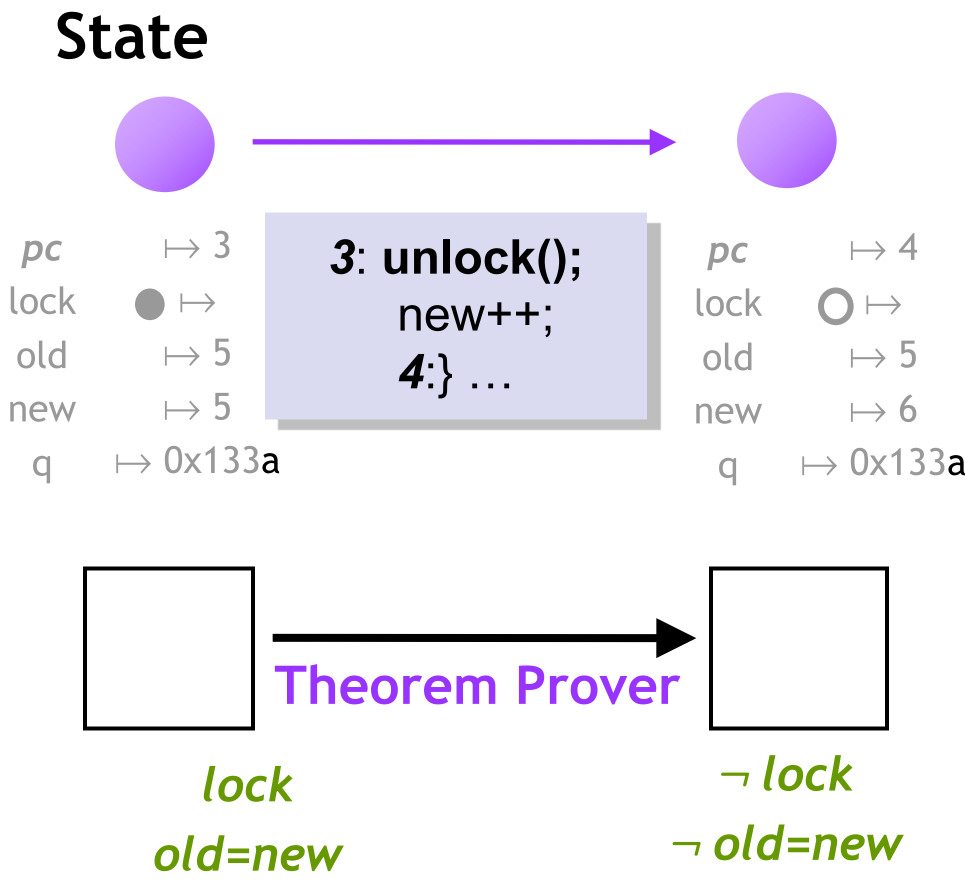
```
3: unlock();
     new++;
4:} …
```

pc		↦ 4
lock	○	↦
old		↦ 5
new		↦ 6
q		↦ 0x133a

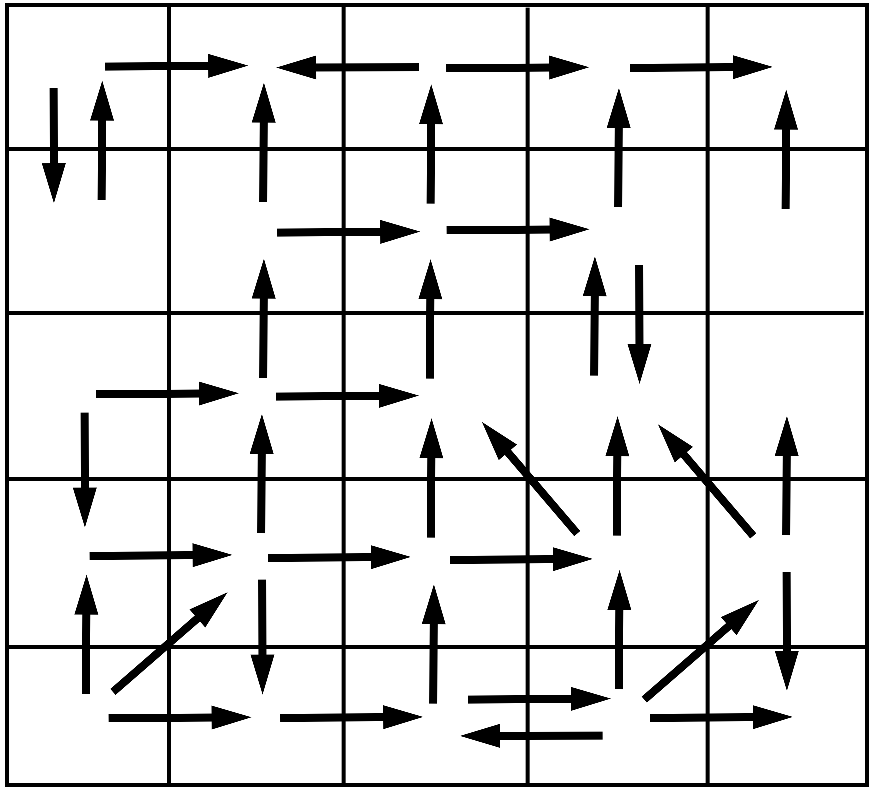
Abstraction

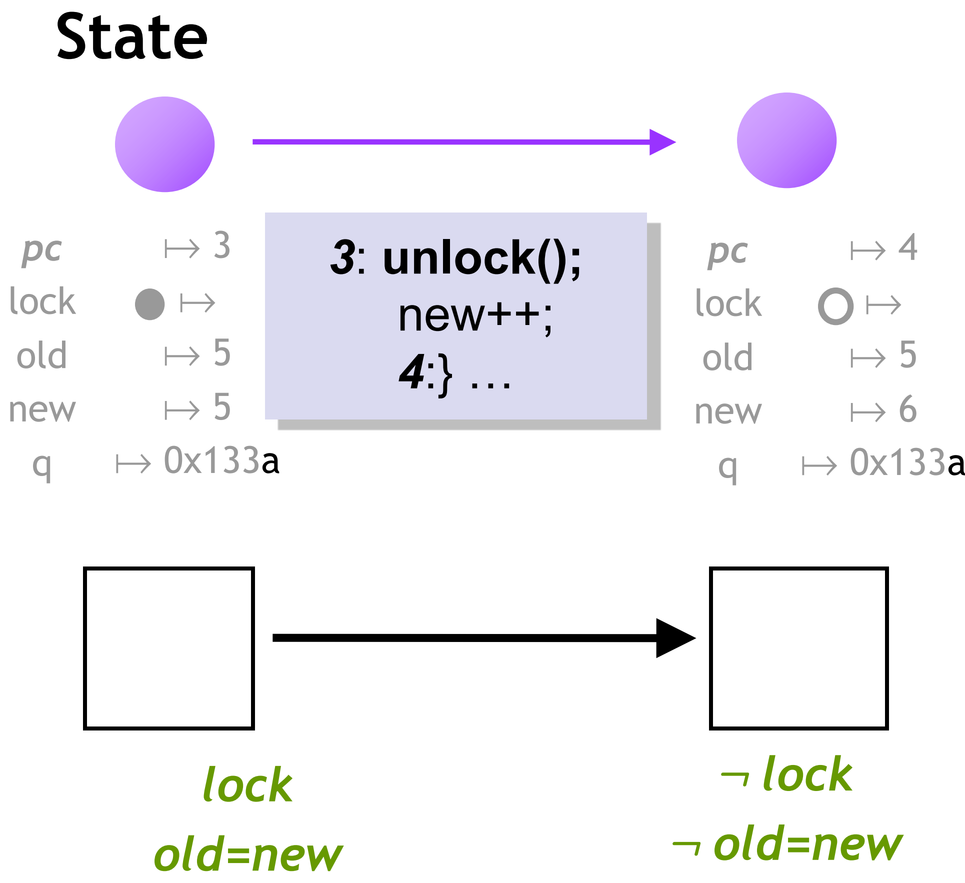


Existential Lifting

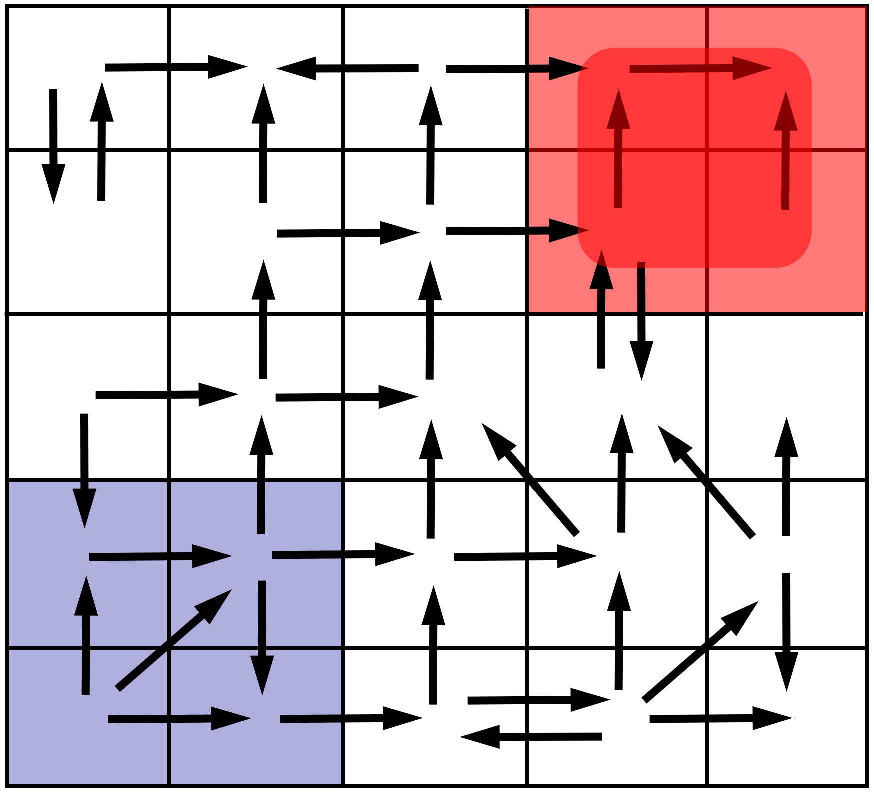


Abstraction





Analyze Abstraction



Analyzing finite graph

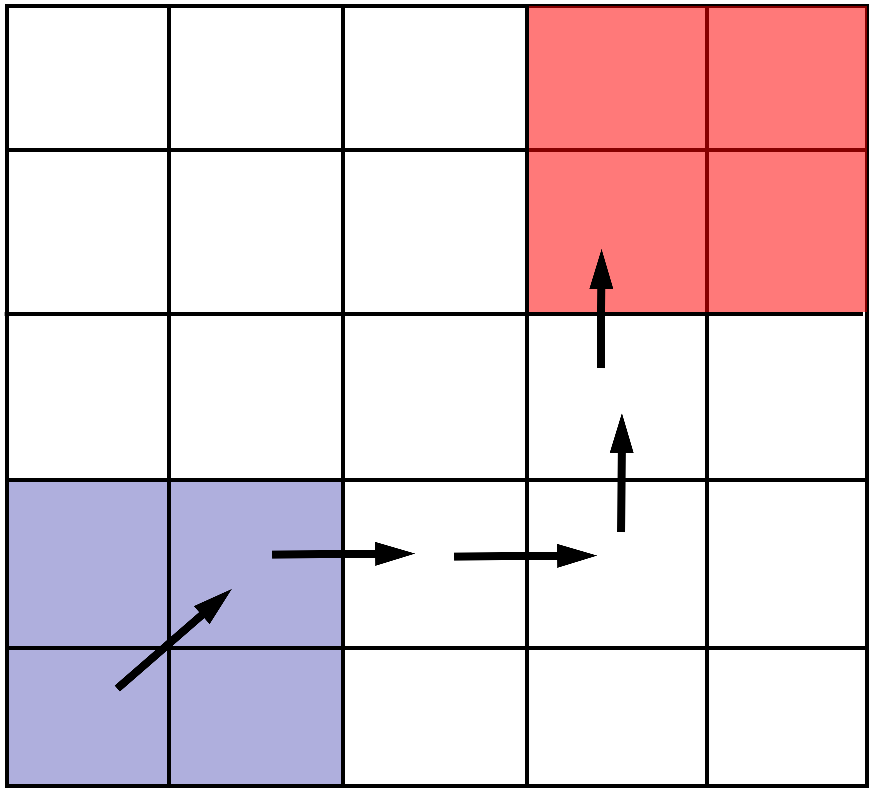
Over-approximate:

- Safe means that system is safe
- No false negatives

Problem:

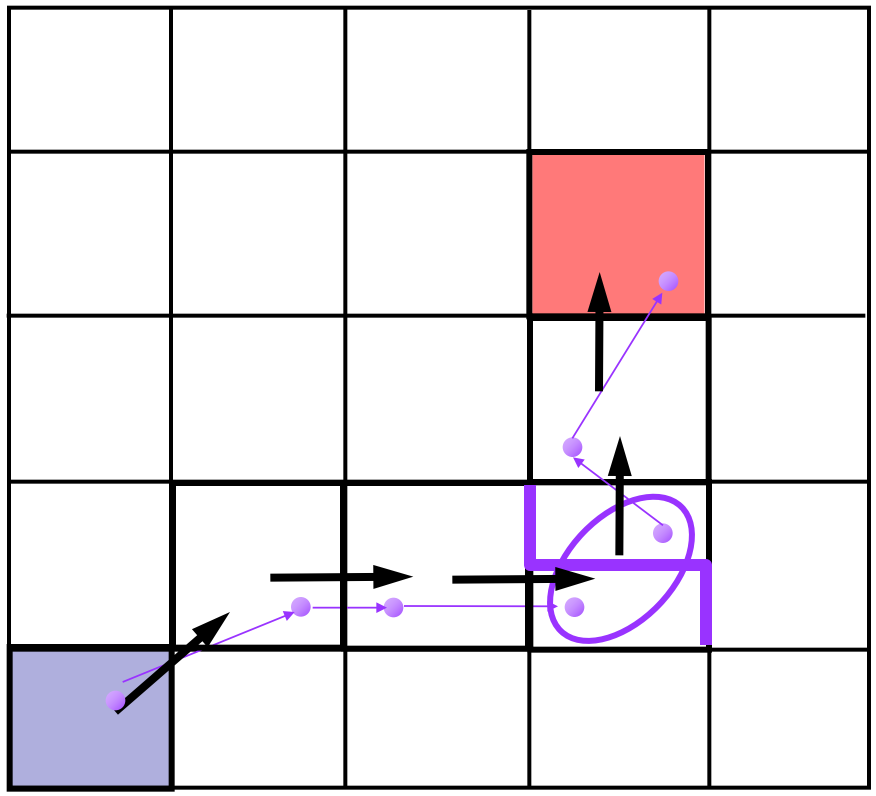
Spurious counterexamples

Idea: Counterex.-Guided Refinement



Solution:
Use spurious counterexamples to refine abstraction

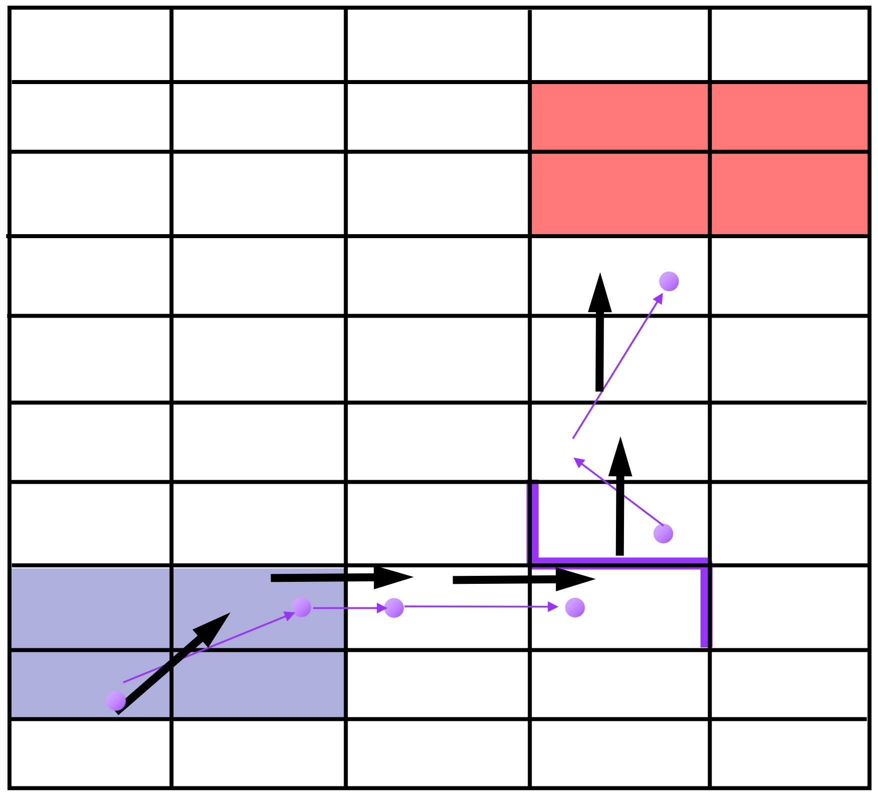
Idea: Counterex.-Guided Refinement



Solution:
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut

Iterative Abstraction Refinement



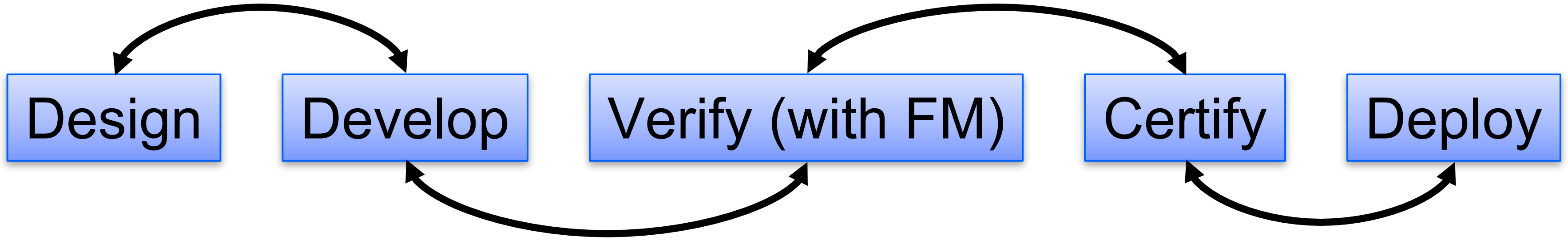
Solution:
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut
2. Build refined abstraction
- eliminates counterexample
3. Repeat search
- till real counterexample or system proved safe

TRUST IN FORMAL METHODS

UNIVERSITY OF WATERLOO

Idealized Development w/ Formal Methods



No expensive testing!

- Verification is exhaustive

Simpler certification!

- Just check formal arguments

Can we trust formal methods tools? What can go wrong?

Trusting Automated Verification Tools

How should automatic verifiers be qualified for certification?

What is the basis for automatic program analysis (or other automatic formal methods) to replace testing?

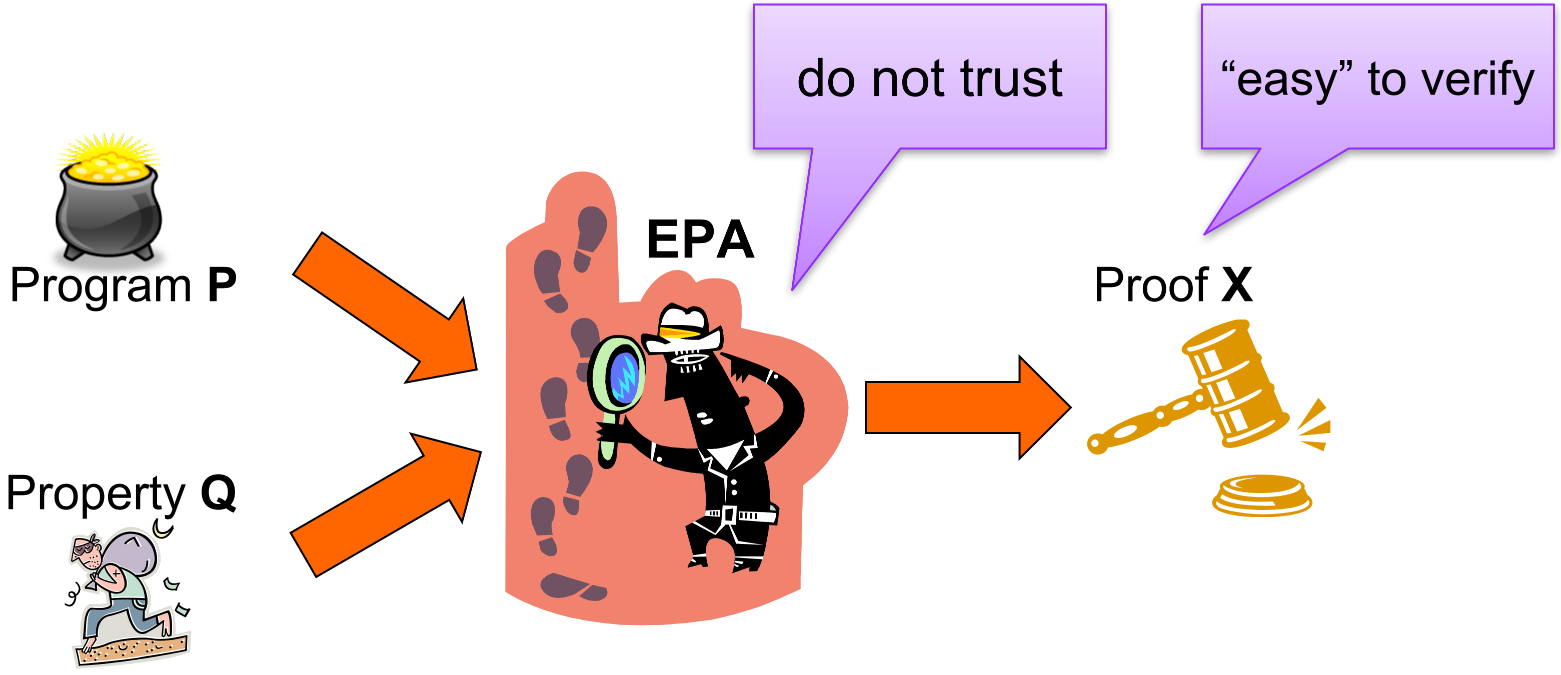
Verify the verifier

- (too) expensive
- verifiers are often very complex tools
- difficult to continuously adapt tools to project-specific needs

Proof-producing (or certifying) verifier

- Only the proof is important – not the tool that produced it
- Only the proof-checker needs to be verified/qualified
- Single proof-checker can be re-used in many projects

Evidence Producing Analysis



X witnesses that **P** satisfies **Q**. **X** can be objectively and independently verified.
Therefore, **EPA** is outside the Trusted Computing Base (**TCB**).

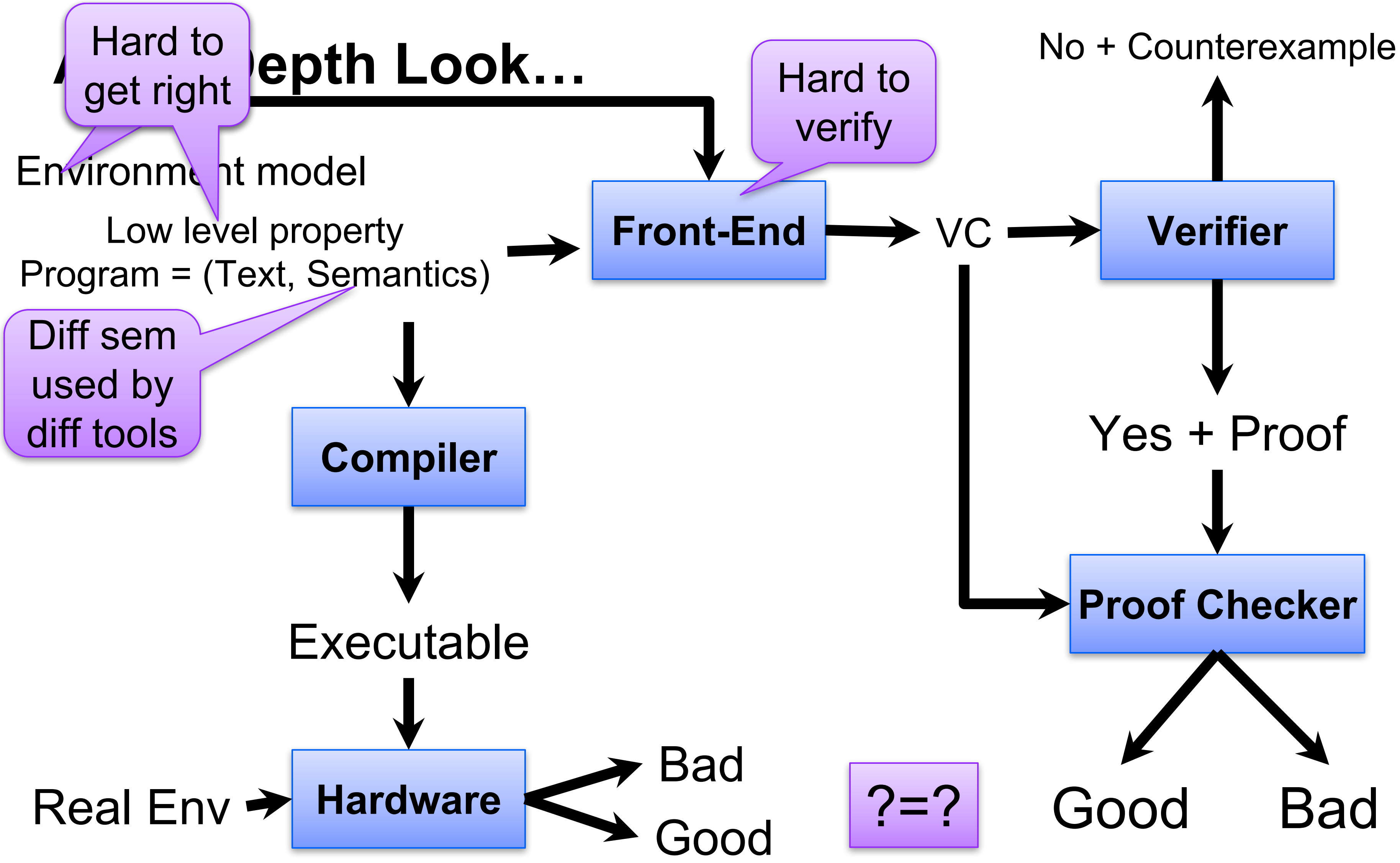
Active research area

- proof carrying code, certifying model checking, model carrying code etc.
- Few tools available. Some preliminary commercial application in the telecom domain.
- **Static** context. Good for **ensuring absence of problems.**
- Low automation. Applies to source or binary. High confidence.

Not that simple in practice !!!

UNIVERSITY OF WATERLOO

A Depth Look…


Hard to get right
Environment model
Low level property
Program = (Text, Semantics)
Diff sem used by diff tools
Front-End
Hard to verify
VC
Verifier
No + Counterexample
Yes + Proof
Proof Checker
Good
Bad
Compiler
Executable
Real Env
Hardware
Bad
Good
?=?

Five Hazards (Gaps) of Automated Verification

Soundness Gap

- Intentional and unintentional unsoundness in the verification engine
- e.g., rational instead of bitvector arithmetic, simplified memory model, etc.

Semantic Gap

- Compiler and verifier use different interpretation of the programming language

Specification Gap

- Expressing high-level specifications by low-level verifiable properties

Property Gap

- Formalizing low-level properties in temporal logic and/or assertions

Environment Gap

- Too coarse / unsound / unfaithful model of the environment

UNIVERSITY OF WATERLOO

Mitigating The Soundness Gap

Proof-producing verifier makes the soundness gap explicit

- the soundness of the proof can be established by a “simple” checker
- all assumptions are stated explicitly

Open questions:

- how to generate proofs for explicit Model Checking
  - e.g., SPIN, Java PathFinder
- how to generate partial proofs for non-exhaustive methods
  - e.g., KLEE, Sage
- how to deal with “intentional” unsoundness
  - e.g., rational arithmetic instead of bitvectors, memory models, …

Vacuity: Mitigating Property Gap

Model Checking Perspective: Never trust a *True* answer from a Model Checker

When a property is violated, a counterexample is a certificate that can be examined by the user for validity

When a property is satisfied, there is no feedback!

It is very easy to formally state something very trivial in a very complex way

```
MODULE main

VAR
  send : {s0,s1,s2};
  recv : {r0,r1,r2};

  ack : boolean;
  req : boolean;

ASSIGN
 init(ack):=FALSE;
 init(req):=FALSE;

 init(send):= s0;
 init(recv):= r0;
```

```
next (send) :=
    case
      send=s0:{s0,s1};
      send=s1:s2;
      send=s2&ack:s0;
      TRUE:send;
    esac;

  next (recv) :=
    case
      recv=r0&req:r1;
      recv=r1:r2;
      recv=r2:r0;
      TRUE: recv;
    esac;
```

```
next (ack) :=
   case
     recv=r2:TRUE;
     TRUE: ack;
   esac;

  next (req) :=
    case
      send=s1:FALSE;
      TRUE: req;
    esac;
```

```
SPEC AG (req -> AF ack)
```

Five Hazards (Gaps) of Automated Verification

Soundness Gap

- Intentional and unintentional unsoundness in the verification engine
- e.g., rational instead of bitvector arithmetic, simplified memory model, etc.

Semantic Gap

- Compiler and verifier use different interpretation of the programming language

Specification Gap

- Expressing high-level specifications by low-level verifiable properties

Property Gap

- Formalizing low-level properties in temporal logic and/or assertions

Environment Gap

- Too coarse / unsound / unfaithful model of the environment