Software Model Checking

Testing, Quality Assurance, and Maintenance
Winter 2017

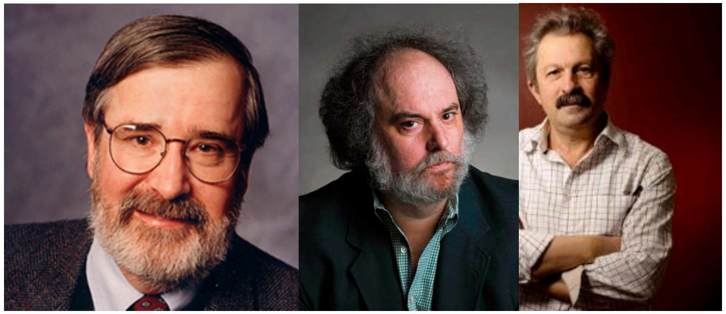
Prof. Arie Gurfinkel

UNIVERSITY OF WATERLOO

(Temporal Logic) Model Checking

Automatic verification technique for finite state concurrent systems.

- Developed independently by Clarke and Emerson and by Queille and Sifakis in early 1980's.
- ACM Turing Award 2007



Specifications are written in propositional temporal logic. (Pnueli 77)

- Computation Tree Logic (CTL), Linear Temporal Logic (LTL), …

Verification procedure is an intelligent exhaustive search of the state space of the design

- State-space explosion

Model Checking since 1981

1981 Clarke / Emerson: CTL Model Checking 10^5
Sifakis / Quielle

1982 EMC: Explicit Model Checker
Clarke, Emerson, Sistla

1990 Symbolic Model Checking 10^{100}
Burch, Clarke, Dill, McMillan

1992 SMV: Symbolic Model Verifier
McMillan

1990s: Formal Hardware Verification in Industry: Intel, IBM, Motorola, etc.

1998 Bounded Model Checking using SAT 10^{1000}
Biere, Clarke, Zhu

2000 Counterexample-guided Abstraction Refinement
Clarke, Grumberg, Jha, Lu, Veith

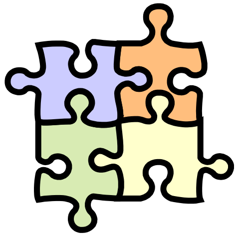
Model Checking since 1981

1981 Clarke / Emerson: CTL Model Checking
Sifakis / Quielle

1982 EMC: Explicit Model Checker
Clarke, Emerson, Sistla

1990 Symbolic Model Checking
Burch, Clarke, Dill, McMillan

1992 SMV: Symbolic Model Verifier
McMillan



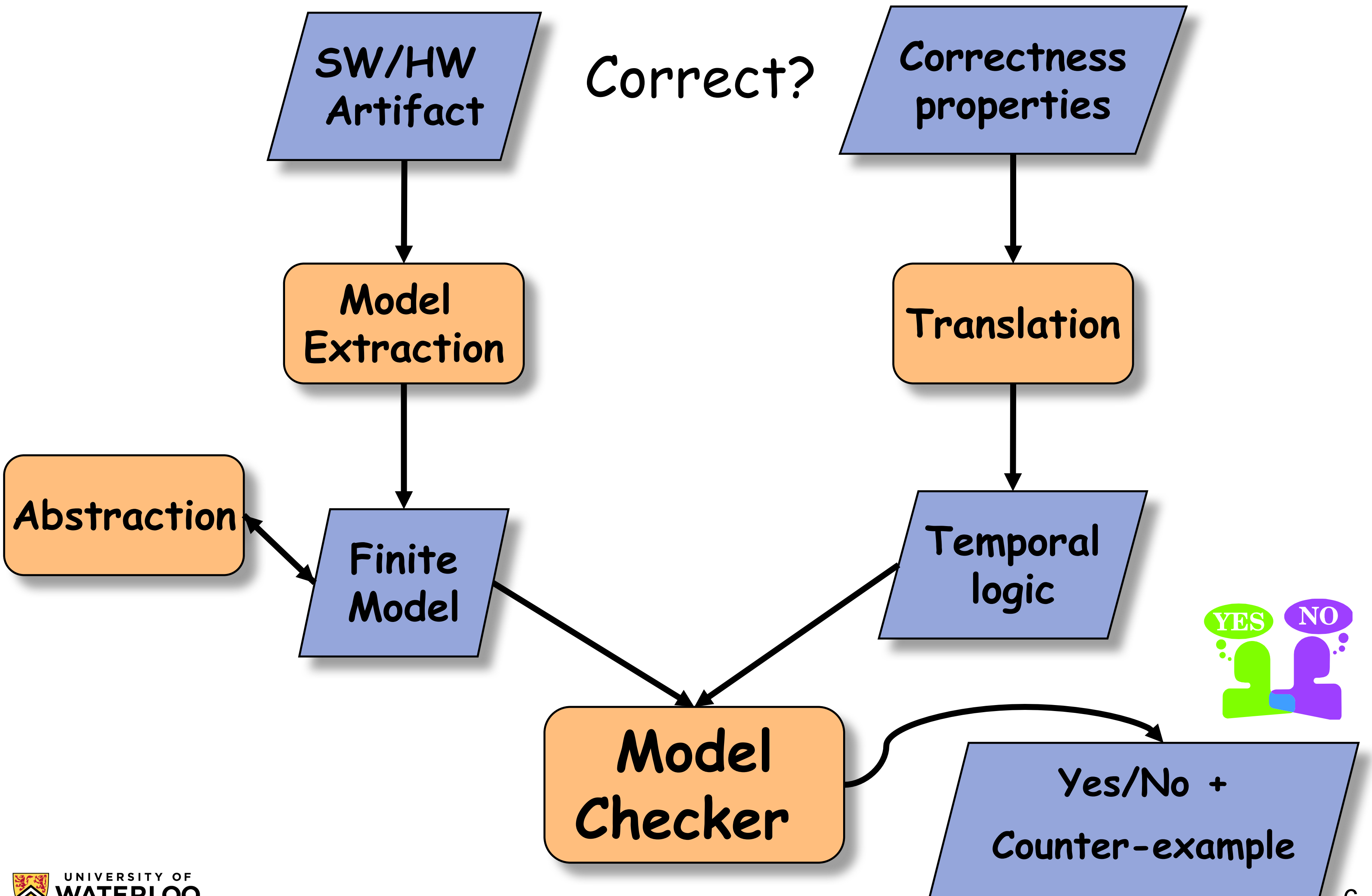
1998 Bounded Model Checking using SAT
Biere, Clarke, Zhu

→ CBMC

2000 Counterexample-guided Abstraction Refinement
Clarke, Grumberg, Jha, Lu, Veith

→ SLAM, MAGIC, BLAST, …

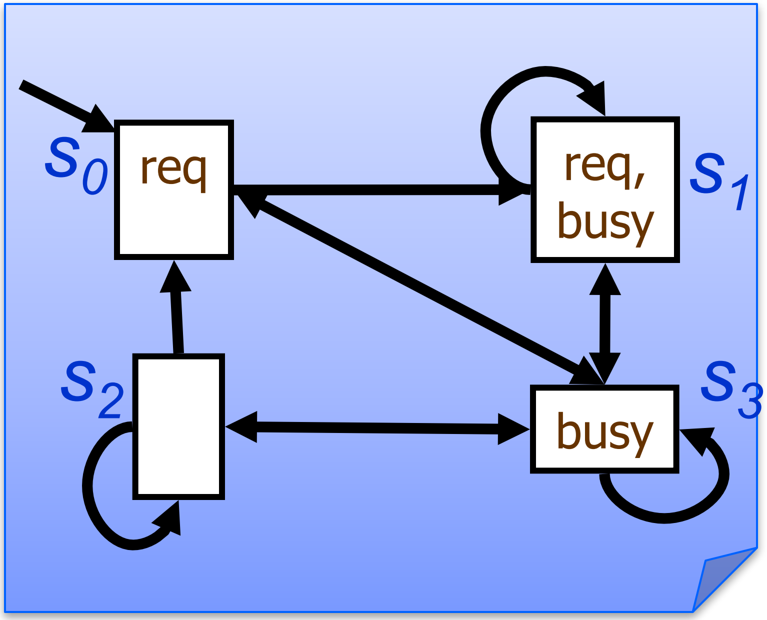
Temporal Logic Model Checking



Models: Kripke Structures

Conventional state machines

- $K = (V, S, s_0, I, R)$
- V is a (finite) set of atomic propositions
- S is a (finite) set of states
- $s_0 \in S$ is a start state
- $I: S \to 2^V$ is a labelling function that maps each state to the set of propositional variables that hold in it
  - That is, $I(S)$ is a set of interpretations specifying which propositions are true in each state
- $R \subseteq S \times S$ is a transition relation



Propositional Variables

Fixed set of atomic propositions, e.g, *{p, q, r}*

Atomic descriptions of a system

- “Printer is busy”
- “There are currently no requested jobs for the printer”
- “Conveyer belt is stopped”

Do not involve time!

Modal Logic

Extends *propositional logic* with modalities to qualify propositions

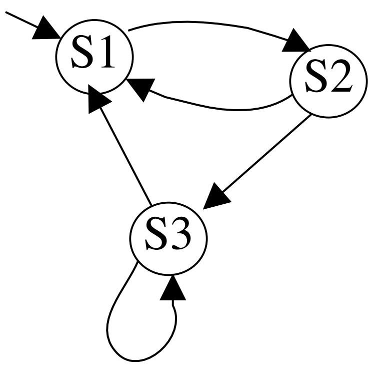
- *"it is raining"* – *rain*
- "it will rain tomorrow" – $\Box$*rain*
  - it is raining in all possible futures
- "it might rain tomorrow" – $\Diamond$*rain*
  - it is raining in some possible futures

Modal logic formulas are interpreted over a collection of *possible worlds* connected by an *accessibility relation*

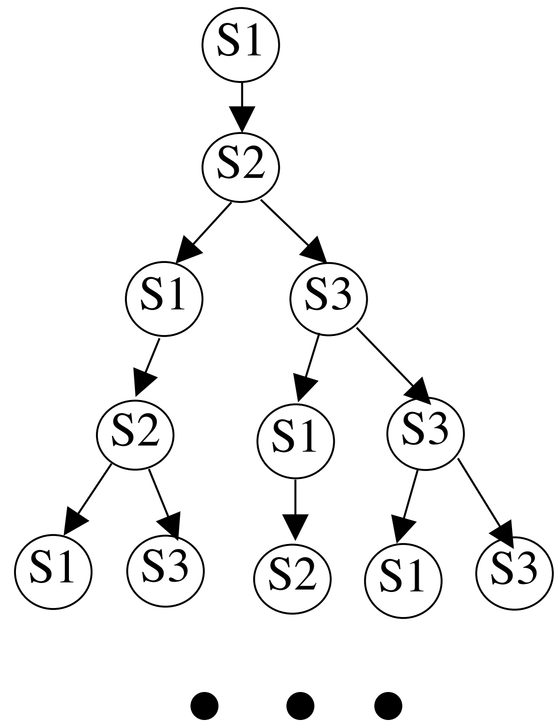
Temporal logic is a modal logic that adds temporal modalities: next, always, eventually, and until

Computation Tree Logic (CTL)

CTL: Branching-time propositional temporal logic
Model - a tree of computation paths



Kripke Structure



Tree of computation

CTL: Computation Tree Logic

Propositional temporal logic with explicit quantification over possible futures

Syntax:

True and *False* are CTL formulas;
propositional variables are CTL formulas;

If φ and ψ are CTL formulae, then so are: $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$

EX φ : φ holds in some next state

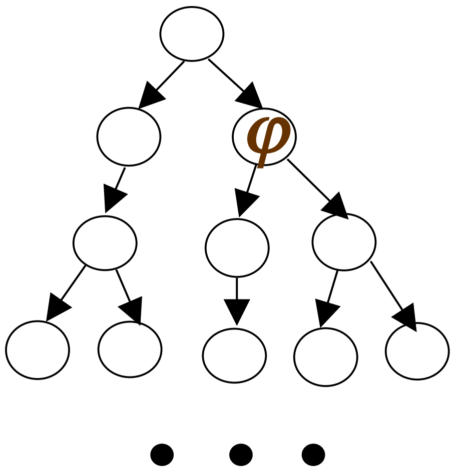
EF φ : along some path, φ holds in a future state

E[φ U ψ] : along some path, φ holds until ψ holds

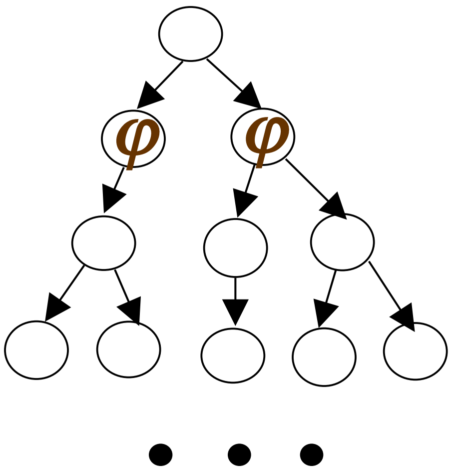
EG φ : along some path, φ holds in every state

- Universal quantification: AX φ , AF φ , A[φ U ψ], AG φ

Examples: EX and AX

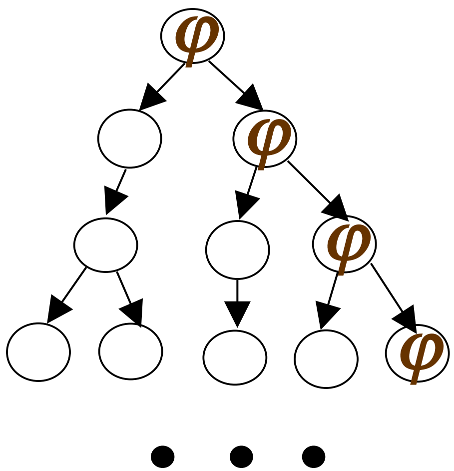


EX φ (exists next)

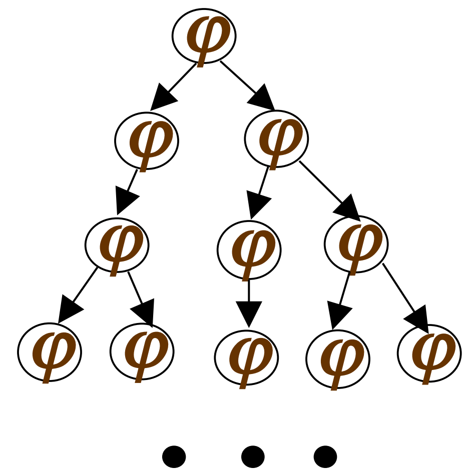


AX φ (all next)

Examples: EG and AG

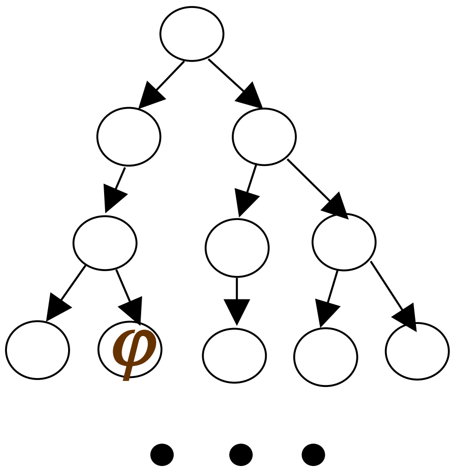


EG φ (exists global)

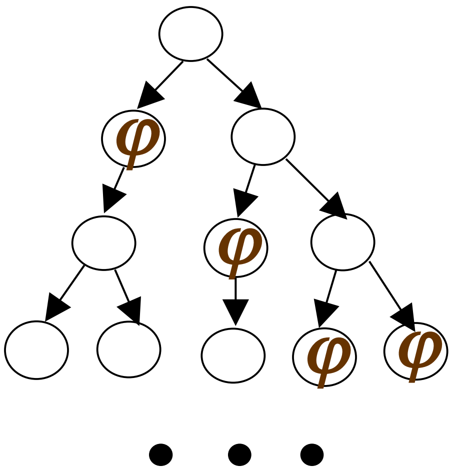


AG φ (all global)

Examples: EF and AF

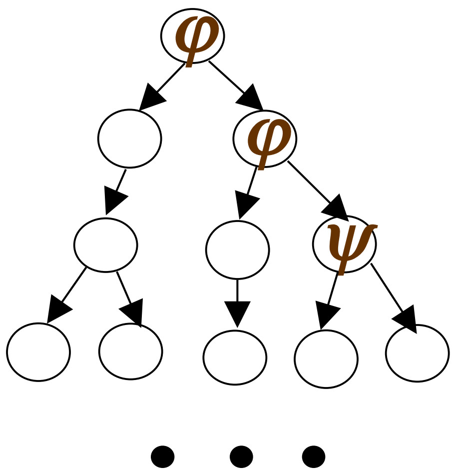


EF φ (exists future)

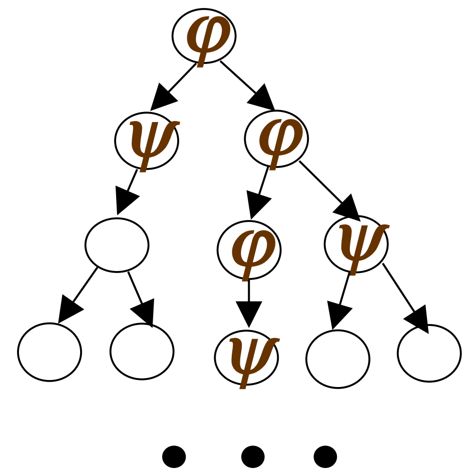


AF φ (all future)

Examples: EU and AU



E[φ U ψ] (exists until)



A[φ U ψ] (all until)

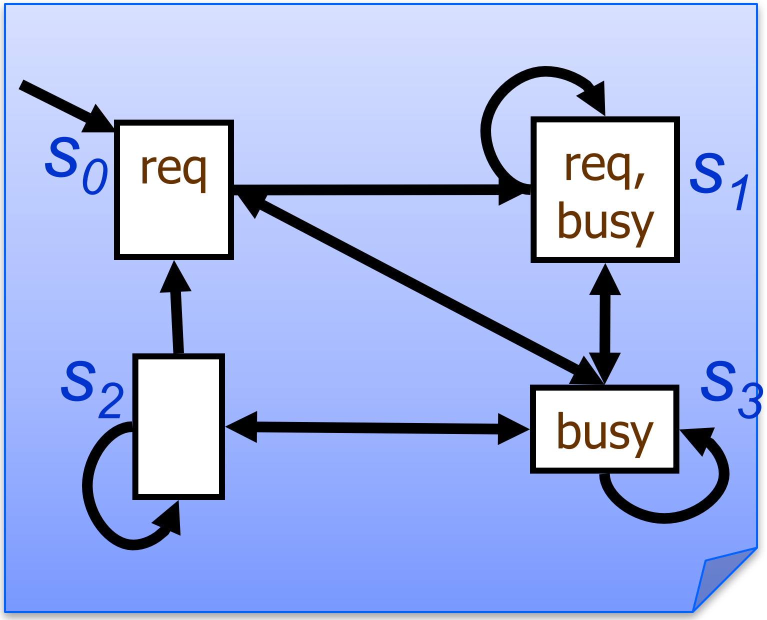
CTL Examples

Properties that hold:

- (AX busy)(s_0)
- (EG busy)(s_3)
- A (req U busy) (s_0)
- E ($\neg$req U busy) (s_1)
- AG (req $\Rightarrow$ AF busy) (s_0)

Properties that fail:

- (AX (req v busy))(s_3)



Some Statements To Express

An elevator can remain idle on the third floor with its doors closed

- EF (state=idle ∧ floor=3 ∧ doors=closed)

When a request occurs, it will eventually be acknowledged

- AG (request ⇒ AF acknowledge)

A process is enabled infinitely often on every computation path

- AG AF enabled

A process will eventually be permanently deadlocked

- AF AG deadlock

Action *s* precedes *p* after *q*

- A[¬*q* U (*q* ∧ A[¬*p* U *s*])]
- Note: hard to do correctly. Use property patterns

Semantics of CTL

$K,s \models \varphi$ – means that formula φ is true in state s. K is often omitted since we always talk about the same Kripke structure

- E.g., $s \models p \wedge \neg q$

$\pi = \pi^0 \pi^1 \ldots$ is a path

π^0 is the current state (root)

π^{i+1} is a successor state of π^i. Then,

AX $\varphi = \forall \pi \cdot \pi^1 \models \varphi$

AG $\varphi = \forall \pi \cdot \forall i \cdot \pi^i \models \varphi$

AF $\varphi = \forall \pi \cdot \exists i \cdot \pi^i \models \varphi$

EX $\varphi = \exists \pi \cdot \pi^1 \models \varphi$

EG $\varphi = \exists \pi \cdot \forall i \cdot \pi^i \models \varphi$

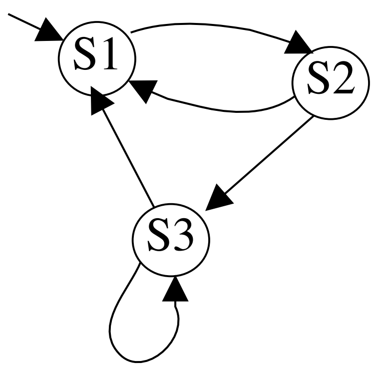
EF $\varphi = \exists \pi \cdot \exists i \cdot \pi^i \models \varphi$

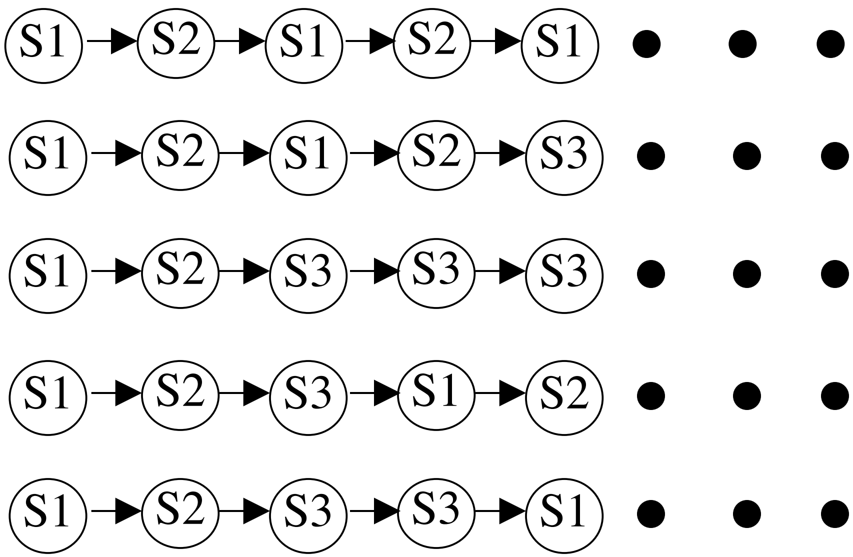
A[φ U ψ] $= \forall \pi \cdot \exists i \cdot \pi^i \models \psi \wedge \forall j \cdot 0 \leq j < i \Rightarrow \pi^j \models \varphi$

E[φ U ψ] $= \exists \pi \cdot \exists i \cdot \pi^i \models \psi \wedge \forall j \cdot 0 \leq j < i \Rightarrow \pi^j \models \varphi$

Linear Temporal Logic (LTL)

For reasoning about complete traces through the system





Allows to make statements about a trace

LTL Syntax

If φ is an atomic propositional formula, it is a formula in LTL

If φ and ψ are LTL formulas, so are $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg \varphi$, φ U ψ (until), X φ (next), Fφ (eventually), G φ (always)

Interpretation: over computations $\pi: \omega \Rightarrow 2^V$ which assigns truth values to the elements of V at each time instant

$\pi \models \mathrm{X}\ \varphi$ iff $\pi^{1} \models \varphi$

$\pi \models \mathrm{G}\ \varphi$ iff $\forall i \cdot \pi^{i} \models \varphi$

$\pi \models \mathrm{F}\varphi$ iff $\exists i \cdot \pi^{i} \models \varphi$

$\pi \models \varphi\ \mathrm{U}\ \psi$ iff $\exists i \cdot \pi^{i} \models \psi \wedge \forall j \cdot 0 \leq j < i \Rightarrow \pi^{j} \models \varphi$

Here, π^{i} is the i'th state on a path

UNIVERSITY OF WATERLOO

Expressing Properties in LTL

Good for safety (G $\neg$) and liveness (F) properties

Express:

- When a request occurs, it will eventually be acknowledged
  - G (request $\Rightarrow$ F acknowledge)
- Each path contains infinitely many *q*'s
  - G F *q*
- At most a finite number of states in each path satisfy $\neg q$ (or property *q* eventually stabilizes)
  - F G *q*
- Action *s* precedes *p* after *q*
  - $[\neg q \text{ U } (q \wedge [\neg p \text{ U } s])]$
  - Note: hard to do correctly.

Safety and Liveness

Safety: Something “bad” will never happen

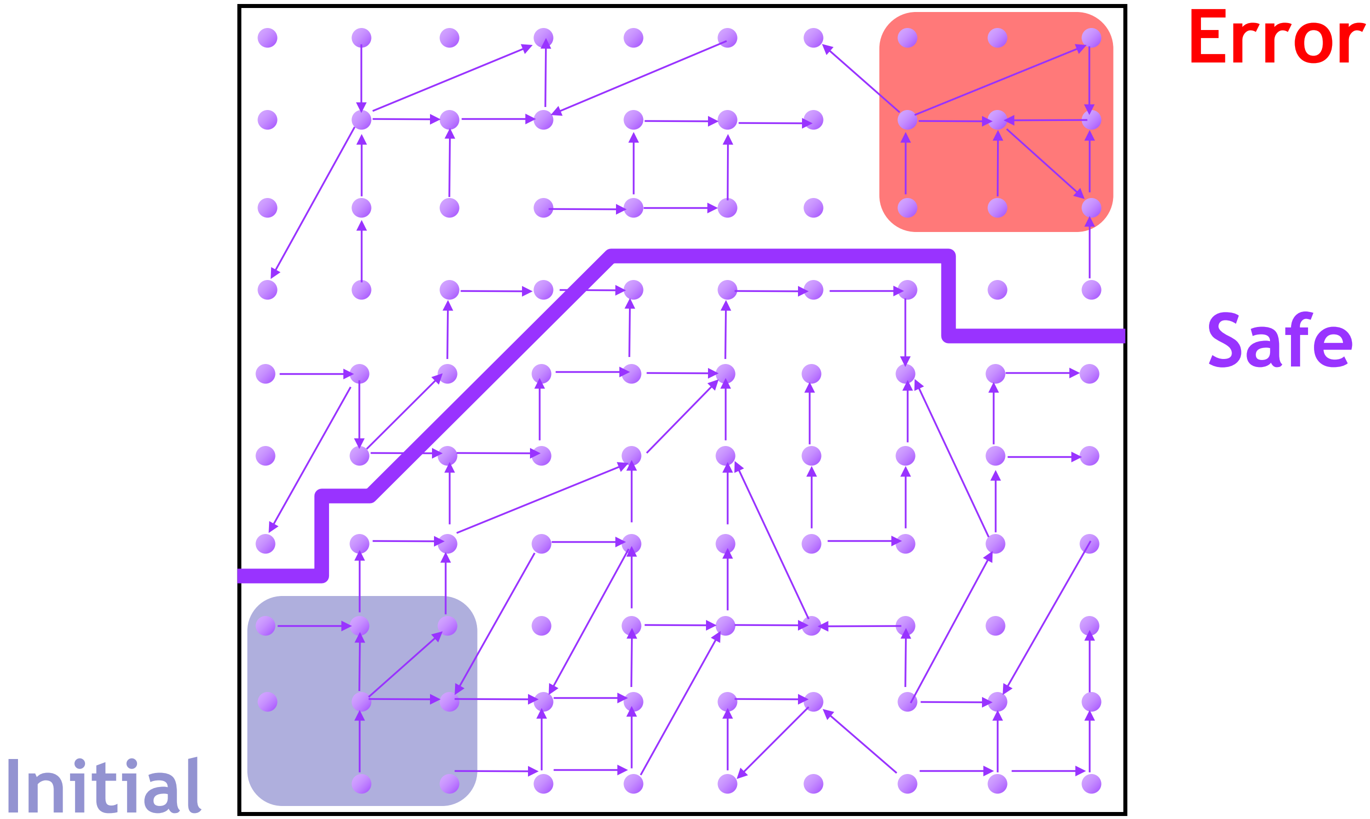
- AG ¬bad
- e.g., mutual exclusion: no two processes are in their critical section at once
- Safety = if false then there is a finite counterexample
- Safety = reachability

Liveness: Something “good” will always happen

- AG AF good
- e.g., every request is eventually serviced
- Liveness = if false then there is an infinite counterexample
- Liveness = termination

Every universal temporal logic formula can be decomposed into a conjunction of safety and liveness

The Safety Verification Problem



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

State Explosion

How fast do Kripke structures grow?

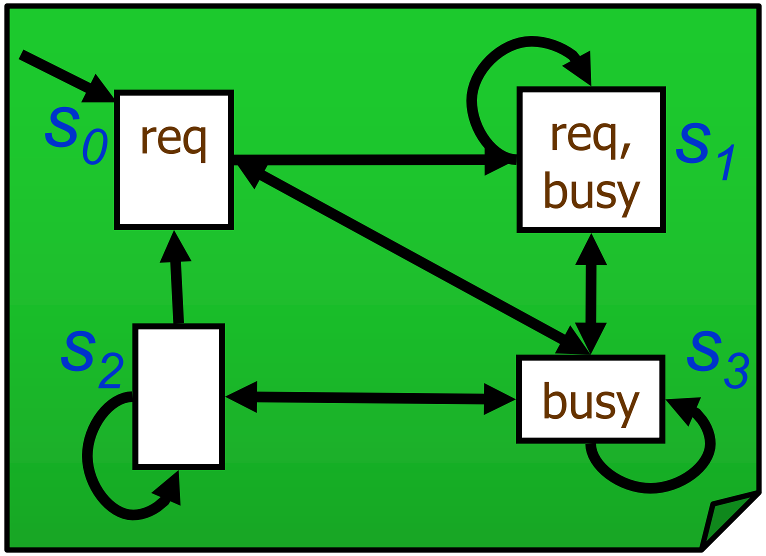
- Composing linear number of structures yields exponential growth!

How to deal with this problem?

- Symbolic model checking with efficient data structures (BDDs, SAT).
  - Do not need to represent and manipulate the entire model
- Abstraction
  - Abstract away variables in the model which are not relevant to the formula being checked
  - Partial order reduction (for asynchronous systems)
  - Several interleavings of component traces may be equivalent as far as satisfaction of the formula to be checked is concerned
- Composition
  - Break the verification problem down into several simpler verification problems

Representing Models Symbolically

A system state represents an interpretation (truth assignment) for a set of propositional variables V



- Formulas represent sets of states that satisfy it
  - False = ∅, True = S
  - req – set of states in which req is
  - true – {s0, s1}
  - busy – set of states in which busy is
  - true – {s1, s3}
  - req ∨ busy = {s0, s1 , s3}
- State transitions are described by relations over two sets of variables: V (source state) and V' (destination state)
  - Transition (s2, s3) is ¬req ∧ ¬ busy ∧ ¬req' ∧ busy'
  - Relation R is described by disjunction of formulas for individual transitions

Pros and Cons of Model-Checking

Often cannot express full requirements

- Instead check several smaller simpler properties

Few systems can be checked directly

- Must generally abstract parts of the system and model the environment

Works better for certain types of problems

- Very useful for control-centered concurrent systems
  - Avionics software
  - Hardware
  - Communication protocols
- Not very good at data-centered systems
  - User interfaces, databases

Pros and Cons of Model Checking (Cont'd)

Largely automatic and fast

Better suited for debugging

- … rather than assurance

Testing vs model-checking

- Usually, find more problems by

  exploring all behaviors of a downscaled system

  than by

  testing some behaviors of the full system

Software Model Checking Workflow

1. Identify module to be analyzed
   - e.g., function, component, device driver, library, etc.
2. Instrument with property assertions
   - e.g., buffer overflow, proper API usage, proper state change, etc.
   - might require significant changes in the program to insert necessary monitors
3. Model environment of the module under analysis
   - provide stubs for functions that are called but are not analyzed
4. Write verification harness that exercises module under analysis
   - similar to unit-test, but can use symbolic values
   - tests many executions at a time
5. Run Model Checker

6. Repeat as needed

Types of Software Model Checking

Bounded Model Checking (BMC)

- look for bugs (bad executions) up to a fixed bound
- usually bound depth of loops and depth of recursive calls
- reduce the problem to SAT/SMT

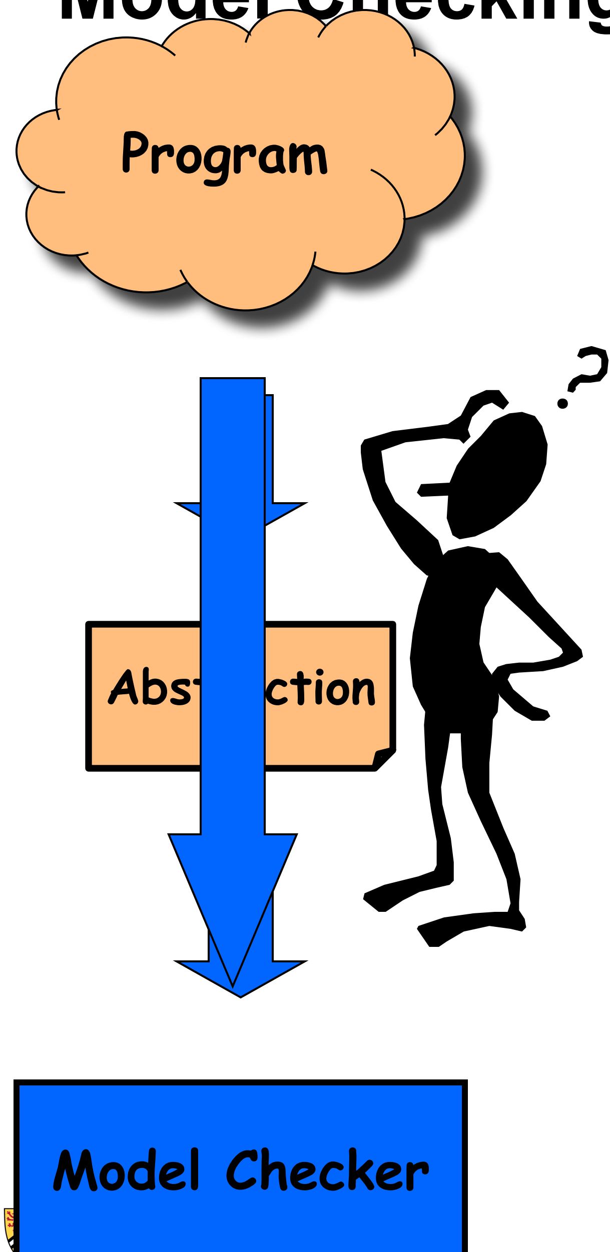
Predicate Abstraction with CounterExample Guided Abstraction Refinement (CEGAR)

- Construct finite-state abstraction of a program
- Analyze using finite-state Model Checking techniques
- Automatically improve / refine abstraction until the analysis is conclusive

Interpolation-based Model Checking (IMC)

- Iteratively apply BMC with increasing bound
- Generalize from bounded-safety proofs
- reduce the problem to many SAT/SMT queries and generalize from SAT/SMT reasoning

Model Checking Software by Abstraction



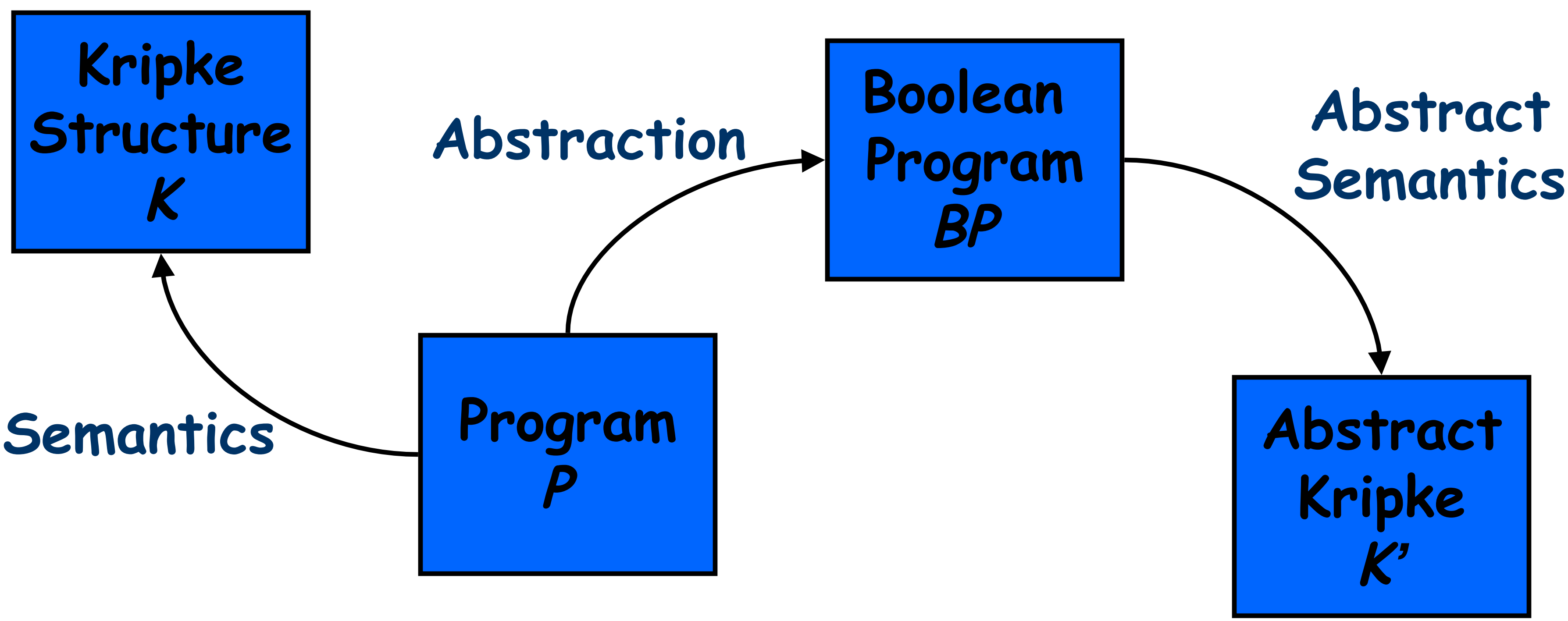
Programs are not finite state

- integer variables
- recursion
- unbounded data structures
- dynamic memory allocation
- dynamic thread creation
- pointers
- …

➲ **Build a finite abstraction**

- **… small enough to analyze**
- **… rich enough to give conclusive results**

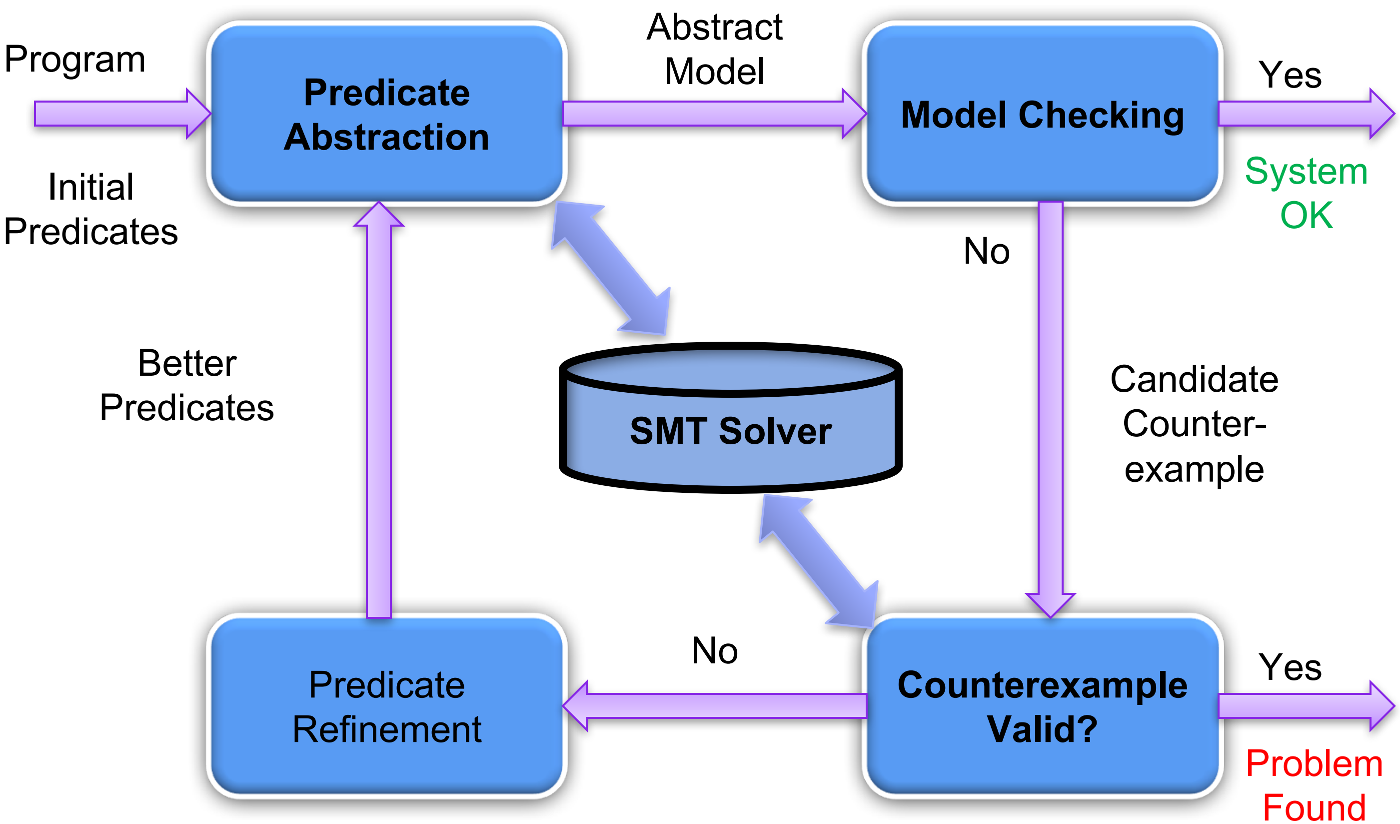
Software Model Checking and Abstraction



Soundness of Abstraction:

BP abstracts *P* implies that *K'* approximates *K*

CounterExample Guided Abstraction Refinement (CEGAR)



Localization Reduction, Kurshan, Bell Labs

Counterexample-guided Abstraction Refinement for Symbolic Model Checking, Clarke et al., CMU

Software Model Checking, SLAM Project, Microsoft, Ball & Rajamani

The Running Example

Program	Property	Expected Answer
(see below)	EF (pc = 5)	False

```
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   y = y - 1;
4: if (x == 2)
5:     error();
6:
```

An Example Abstraction

Program

```
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   y = y - 1;
4: if (x == 2)
5:     error();
6:
```

Abstraction

(with y<=2)

```
bool b is (y <= 2)
1: b = T;

2: while (b)
3:   b = ch(b,f);
4: if (*)
5:     error();
6:
```

Boolean (Predicate) Programs (BP)

Variables correspond to predicates

Usual control flow statements

while, if-then-else, goto

Expressions

usual Boolean expressions, plus

`*` — **unknown**

`ch(a,b)` — **if** a **then** **true** **else** **if** b **then** **false** **else** *

```
if a then
  true
else
  if b then
    false
  else *
```

Parallel Assignment

p_1 = ch(a_1,b_1), p_2 = ch(a_2,b_2), ...

b_1 = ch(b_1,$\neg b_1$), b_2 = ch($b_1 \vee b_2$, f), b_3=ch(f,f)

Detour: Pre- and Post-Conditions

A *Hoare triple* {P} C {Q} is a logical statement that holds when

For any state *s* that satisfies P, if executing statement C on *s* terminates with a state *s'*, then *s'* satisfies Q.

Statement

{P} C {Q}

Pre-condition (boolean formula)

Post-condition (boolean formula)

Detour: Weakest Liberal Pre-Condition

The *weakest liberal precondition of a statement C with respect to a post-condition Q* (written WLP(C,Q)) is a formula P such that

1. {P} C {Q}
2. for all other P' such that {P'} C {Q},
   P' $\Rightarrow$ P (P is weaker then P').

UNIVERSITY OF WATERLOO

Detour: Weakest Liberal Preconditions

```
{3>y} x = 3 {x>y}                    ✓

{x>0} x = 2+y {y>0}                  ✗

{*x>3 V x = &y} y=5 {*x>3}           ✓

{false} y=5 {y<0}                    ✓
```

Calculating Weakest Preconditions

Assignment (easy)

- WLP (x=e, Q) = Q[x/e]
  - Intuition: after an assignment, x gets the value of e, thus Q[x/e] is required to hold before x=e is executed

Examples:

WLP (x:=0, x=y) = (x=y)[x/0] = (0==y)

WLP (x:=0, x=y+1) = (x=y+1)[x/0] = (0 == y+1)

WLP (y:=y-1,y<=2) = (y<=2)[y/y-1] = (y-1 <= 2)

WLP(y:=y-1,x=2) = (x=2)[y/y-1] = (x == 2)

Boolean Program Abstraction

Update `p = ch(a, b)` is an approximation of a concrete statement `S` iff {a} S {p} and {b} S {¬p} are valid

- i.e., `y = y - 1` is approximated by
  - `(x == 2) = ch(x ==2, x!=2)`, and
  - `(y <= 2) = ch(y<=2,false)`

Parallel assignment approximates a concrete statement `S` iff all of its updates approximate `S`

- i.e., `y = y - 1` is approximated by

  `(x == 2) = ch(x ==2, x!=2),`

  `(y <= 2) = ch(y<=2,false)`

A Boolean program approximates a concrete program iff all of its statements approximate corresponding concrete statements

Computing An Abstract Update

```
// S a statement under abstraction
// P a list of predicates used for abstraction
// t a target predicate for the update
absUpdate (Statement S, List<Predicates> P, Predicate q) {
  resT, resF = false, false;
  // foreach monomial (full conjunction of literals) in P
  foreach m : monomials(P) {
    if (SMT_IS_VALID("m ⇒ WLP(S,q)") resT = resT V m;
    if (SMT_IS_VALID("m ⇒ WLP(S,¬q)") resF = resF V m;
  }
  return "q = ch(resT, resF)"
}
```

UNIVERSITY OF WATERLOO

absUpdate (y=y-1, p={y<=2}, q=(y<=2))

```
y = y - 1;
```

```
P is   {y <= 2}
q is   (y <= 2)
```

```
WLP(y=y-1,y<=2)      is (y-1) <= 2

WLP(y=y-1,¬(y<=2)) is (y-1) > 2
```

absUpdate

```
(y<=2) = ch (y<=2,f)
```

SMT Queries:

(y<=2) ⇒ (y-1) <= 2	✓
¬(y<=2) ⇒ (y-1) <= 2	✗
(y<=2) ⇒ (y-1) > 2	✗
¬(y<=2) ⇒ (y-1) > 2	✗

Example: Abstracting Skip Statement

```
skip
```

```
p is {x=0}
q is {y=0}
z is {x=y}
```

abs

```
p = ch (p        , !p        ),
q = ch (q        , !q        ),
z = ch (z        , !z        )
```

UNIVERSITY OF WATERLOO

Example: Abstracting Skip Statement

```
skip
```

abs

```
p is {x=0}
q is {y=0}
z is {x=y}
```

```
p = ch (p||(q&&z), !p||(q&&!z)),
q = ch (q||(p&&z), !q||(p&&!z)),
z = ch (z||(p&&q), !z||(p!=q))
```

UNIVERSITY OF WATERLOO

The result of abstraction

Program

```
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   y = y - 1;
4: if (x == 2)
5:      error();
6:
```

Abstraction

(with y<=2)

```
bool b is (y <= 2)
1: b = T;

2: while (b)
3:   b = ch(b,f);
4: if (*)
5:      error();
6:
```

But what is the semantics of Boolean programs?

UNIVERSITY OF WATERLOO

BP Semantics: Overview

Over-Approximation

- treat “unknown” as non-deterministic
- good for establishing correctness of universal properties

Under-Approximation

- treat “unknown” as abort
- good for establishing failure of universal properties

Exact Approximation

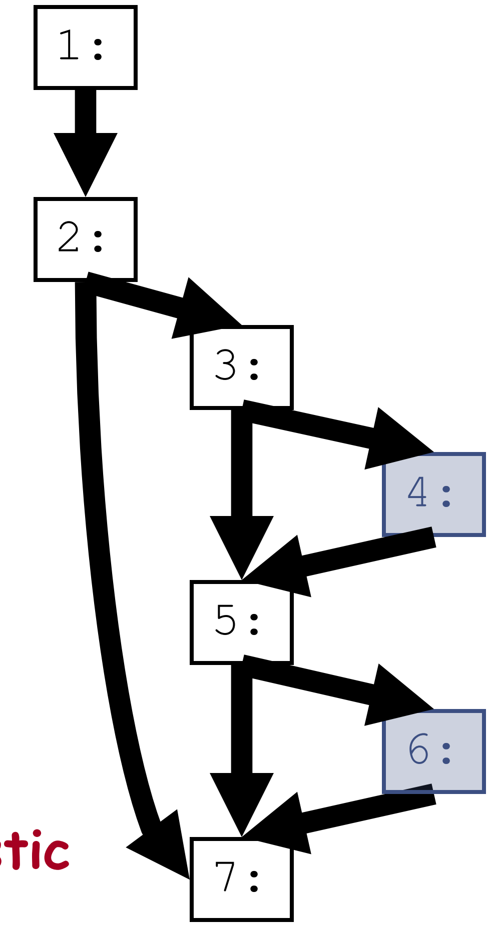
- Treat “unknown” as a special unknown value
- good for verification and refutation
- good for universal, existential, and mixed properties

BP Semantics: Over-Approximation

Abstraction

```
1: ;
2: if (nondet) {
3:   if (*)
4:       error();
5:   if (nondet)
6:      error();
7: }
```

Over-Approximation



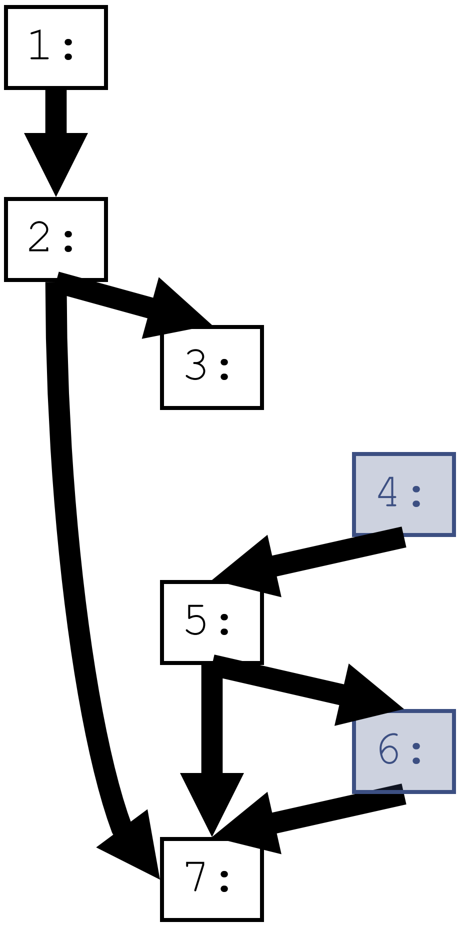
Unknown is treated as non-deterministic

BP Semantics: Under-Approximation

Abstraction

```
1: ;
2: if (nondet) {
3:   if (*)
4:       ERROR;
5:   if (nondet)
6:      ERROR;
7: }
```

Under-Approximation



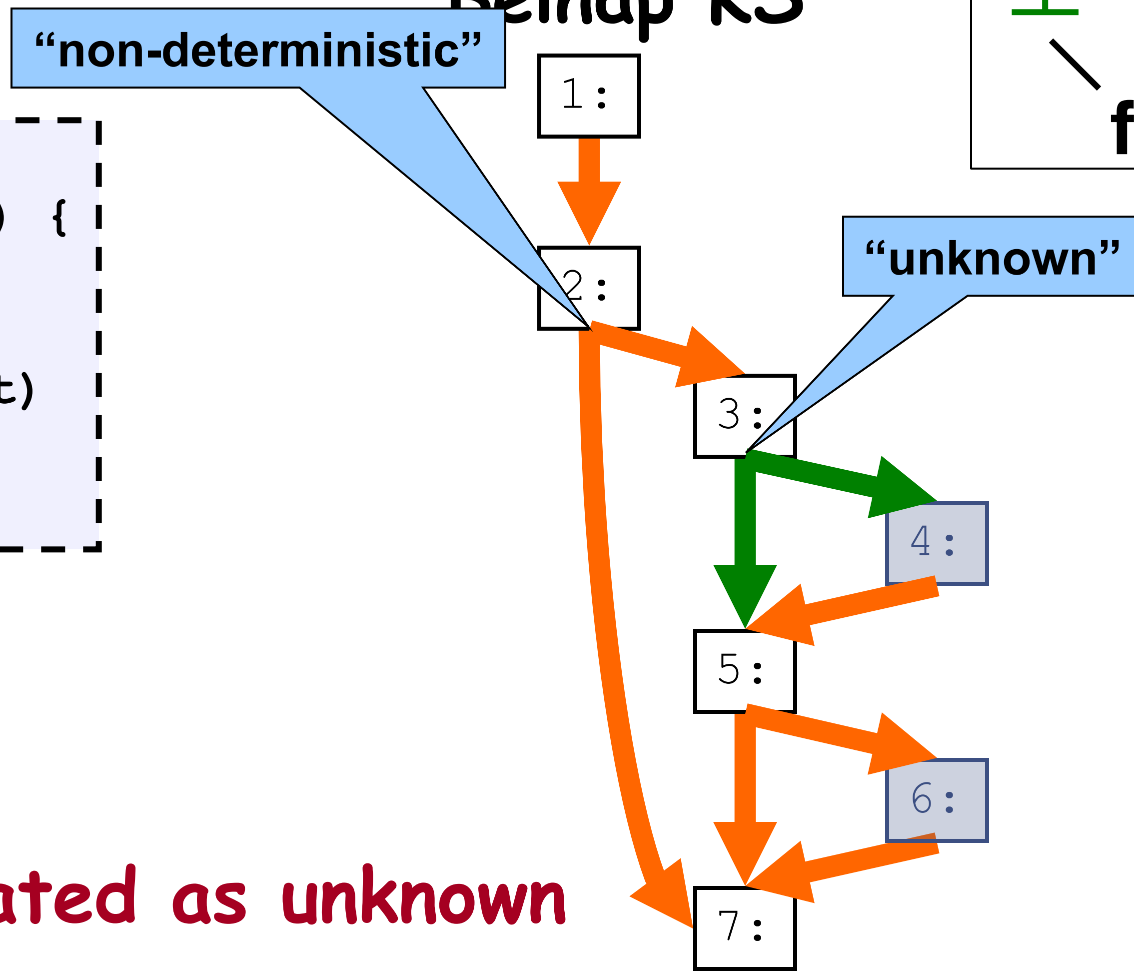
Unknown is treated as abort

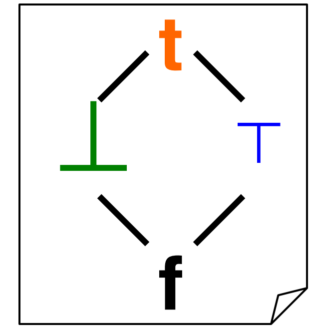
BP Semantics: Exact Approximation

Abstraction

```
1: ;
2: if (nondet) {
3:   if (*)
4:       ERROR;
5:   if (nondet)
6:      ERROR;
7: }
```

Exact Belnap KS





Unknown is treated as unknown

Summary: The Three Semantics

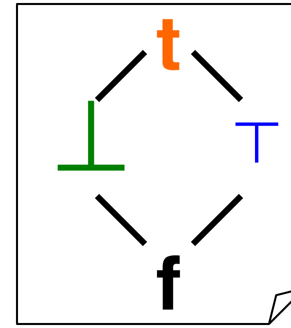
Concrete

```
y = y - 1;
```

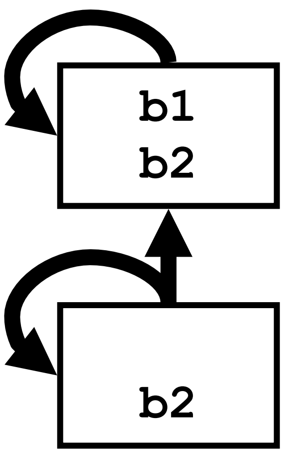
Abstract

b1 is (`y <= 2`)
b2 is (`x == 2`)

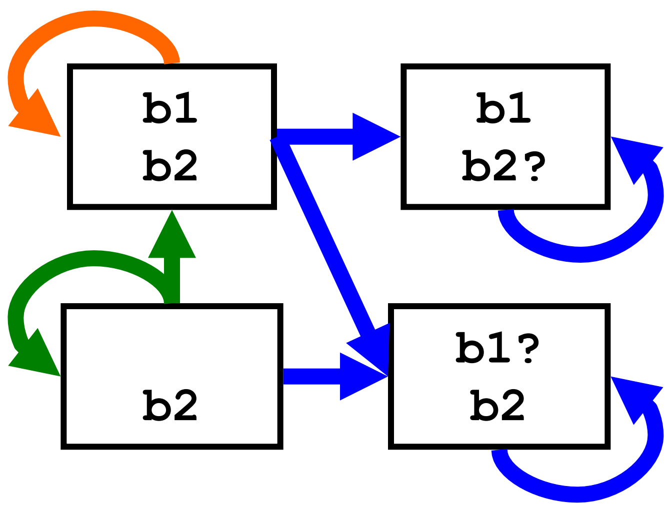
```
b1 = ch(b1,f);
b2 = ch(b2,¬b2)
```



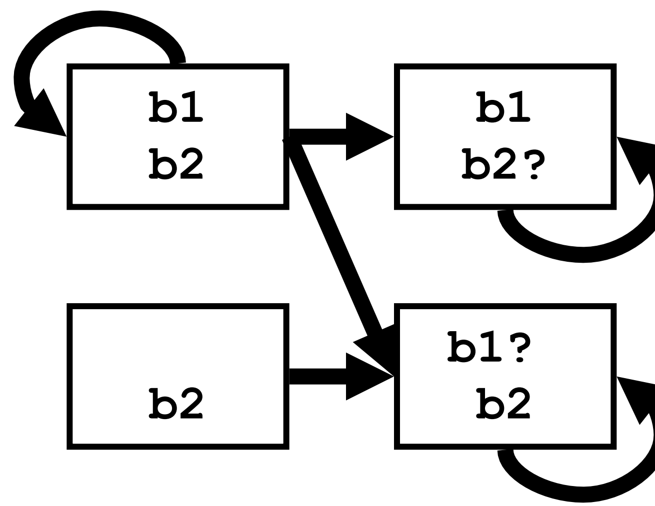
Over-Approx



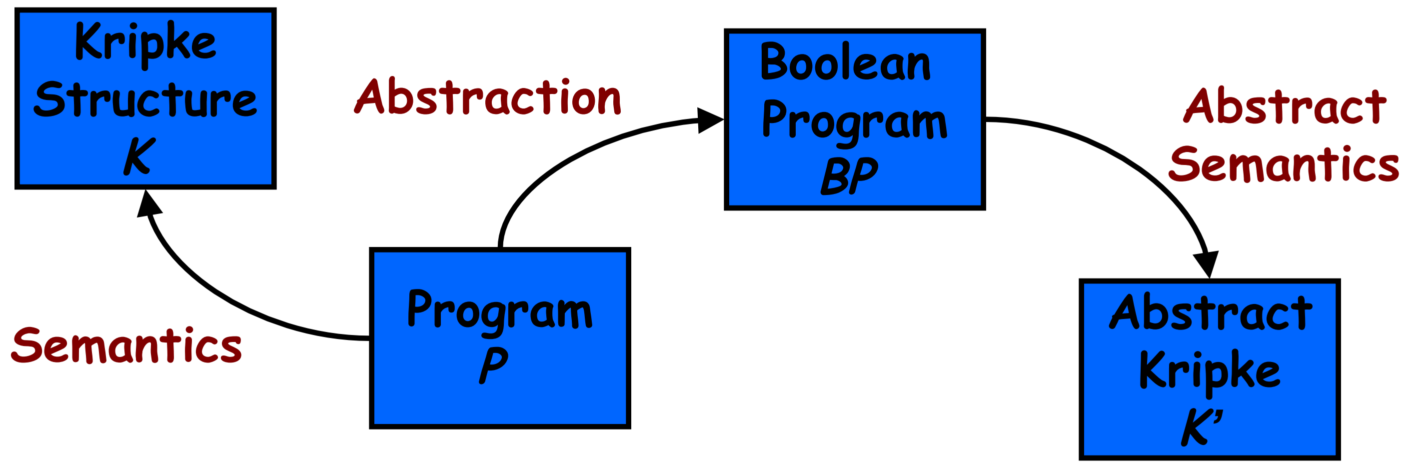
Belnap (Exact)



Under-Approx



Summary: Program Abstraction



Abstract a program P by a Boolean program BP

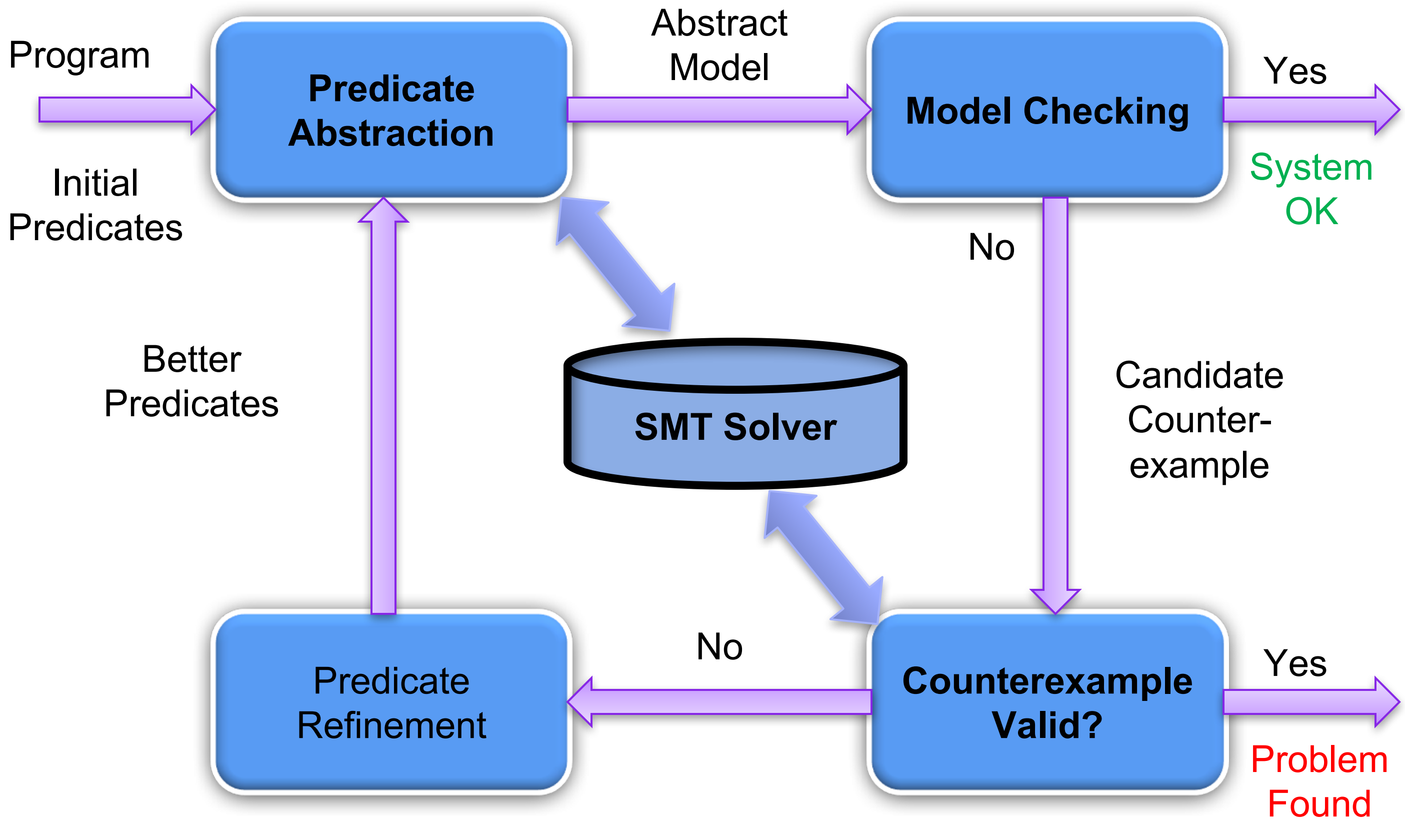
Pick an abstract semantics for this BP:

- Over-approximating
- Under-approximating
- Belnap (Exact)

Yield relationship between K and K':

- Over-approximation
- Under-approximation
- Belnap abstraction

CounterExample Guided Abstraction Refinement (CEGAR)



Localization Reduction, Kurshan, Bell Labs

Counterexample-guided Abstraction Refinement for Symbolic Model Checking, Clarke et al., CMU

Software Model Checking, SLAM Project, Microsoft, Ball & Rajamani

Example: Is ERROR Unreachable?

Program

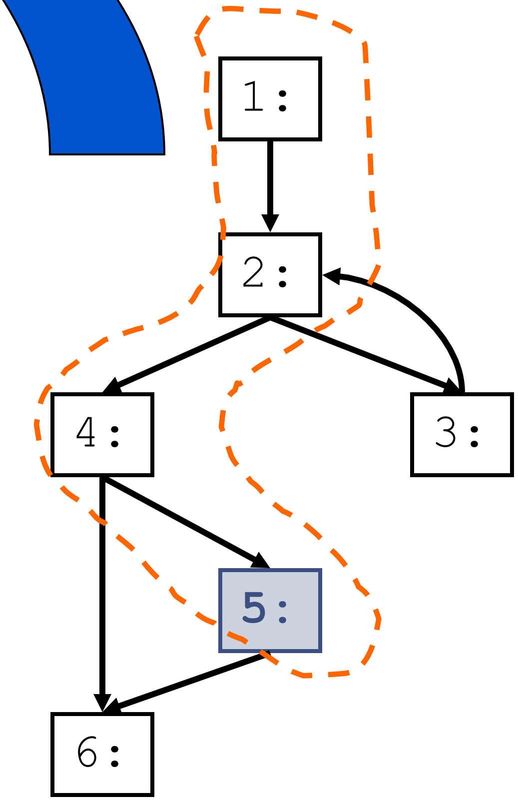
Need This!

```
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:       y = y - 1;
4: if (x == 2)
5:       error();
6:
```

Abstraction

```
1: ;

2: while (*)
3:       ;
4: if (*)
5:       error();
6:
```

Over-Approximation



CEGAR steps

Abstract ⟹ Translate ⟹ Check ⟹ Validate ⟹ Repeat

Example: Is ERROR Unreachable?

Program

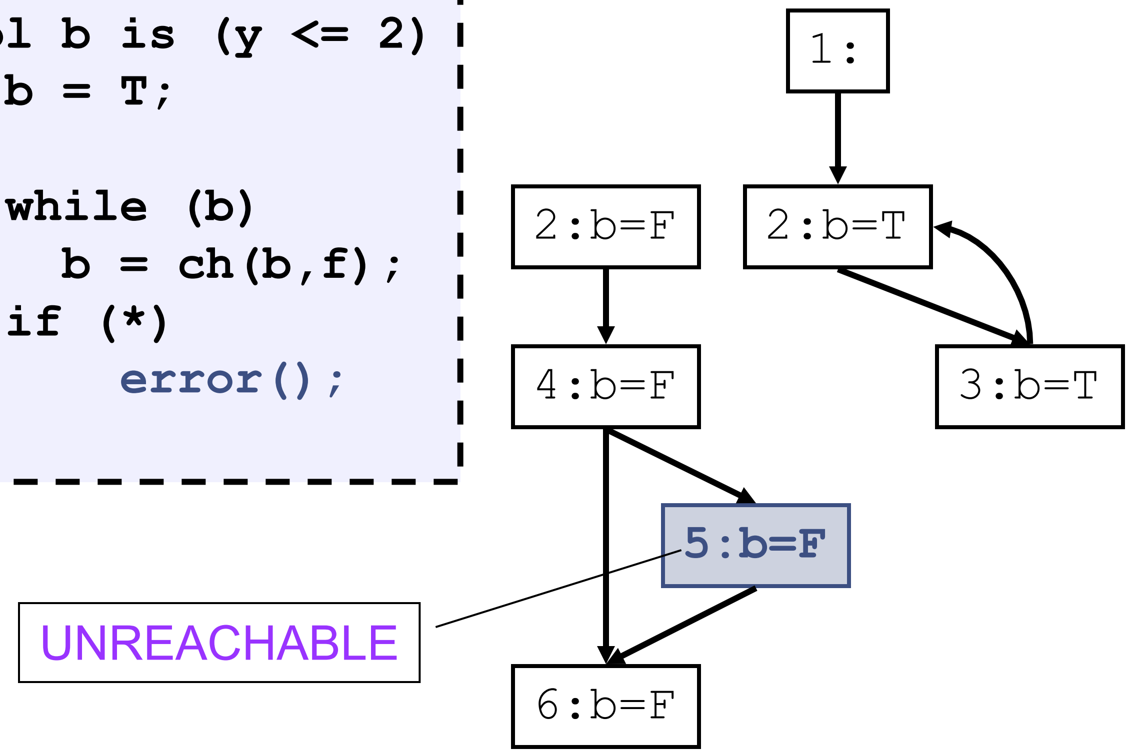
```
1:  int x = 2;
    int y = 2;
2:  while (y <= 2)
3:     y = y - 1;
4:  if (x == 2)
5:       error();
6:
```

Abstraction

(with y<=2)

```
bool b is (y <= 2)
1:  b = T;

2:  while (b)
3:     b = ch(b,f);
4:  if (*)
5:       error();
6:
```

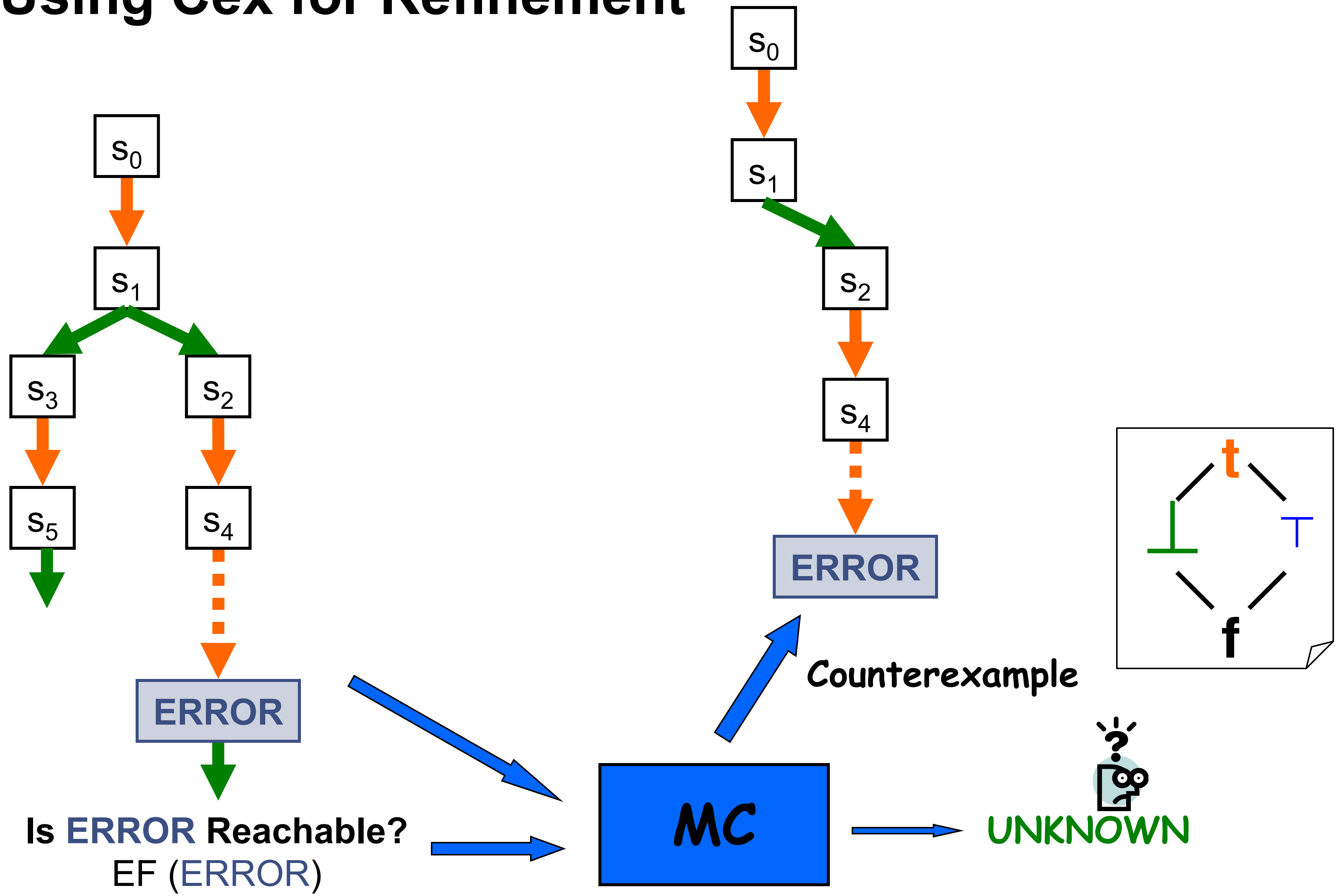
Over-Approximation



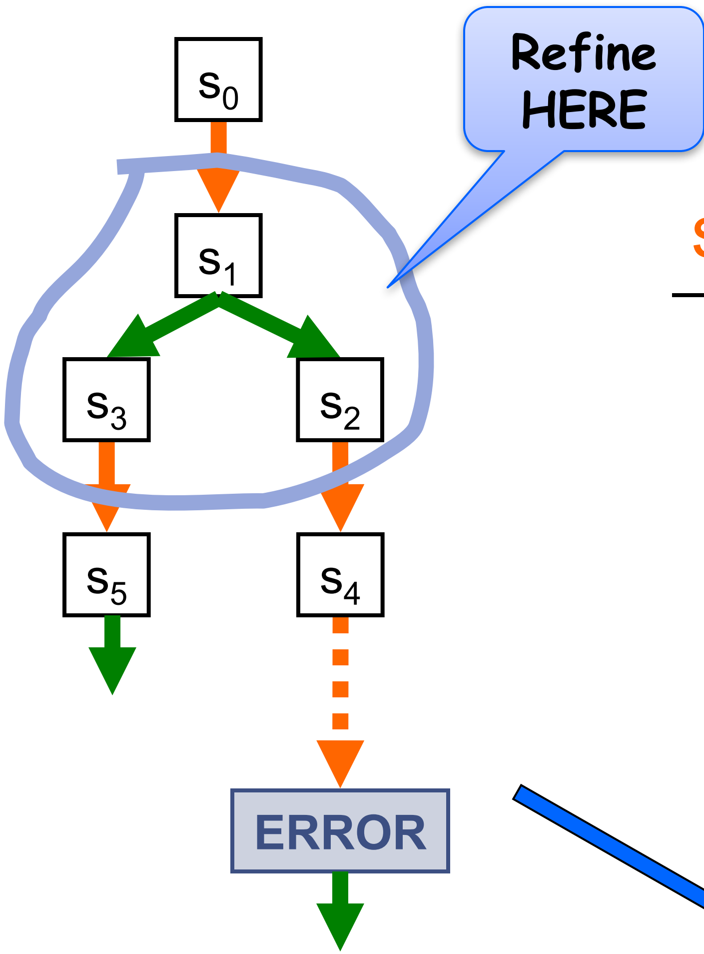
CEGAR steps

Abstract ⟹ Translate ⟹ Check ⟹ NO ERROR

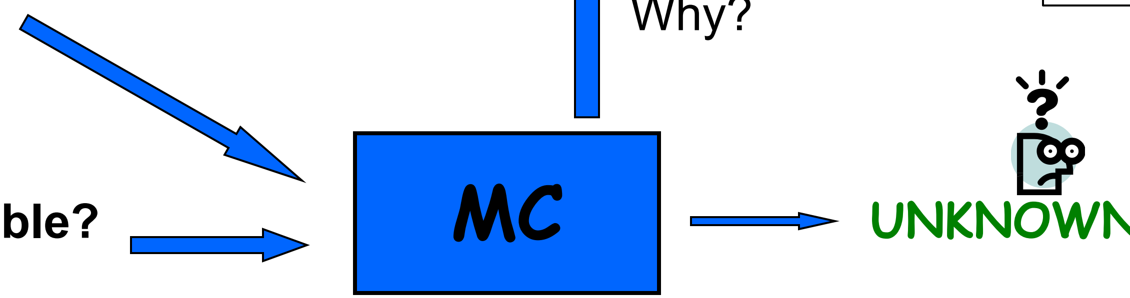
Using Cex for Refinement



Using Proofs for Refinement



Is ERROR Reachable?
EF (ERROR)

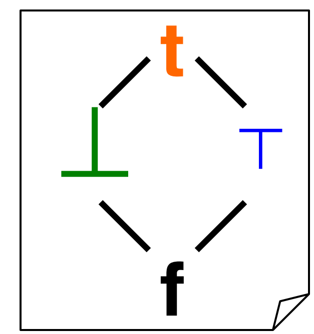


Why?

cause

can stop here

$$\dfrac{\dfrac{s_0 \to s_1 \qquad \dfrac{s_1 \to s_2 \qquad EF_2(ERROR)(s_2) = t}{EF_3(ERROR)(s_1) = \bot}}{EF_4\,(ERROR)(s_0) = \bot}}{\dfrac{\exists n\; EF_n\,(ERROR)(s_0) = \bot}{EF\,(ERROR)\,(s_0) = \bot}}$$



Finding Refinement Predicates

Recall

- each abstract state is a conjunction of predicates
  - i.e., $y<=2 \wedge x==2$ $y>2 \wedge x!=2$ etc.
- each abstract transition corresponds to a program statement

Result from a partial proof

Unknown transition
$s_1 \rightarrow s_2$

MC needs to know validity of

$$\{s_1\}\ C\ \{s_2\}$$

C is the statement corresponding to the transition

UNIVERSITY OF WATERLOO

Refinement via Weakest Liberal Precondition

If $s_1 \rightarrow s_2$ corresponds to a conditional statement

- refine by adding the condition as a new predicate

If $s_1 \rightarrow s_2$ corresponds to a statement C

- Find a predicate p in s_2 with uncertain value
  - i.e., $\{s_1\}$ C $\{p\}$ is not valid
- refine by adding WLP(C,p)

Finding New Predicate Example

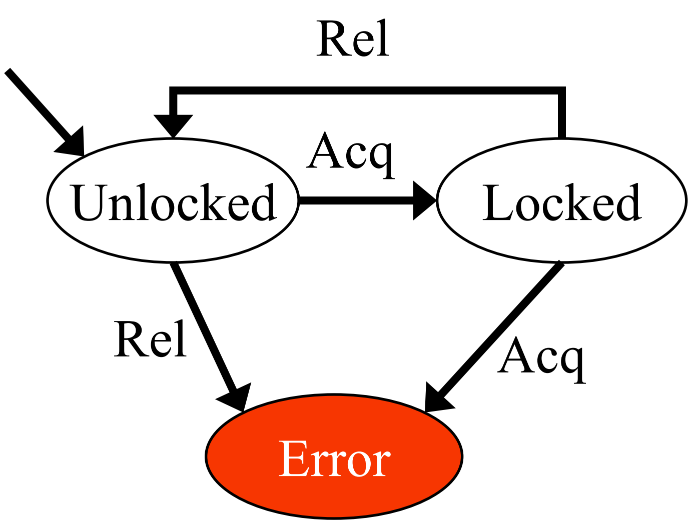
$s_1 \rightarrow s_2$ is unknown

`{y>2∧x==2} y = y-1 {y>2∧x==2}`

`{ {y>2∧x==2} y = y-1 {x==2} }`

new predicate `WLP(y = y-1, y>2) = y>3`

Example of Predicate Abstraction




```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;

   if(request){
     request = request->Next;
     KeReleaseSpinLock();
     nPackets++;
   }
} while (nPackets != nPacketsOld);

KeReleaseSpinLock();
```

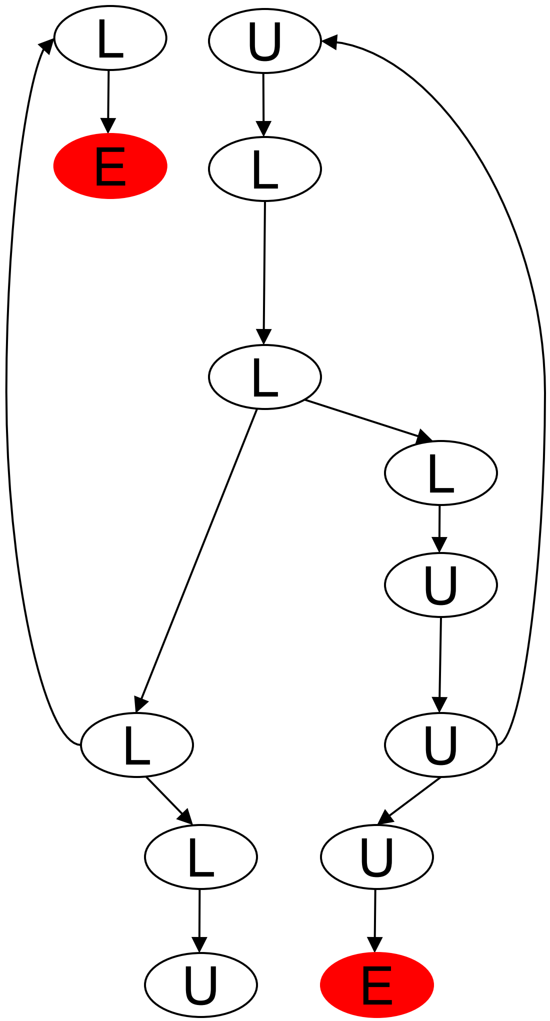
Abstraction (via Boolean program)

```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;

   if(request){
    request = request->Next;
    KeReleaseSpinLock();
    nPackets++;
   }
} while(nPackets!=nPacketsOld);

KeReleaseSpinLock();
```

```
s:=U;
do {
   assert(s=U); s:=L;

   if(*){

    assert(s=L); s:=U;

   }
} while (*);

assert(s=L); s:=U;
```

UNIVERSITY OF WATERLOO

Abstraction (via Boolean program)




```
s:=U;
do {
   assert(s=U); s:=L;

   if(*){

    assert(s=L); s:=U;

   }
} while (*);

assert(s=L); s:=U;
```

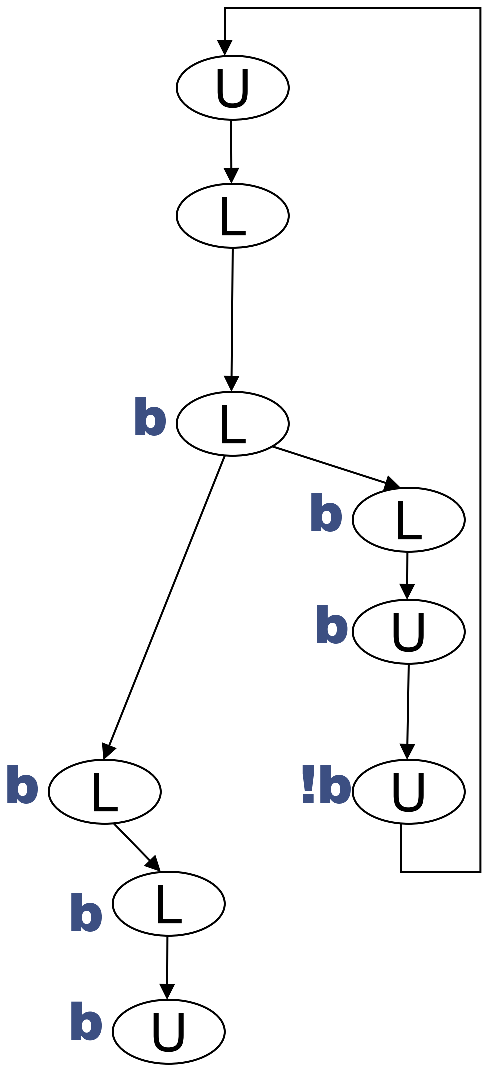
Refined Boolean Abstraction

b : (nPacketsOld == nPackets)

```
do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets;

  if(request){
   request = request->Next;
   KeReleaseSpinLock();
   nPackets++;
  }
} while(nPackets!=nPacketsOld);

KeReleaseSpinLock();
```

```
s:=U;
do {
  assert(s=U); s:=L;

  b := true;

  if(*){

   assert(s=L); s:=U;
   b := b ? false : *;
  }
} while ( !b   );

assert(s=L); s:=U;
```

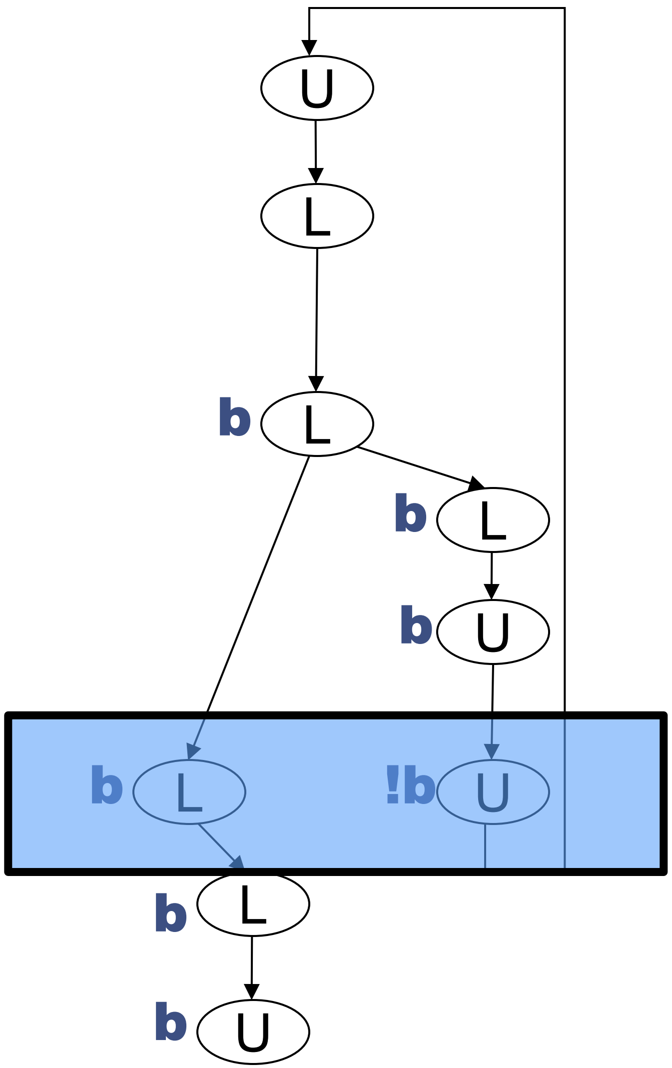
Refined Boolean Abstraction



b : (nPacketsOld == nPackets)

```
s:=U;
do {
   assert(s=U); s:=L;

   b := true;

   if(*){

    assert(s=L); s:=U;
    b := b ? false : *;
   }
} while ( !b  );

assert(s=L); s:=U;
```

Inductive Invariant



Inductive invariant is the set of states reachable at the head of the loop

$$(b \wedge L) \vee (\neg b \wedge U)$$

$$\equiv \quad b \iff L$$

$$\equiv \quad \mathtt{nPacketsOld} = \mathtt{nPackets} \iff \mathtt{Locked}$$

Lock is held iff `nPacketsOld == nPackets`

Summary: Predicate Abstraction and CEGAR

Predicate abstraction with CEGAR is an effective technique for analyzing behavioral properties of software systems

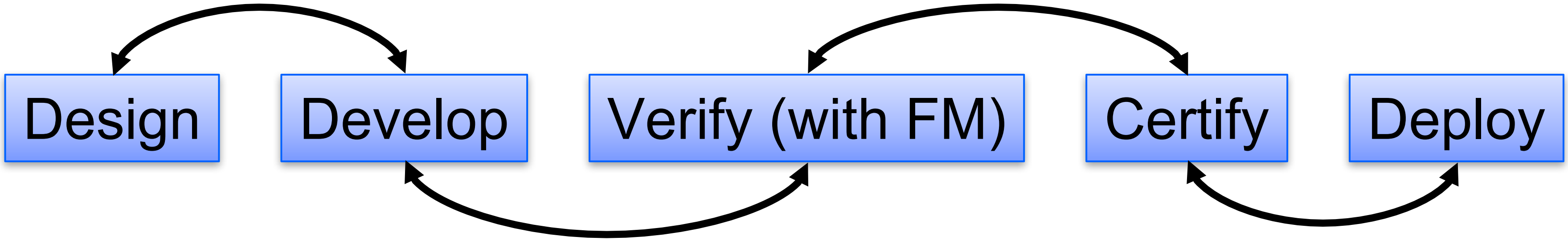
Combines static analysis and traditional model-checking

Abstraction is essential for scalability

- Boolean programs are used as an intermediate step
- Different abstract semantics lead to different abs.
  - over-, under-, Belnap

Automatic abstraction refinement finds the “right” abstraction incrementally

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?

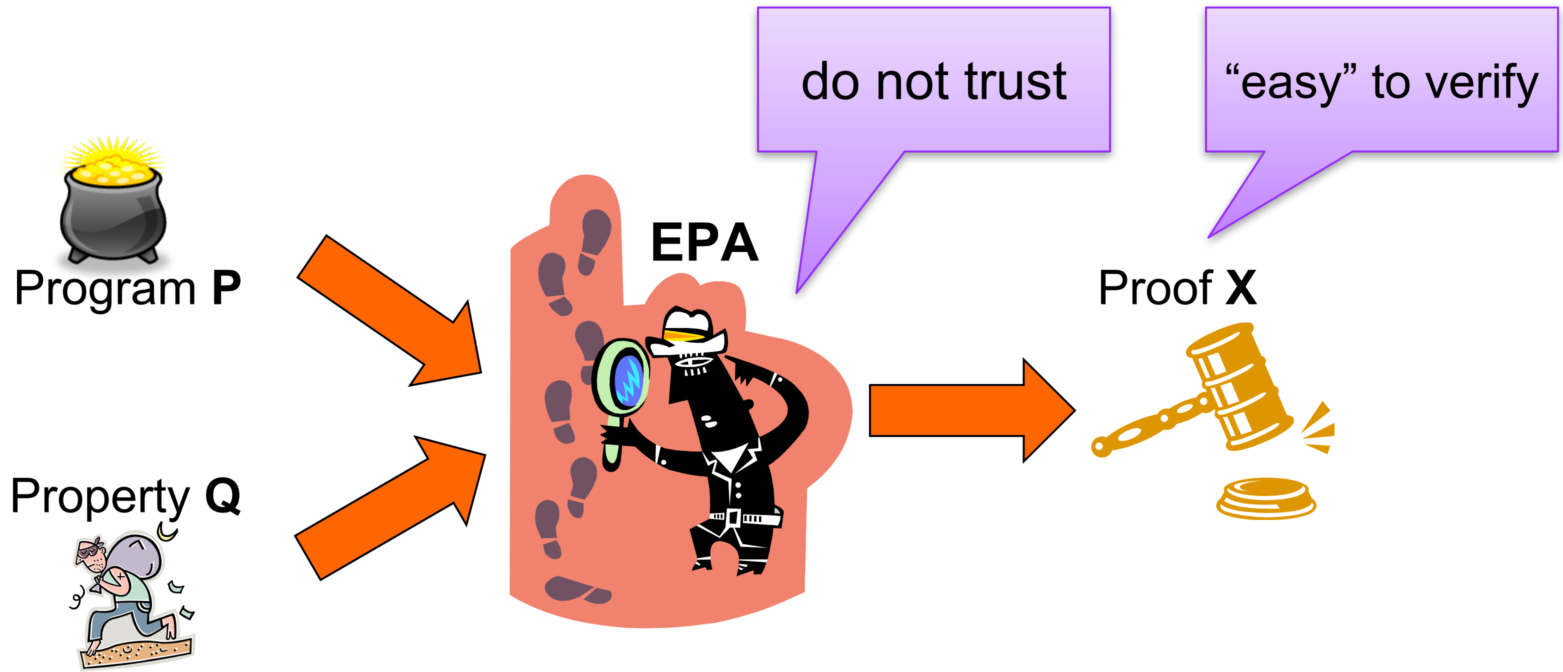
Verify the verifier

- (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



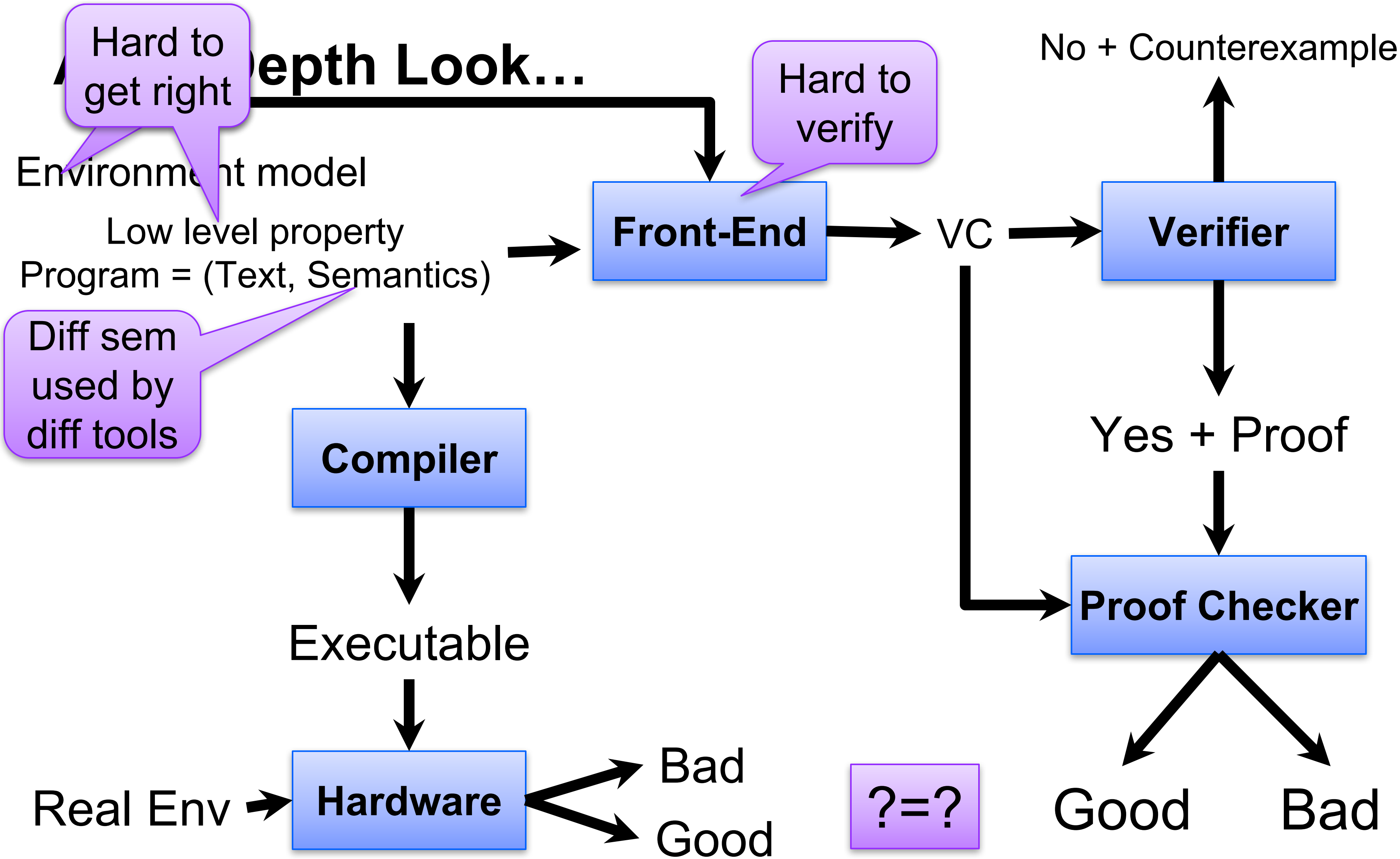
X witnesses that **P** satisfies **Q**. **X** can be objectively and independently verified.
Therefore, **EPA** is outside the Trusted Computing Base (**TCB**).

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 epth 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

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