Model Checking

Lecture 13: CPEN 400P

Karthik Pattabiraman, UBC

(Slides based on Arie Gurfinkel's slides at the University of Waterloo in ECE 750T)

Outline

What is model checking ?

Kripke Structures

CTL (Computation Tree Logic)

Model Checking of Programs

Counter Example Guided Abstraction Refinement (CEGAR)

Model Checking

Definition: Method for checking whether a finite-state model of a system meets a given specification (also known as correctness)

- Can be applied to both hardware and software systems

Largely automatic and fast

Better suited for debugging

- … rather than assurance

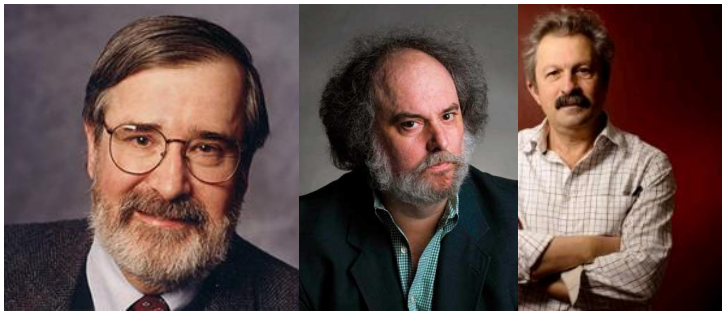
Model-checking Vs Testing

- Usually, model checking finds more problems by
  exploring all behaviours of a downscaled system
  than by
  testing some behaviours of the full system

(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

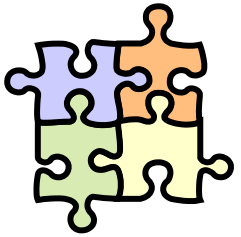
Model Checking since 1981

1981 Clarke / Emerson: CTL Model Checking
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, …

Outline

What is model checking ?

Kripke Structures

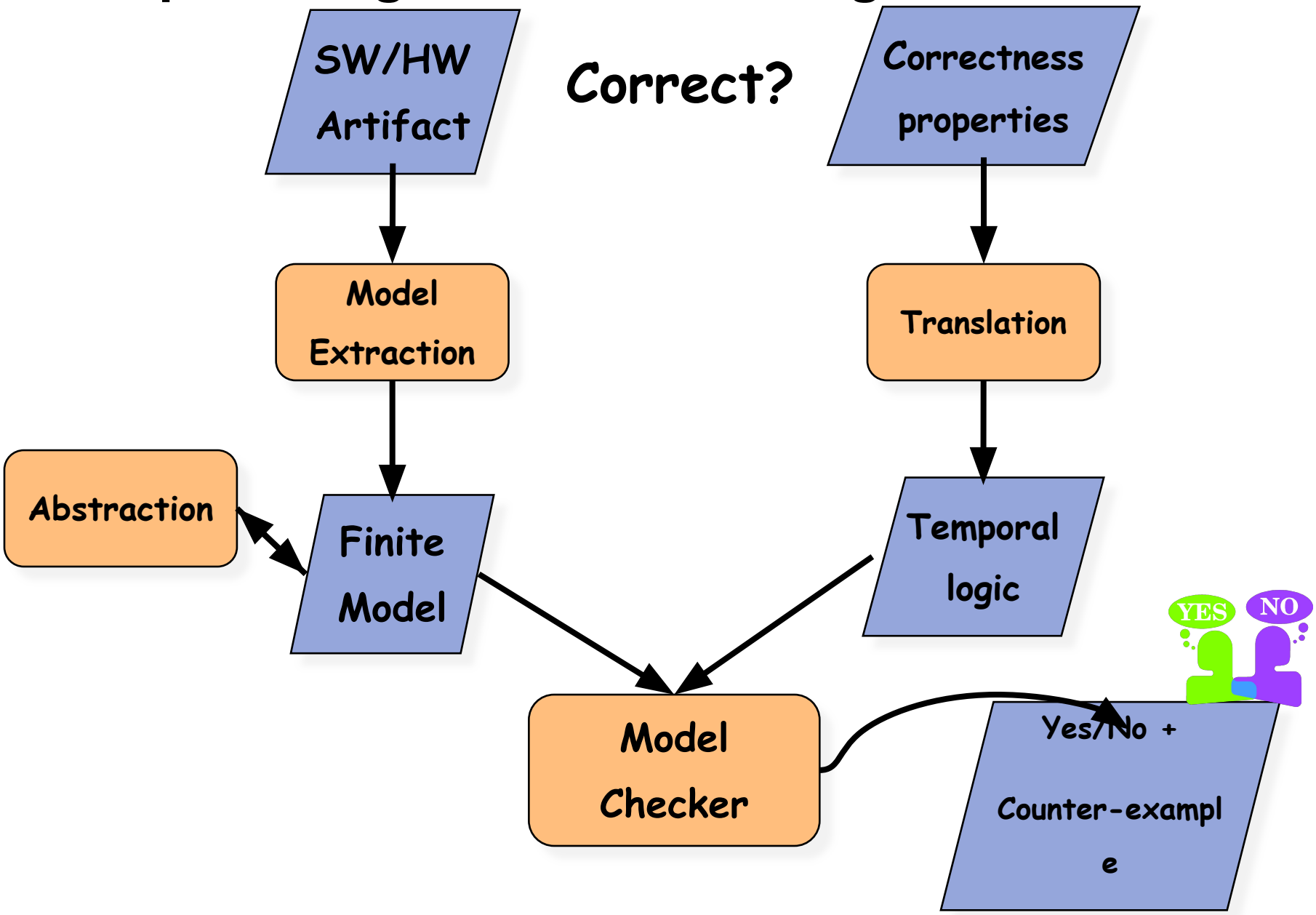
CTL (Computation Tree Logic)

LTL (Linear Temporal Logic)

Model Checking of Programs

Counter Example Guided Abstraction Refinement (CEGAR)

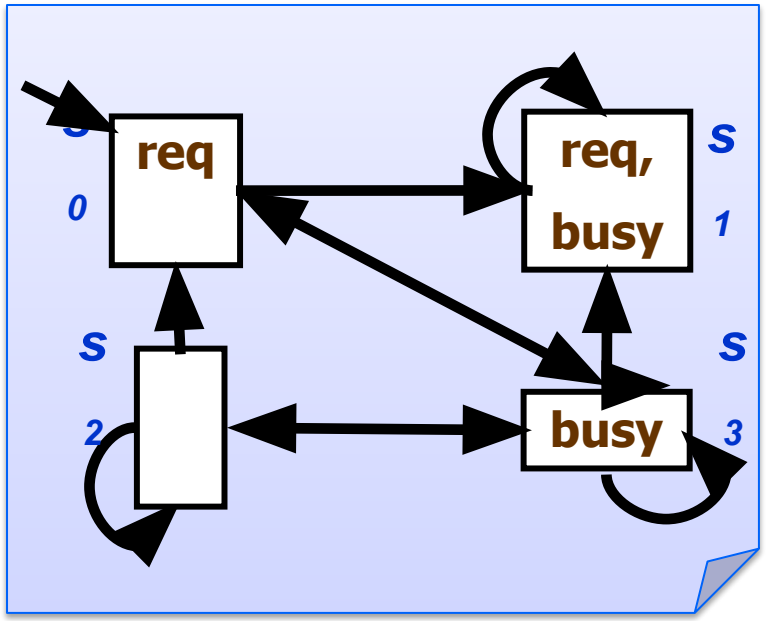
Temporal Logic Model Checking



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
- $I: S \rightarrow 2^V$ 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

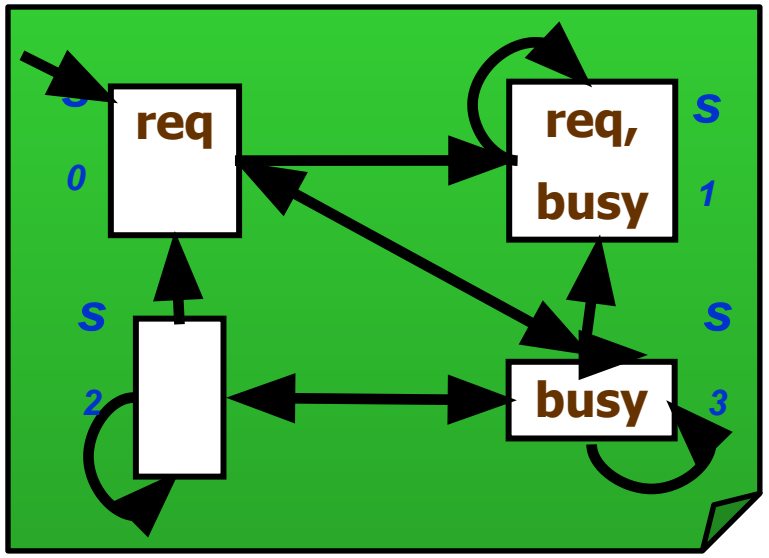
- “Printer is busy”
- “There are currently no requested jobs for the printer”
- “Conveyer belt is stopped”

Do not involve time!

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

Outline

What is model checking ?

Kripke Structures

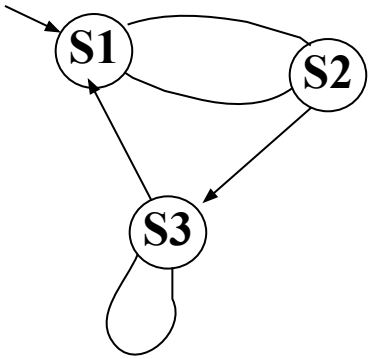
CTL (Computation Tree Logic)

Model Checking of Programs

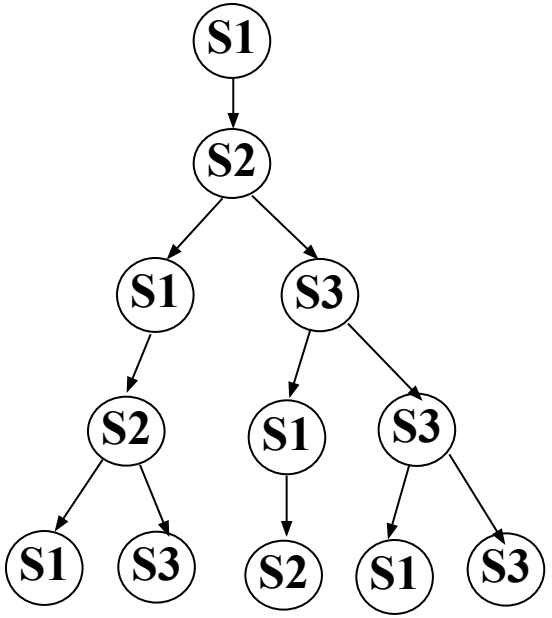
Counter Example Guided Abstraction Refinement (CEGAR)

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 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: $\neg \phi$, $\phi \wedge \psi$, $\phi \vee \psi$

EX ϕ : ϕ holds in some next state

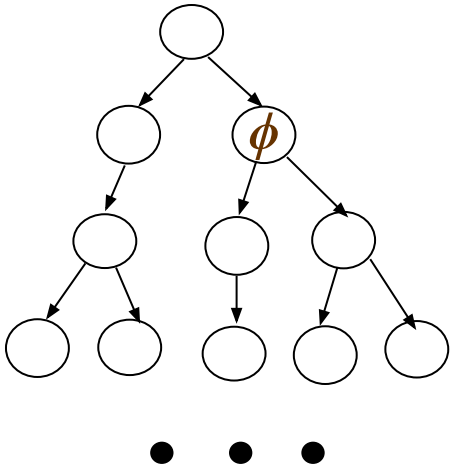
EF ϕ : along some path, ϕ holds in a future state

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

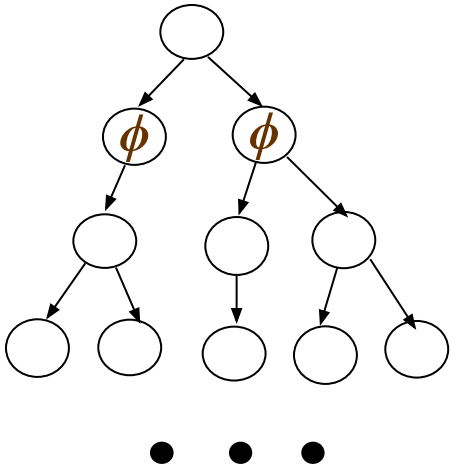
EG ϕ : along some path, ϕ holds in every state

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

Examples: EX and AX

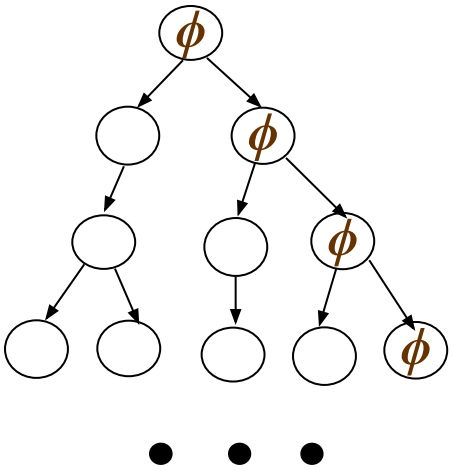


EX ϕ (exists next)

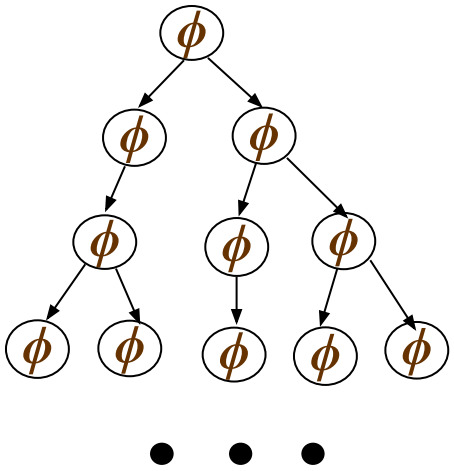


AX ϕ (all next)

Examples: EG and AG

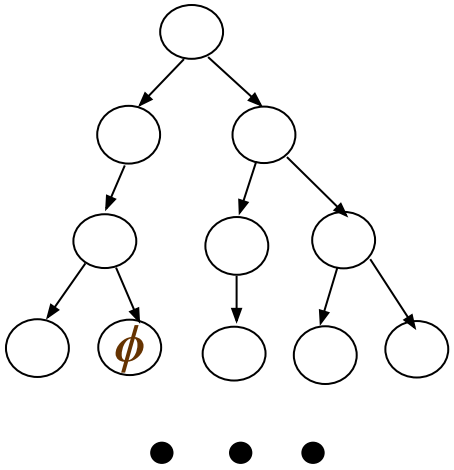


EG ϕ (exists global)

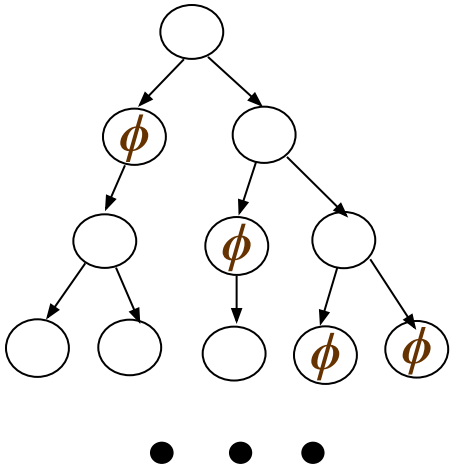


AG ϕ (all global)

Examples: EF and AF

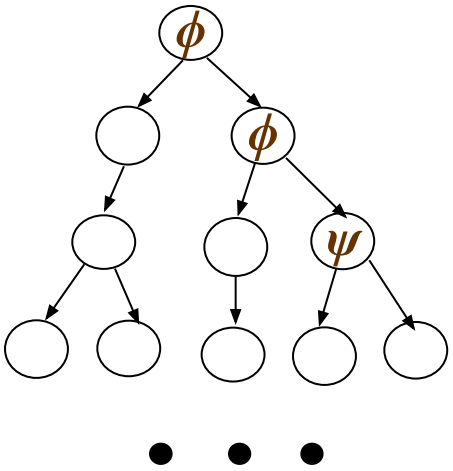


EF ϕ (exists future)

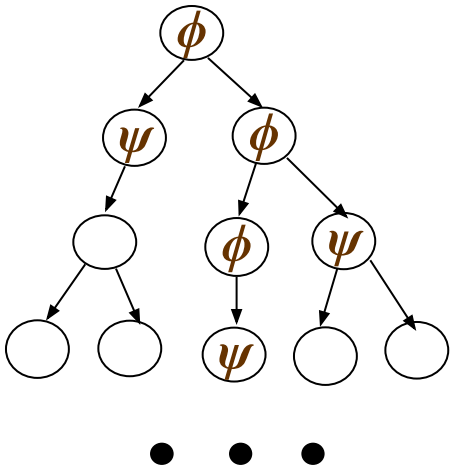


AF ϕ (all future)

Examples: EU and AU



E[ϕ U ψ] (exists until)



A[ϕ U ψ] (all until)

Examples of CTL

Let P mean “It’s cloudy outside”

Let Q mean “It’s going to rain”

Then,

AG(Q) - It’s going to rain everyday starting from today, regardless of what happens

EG(P) - It’s possible that it’s going to be cloudy forever in the future (at some point)

EF(Q) - It’s possible that it’s going to rain in the future at least for one day

AF(Q) - It’s possible that at every point in the future, we will have at least one day where it rains

AF EG (Q) - It’s always possible that in the future, it’s going to rain for the rest of time (i.e., forever)

AG(P U Q)

From now on until it rains, it’s going to be cloudy every single day in the future. Once it rains, then it may not rain anymore after that. Also, it’s guaranteed to rain eventually, even if only for a single day, in the future.

EF(EX(P) U AG(Q))

It’s possible that there will eventually come a time when it will rain forever (AG.Q), and that before that time, there will always be some day in the future, for which it’s going to be cloudy the next day (EX.P)

Semantics of CTL

$K,s \models \phi$ – 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,

$AX\ \phi = \forall \pi \cdot \pi^1 \models \phi$ $\quad$ $EX\ \phi = \exists \pi \cdot \pi^1 \models \phi$

$AG\ \phi = \forall \pi \cdot \forall i \cdot \pi^i \models \phi$ $\quad$ $EG\ \phi = \exists \pi \cdot \forall i \cdot \pi^i \models \phi$

$AF\ \phi = \forall \pi \cdot \exists i \cdot \pi^i \models \phi$ $\quad$ $EF\ \phi = \exists \pi \cdot \exists i \cdot \pi^i \models \phi$

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

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

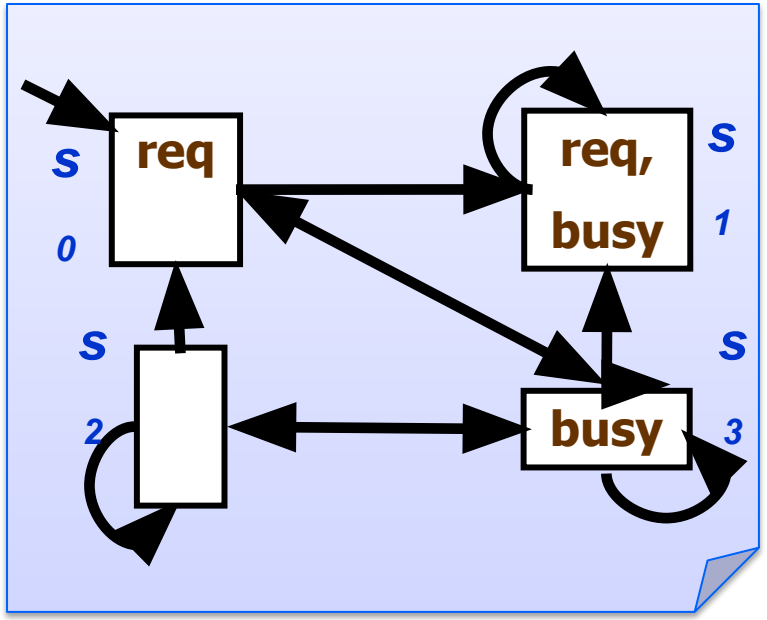
CTL Examples

Properties that hold:

- (AX busy)(s_0)
- (EG busy)(s_3)
- A (req U busy) (s_0)
- E (¬req U busy) (s_1)
- AG (req ⇒ AF busy) (s_0)

Properties that fail:

- (AX (req ∨ busy))(s_3)



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

Class Activity: Write the CTL formula for these

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

-

A process is enabled infinitely often on every computation path

-

A process will eventually be permanently deadlocked

-

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

-

- Note: hard to do correctly.

Class Activity: Solution

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

- EF (state=idle $\wedge$ floor=3 $\wedge$ doors=closed)

When a request occurs, it will eventually be acknowledged

- AG (request $\Rightarrow$ 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[$\neg q$ U ($q \wedge$ A[$\neg p$ U s])]
- Note: hard to do correctly.

Outline

What is model checking ?

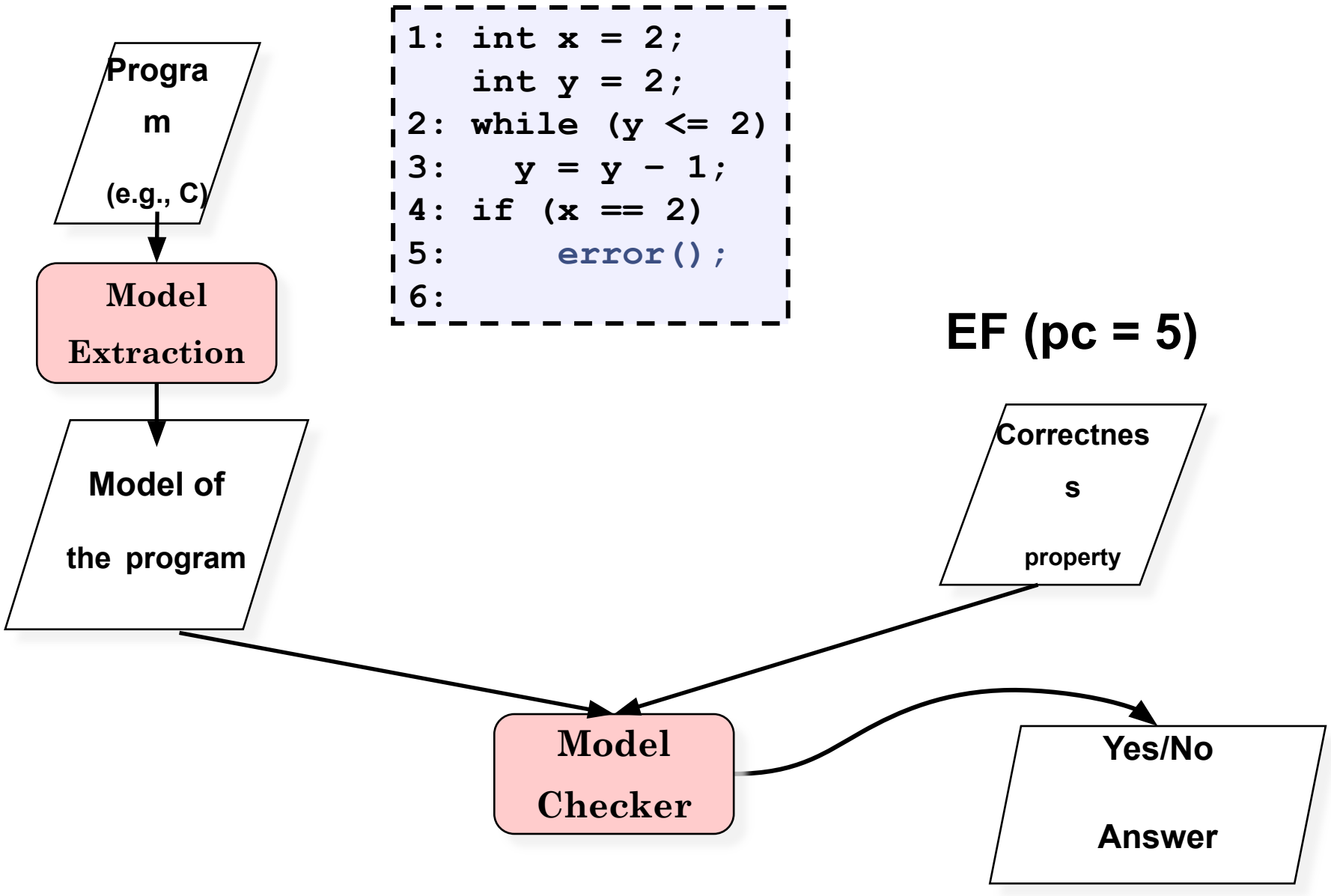
Kripke Structures

CTL (Computation Tree Logic)

Model Checking of Programs

Counter Example Guided Abstraction Refinement (CEGAR)

Software Model Checking



Assumptions: In Our Programming Language…

All variables are global

Functions are in-lined

`int` is integer

- i.e., no overflow

Special statements:

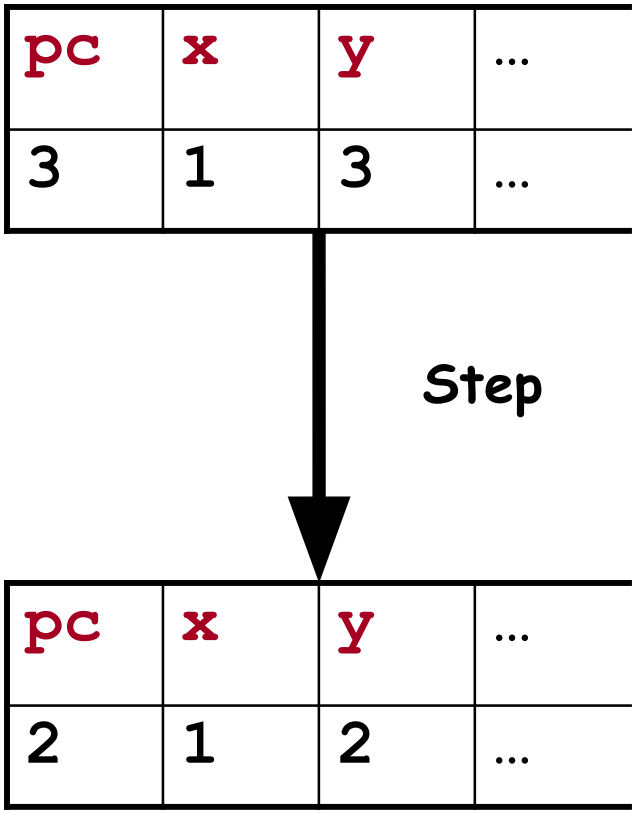
`skip`	**do nothing**
`assume(e)`	**if `e` then `skip` else abort**
`x,y=e1,e2`	**`x, y` are assigned `e1,e2` in parallel**
`x=nondet()`	**`x` gets an arbitrary value**
`goto L1,L2`	**non-deterministically go to `L1` or `L2`**

From Programs to Kripke Structures

Program

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

State



Property: EF (pc = 5)

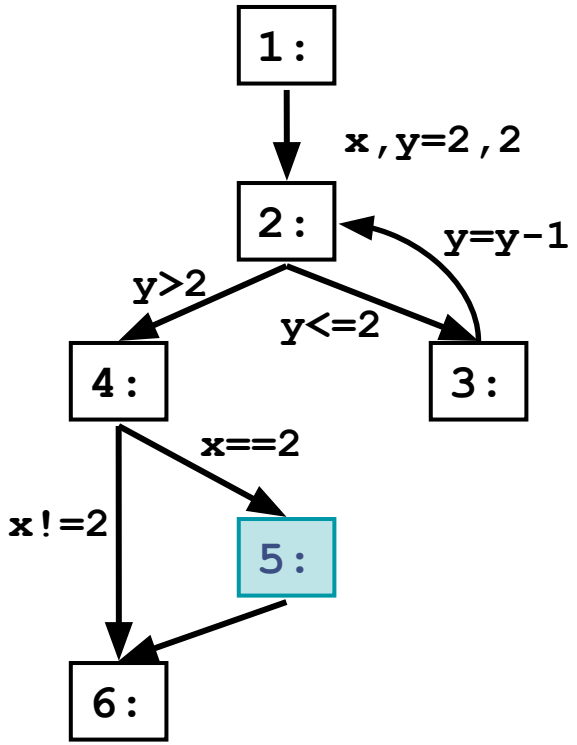
Programs as Control Flow Graphs

Program

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

Semantics S

Labeled CFG



Modeling in Software Model Checking

Software Model Checker works directly on the source code of a program

- Whole-program-analysis technique
- requires the user to provide the model of the environment with which the program interacts
  - e.g., physical sensors, operating system, external libraries, specifications

Programing languages already provide convenient primitives to describe behavior

- Extended to modeling and specification languages by adding new features
  - non-determinism: like random values, but without a probability distribution
  - assumptions: constraints on “random” values
  - assertions: an indication of a failure

From Programming to Modeling

Extend C programming language with 3 modeling features

Assertions

- assert(e) – aborts an execution when e is false, no-op otherwise

```
void assert (bool b) { if (!b)  error(); }
```

Non-determinism

- nondet_int() – returns a non-deterministic integer value

```
int nondet_int () { int x; return x; }
```

Assumptions

- assume(e) – “ignores” execution when e is false, no-op otherwise

```
void assume (bool e) { while (!e) ;  }
```

Non-determinism vs. Randomness

A *deterministic* function always returns the same result on the same input

- e.g., F(5) = 10

A *non-deterministic* function may return different values on the same input

- e.g., G(5) in [0, 10] “G(5) returns a non-deterministic value between 0 and 10”

A *random* function may choose a different value with a probability distribution

- e.g., H(5) = (3 with prob. 0.3, 4 with prob. 0.2, and 5 with prob. 0.5)

Non-deterministic choice cannot be implemented !

- used to model the worst possible adversary/environment

Modeling with Non-determinism

```
int x, y;

void main (void)
{
  x = nondet_int ();

  assume (x > 10);
  assume (x <= 100);
  y = x + 1;

  assert (y > x);
  assert (y < 200);

}
```

What happens in this program ? Is there an execution of the program for which either assert is violated ?

Using nondet for modeling

Library spec:

- "foo is given via grab_foo(), and is busy until returned via return_foo()"

Model Checking stub:

```
int nondet_int ();
int is_foo_taken = 0;
int grab_foo () {
  if (!is_foo_taken)
    is_foo_taken = nondet_int ();
  return is_foo_taken; }
```

```
void return_foo ()
{ is_foo_taken = 0; }
```

Dangers of unrestricted assumptions

Assumptions can lead to vacuous correctness claims!!!

Is this program correct?

```
if (x > 0) {
  assume (x < 0);
  assert (0); }
```

Assume must either be checked with assert or used as an idiom:

```
x = nondet_int ();
y = nondet_int ();
assume (x < y);
```

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 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

Outline

What is model checking ?

Kripke Structures

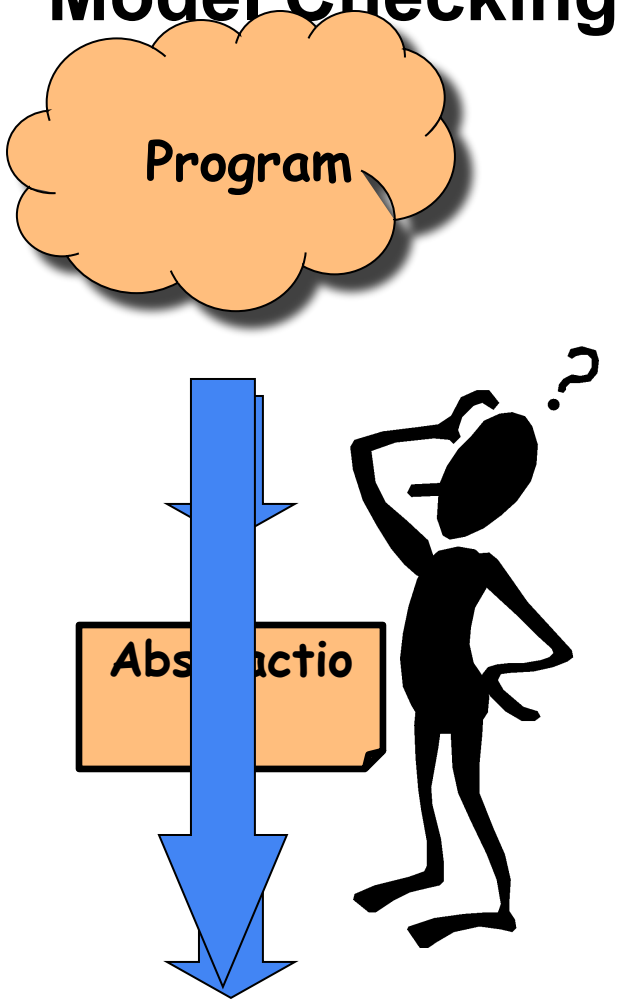
CTL (Computation Tree Logic)

LTL (Linear Temporal Logic)

Model Checking of Programs

Counter Example Guided Abstraction Refinement (CEGAR)

Model Checking Software by Abstraction

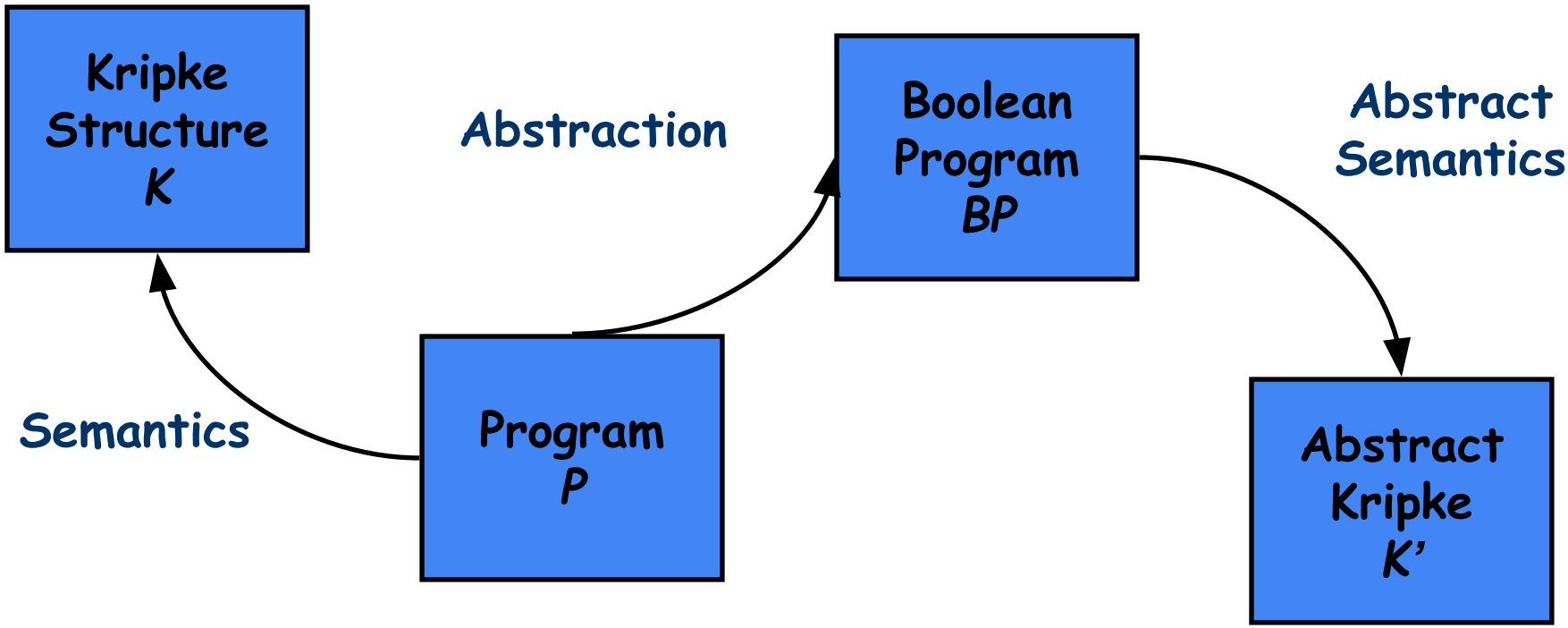


Programs are not finite state machines

- 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**

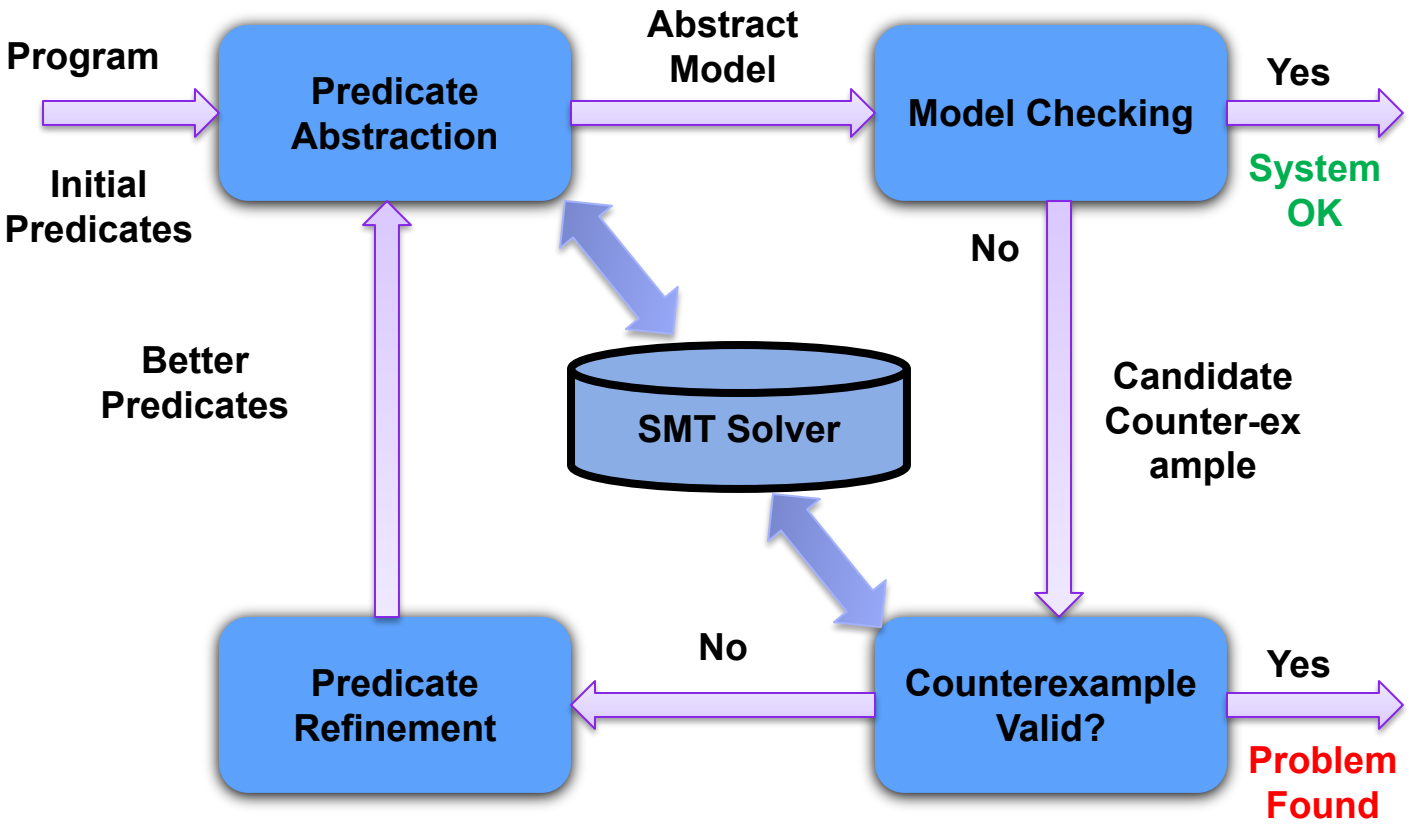
Software Model Checking and Abstraction



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

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:
```

BP Semantics: Overview

Over-Approximation

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

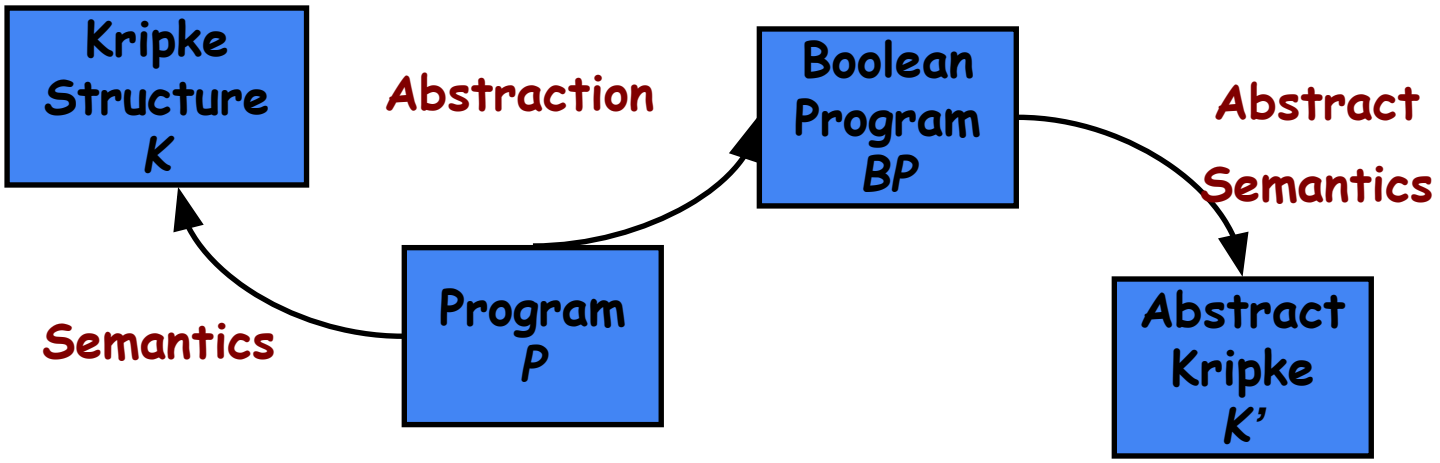
Under-Approximation

- 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

Summary: Program Abstraction



Abstract a program P by a Boolean program BP

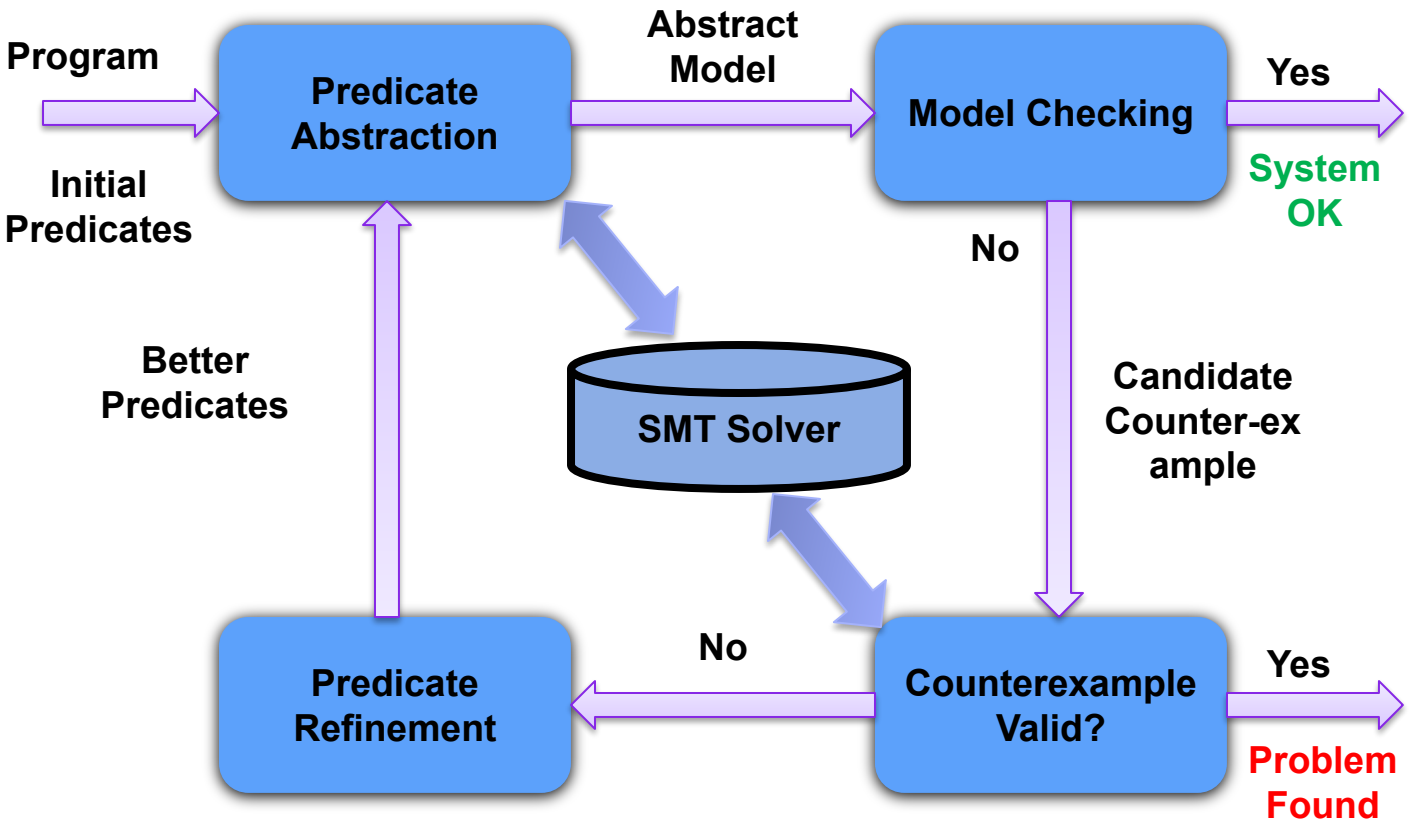
Pick an abstract semantics for this BP:

- 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

Outline

What is model checking ?

Kripke Structures

CTL (Computation Tree Logic)

Model Checking of Programs

Counter Example Guided Abstraction Refinement (CEGAR)