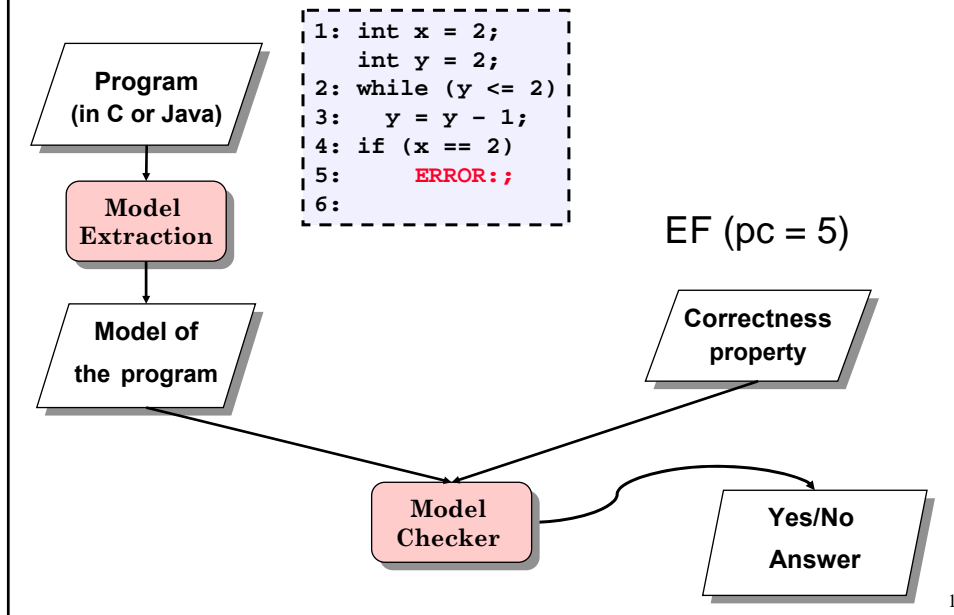
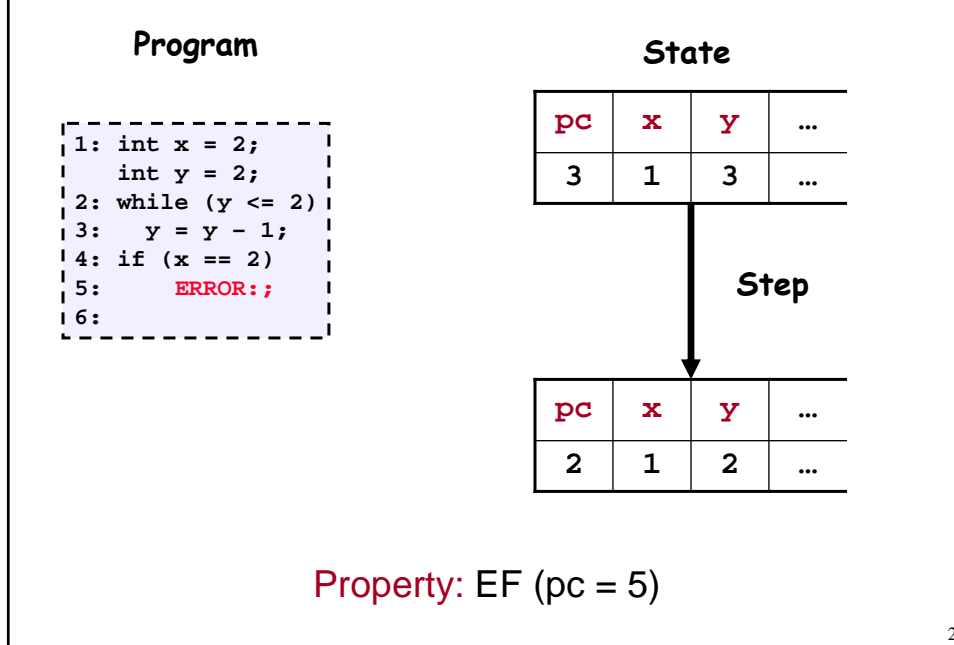


Software Model Checking



From Programs to Kripke Structures



In Our Programming Language...

- ⊃ All variables are global
- ⊃ Functions are in-lined
- ⊃ `int` is integer
 - ↳ i.e., no overflow
- ⊃ Special statements:

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

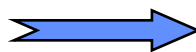
3

Programs as Control Flow Graphs

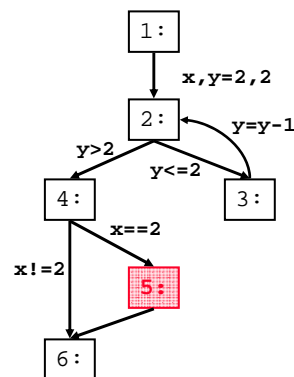
Program

```
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



4

Model Checking Software



⇒ Programs are not finite state

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

Model Checker

5

Model Checking Software



Abstraction



Model Checker

⇒ Programs are not finite state

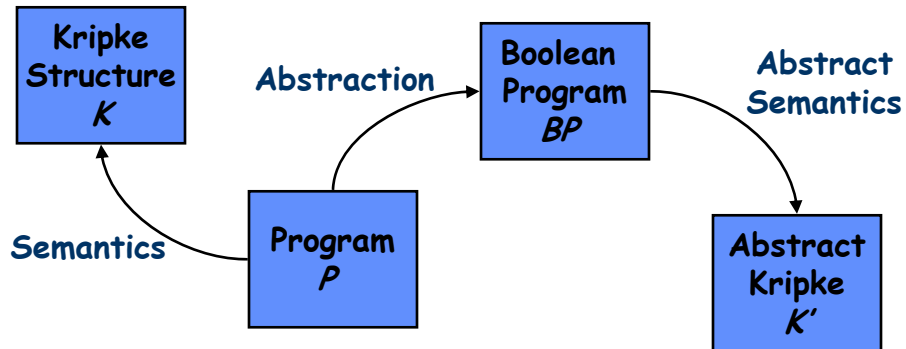
- ↪ integer variables
- ↪ recursion
- ↪ unbounded data structures
- ↪ dynamic memory allocation
- ↪ dynamic thread creation
- ↪ pointers
- ↪ ...

⇒ Build a finite abstraction

- ↪ ... small enough to analyze
- ↪ ... rich enough to give conclusive results

6

Software Model Checking and Abstraction

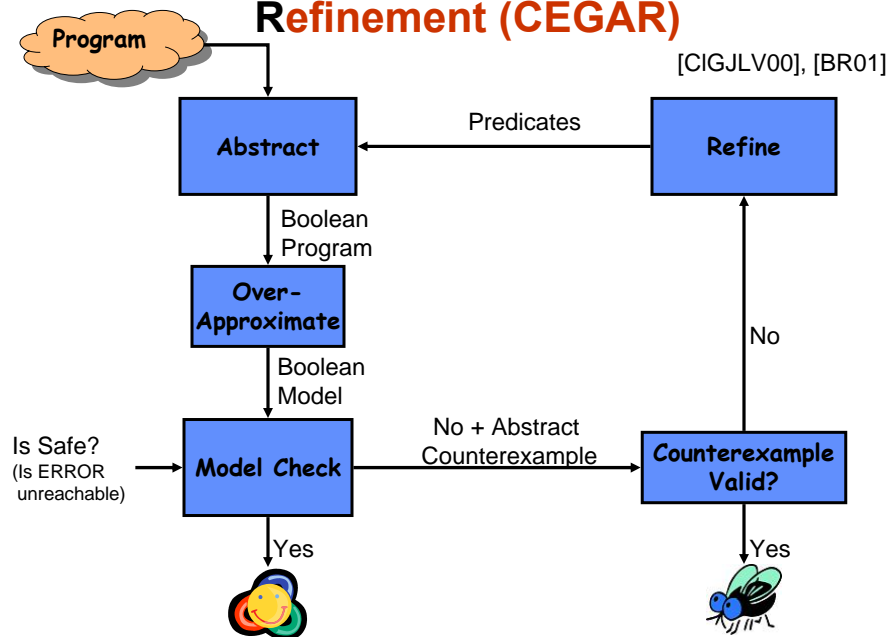


Soundness of Abstraction:

BP abstracts P implies that K' approximates K

7

CounterExample Guided Abstraction Refinement (CEGAR)



8

Outline

- ⊃ Programming Language
 - ↳ syntax and semantics
- ⊃ Predicate Abstraction for Programs
 - ↳ Boolean Programs as intermediate representation
 - ↳ Automatic computation of abstraction
- ⊃ Three abstract semantics of Boolean Programs
 - ↳ over-, under-, and Belnap abstractions
- ⊃ Discovering the “right” abstraction automatically
 - ↳ Counterexample-guided abstraction refinement
 - ↳ Finding a place to refine
 - counterexample- and proof-guided approaches
 - ↳ Discovering new predicates
- ⊃ Overview of state-of-the-art software MCs

9

The Running Example

Program	Property	Expected Answer
<pre>1: int x = 2; int y = 2; 2: while (y <= 2) 3: y = y - 1; 4: if (x == 2) 5: ERROR: ; 6:</pre>	EF (pc = 5)	False
	EG (pc ≠ 4) (loop does not terminate)	True

10

Boolean (Predicate) Programs (BP)

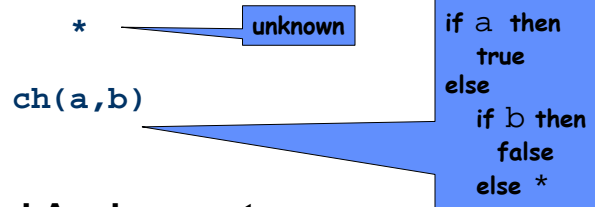
⊃ Variables correspond to predicates

⊃ Usual control flow statements

while, if-then-else, goto

⊃ Expressions

usual Boolean expressions



⊃ Parallel Assignment

$p_1 = \text{ch}(a_1, b_1), \quad p_2 = \text{ch}(a_2, b_2), \quad \dots$

$b_1 = \text{ch}(b_1, \neg b_1), \quad b_2 = \text{ch}(b_1 \vee b_2, f), \quad b_3 = \text{ch}(f, f)$

11

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 $y \leq 2$)

```

bool b is (y <= 2)
1: b = T;
2: while (b)
3:   b = ch(b, f);
4: if (*)
5:   ERROR;
6:

```

12

Boolean Program Abstraction

⇒ Update $p = \text{ch}(a, b)$ is an approximation of a concrete statement s iff $\{a\}S\{p\}$ and $\{b\}S\{\neg p\}$ are valid

⇐ i.e., $y = y - 1$ is approximated by

➤ $(x == 2) = \text{ch}(x == 2, x != 2)$, and

➤ $(y <= 2) = \text{ch}(y <= 2, \text{false})$

⇒ Parallel assignment approximates a concrete statement s iff all of its updates approximate s

⇐ i.e., $y = y - 1$ is approximated by

➤ $(x == 2) = \text{ch}(x == 2, x != 2)$

➤ $(y <= 2) = \text{ch}(y <= 2, \text{false})$

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

13

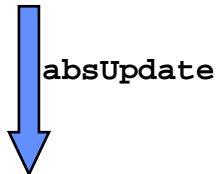
Computing An Abstract Update

```
// pre-condition: P.contains (t)
// 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 t) {
    resT, resF = false, false;
    foreach m : mono(P) {
        if (tpQ("m ⇒ WP(S,t)") resT = resT ∨ m;
        if (tpQ("m ⇒ WP(S,¬t)") resF = resF ∨ m;
    }
    return "t = ch(resT, resF)"
}
```

14

Example Computation

$y = y - 1;$



$(y \leq 2) = \text{ch } (y \leq 2, f)$

P is $\{y \leq 2\}$

t is $(y \leq 2)$

Theorem Prover Queries:

$(y \leq 2) \Rightarrow (y - 1) \leq 2$ ✓

$(y > 2) \Rightarrow (y - 1) \leq 2$ ✗

$(y \leq 2) \Rightarrow (y - 1) > 2$ ✗

$(y > 2) \Rightarrow (y - 1) > 2$ ✗

Aside:

$y > 2$ is $\neg(y \leq 2)$

15

Over-Approximating Semantics of BP

State: boolean valuations to *all* predicates

i.e., $b_1 \wedge b_2, \neg b_1 \wedge b_2, b_2 \wedge \neg b_1, \neg b_1 \wedge \neg b_2$

Semantics of an update

$O(p = \text{ch}(q, r)) (s, v) = (v = t \text{ and } s \neq r) \text{ or } (v = f \text{ and } s \neq q)$

Semantics of a parallel assignment

$O(\{U_i\}) (s, t) = \bigwedge_{i=\{1..n\}} O(U_i) (s, t(b_i))$

Program

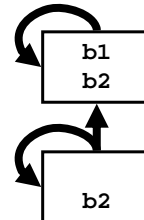
$y = y - 1;$

BP

$b1 = \text{ch}(b1, f),$
 $b2 = \text{ch}(b2, !b2)$

$b1$ is $y \leq 2$
 $b2$ is $x == 2$

O(BP)



16

Under-Approximating Semantics of BP

State: boolean valuations to *some* predicates

i.e., $b_1, \neg b_1, b_2 \wedge \neg b_1, \neg b_1 \wedge \neg b_2$, etc.

Semantics of an update

$U(p = \text{ch}(q, r))(s, v) = (v = t \text{ and } a \models q) \text{ or } (v = f \text{ and } a \models r) \text{ or } (v = \perp)$

Semantics of a parallel assignment

conjunction as before

Program

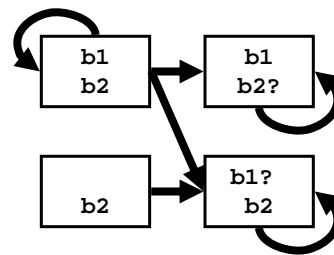
```
y = y - 1
```

BP

```
b1=ch(b1, f),
b2=ch(b2, !b2)
```

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

U(BP)



17

Belnap (Exact) Semantics of BP

State: boolean valuations to *some* predicates

i.e., $b_1, \neg b_1, b_2 \wedge \neg b_1, \neg b_1 \wedge \neg b_2$, etc.

Semantics of an update

$E(p = \text{ch}(q, r))(s, v) = (v = t \wedge s \models_3 \text{ch}(q, r)) \text{ or } (v = f \wedge s \models_3 \neg \text{ch}(q, r)) \text{ or } (v = \perp \wedge \top)$

Semantics of a parallel assignment

conjunction as before

Program

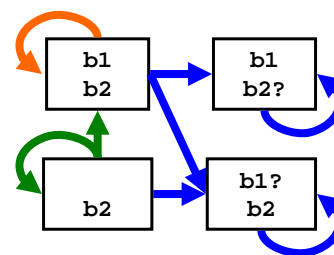
```
y = y - 1
```

BP

```
b1=ch(b1, f),
b2=ch(b2, !b2)
```

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

E(BP)



18

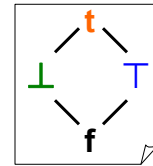
Summary: The Three Semantics

Concrete

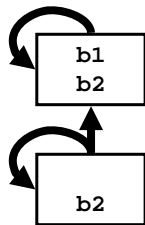
```
y = y - 1;
```

Abstract

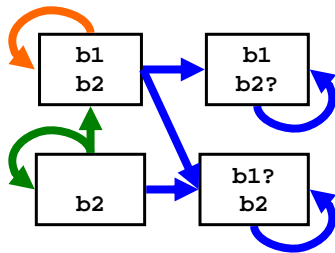
```
b1 is (y <= 2)
b2 is (x == 2)
b1 = ch(b1, f);
b2 = ch(b2, -b2)
```



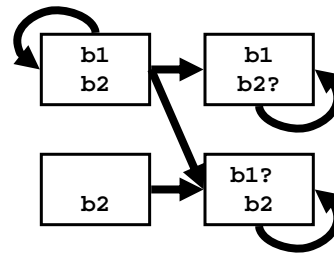
Over-Approx



Belnap (Exact)

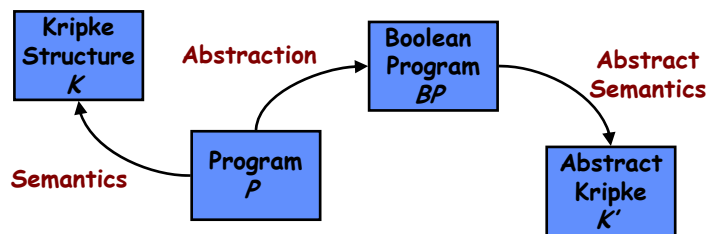


Under-Approx



19

Summary: Program Abstraction



1. Abstract a program P by a boolean program BP
2. Pick an abstract semantics for this BP :
 1. Over-approximating
 2. Under-approximating
 3. Belnap (Exact)
3. Yield relationship between K and K' :
 1. Over-approximation
 2. Under-approximation
 3. Belnap abstraction

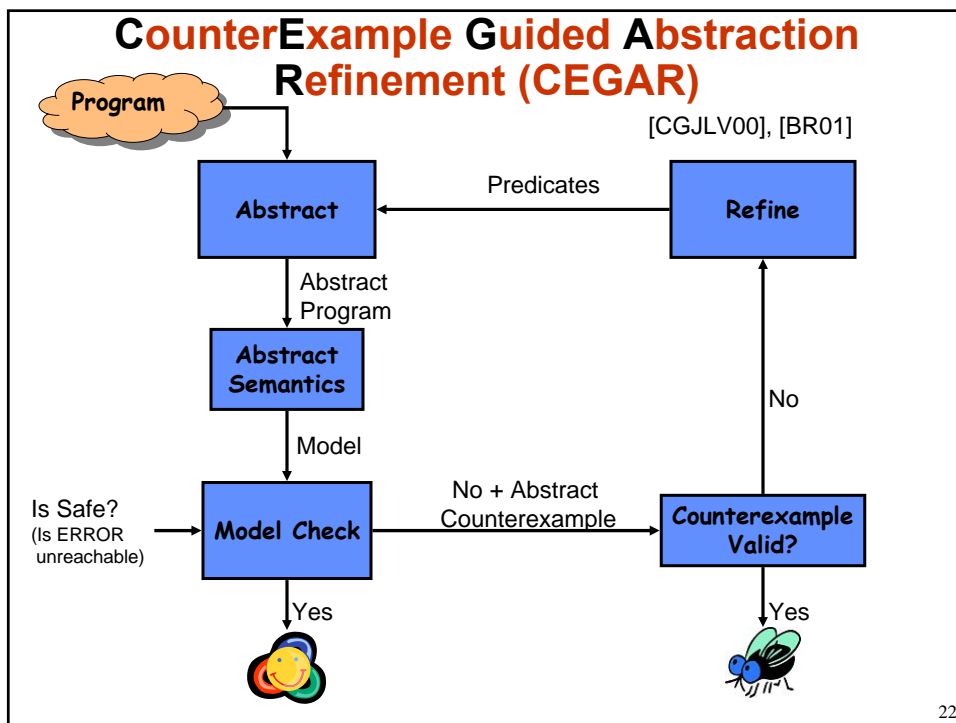
20

Outline

- ⊃ Programming Language
 - ↳ syntax and semantics
- ⊃ Predicate Abstraction for Programs
 - ↳ Boolean Programs as intermediate representation
 - ↳ Automatic computation of abstraction
- ⊃ Three abstract semantics of Boolean Programs
 - ↳ over-, under-, and Belnap abstractions
- ⊃ Discovering the “right” abstraction automatically
 - ↳ Counterexample-guided abstraction refinement
 - ↳ Finding a place to refine
 - counterexample- and proof-guided approaches
 - ↳ Discovering new predicates
- ⊃ Overview of state-of-the-art software MCs

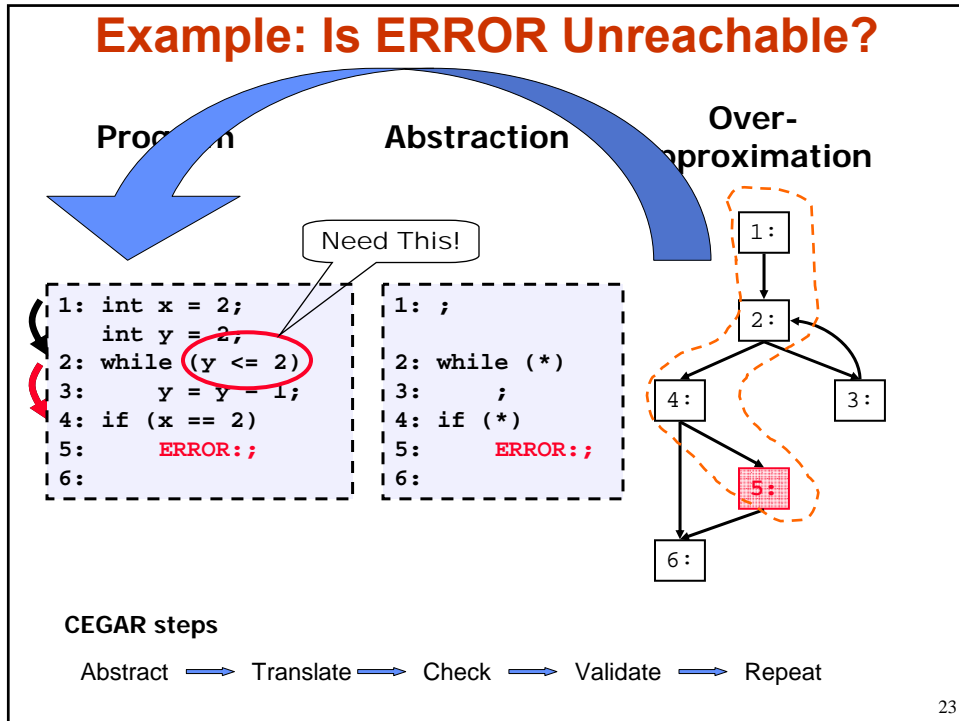
21

CounterExample Guided Abstraction Refinement (CEGAR)

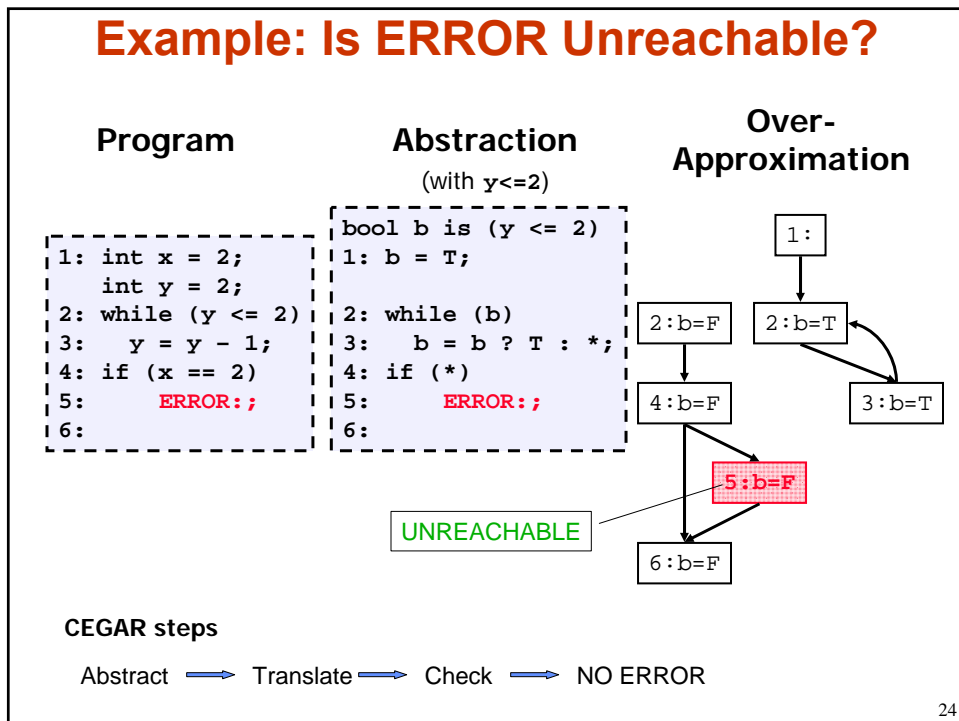


22

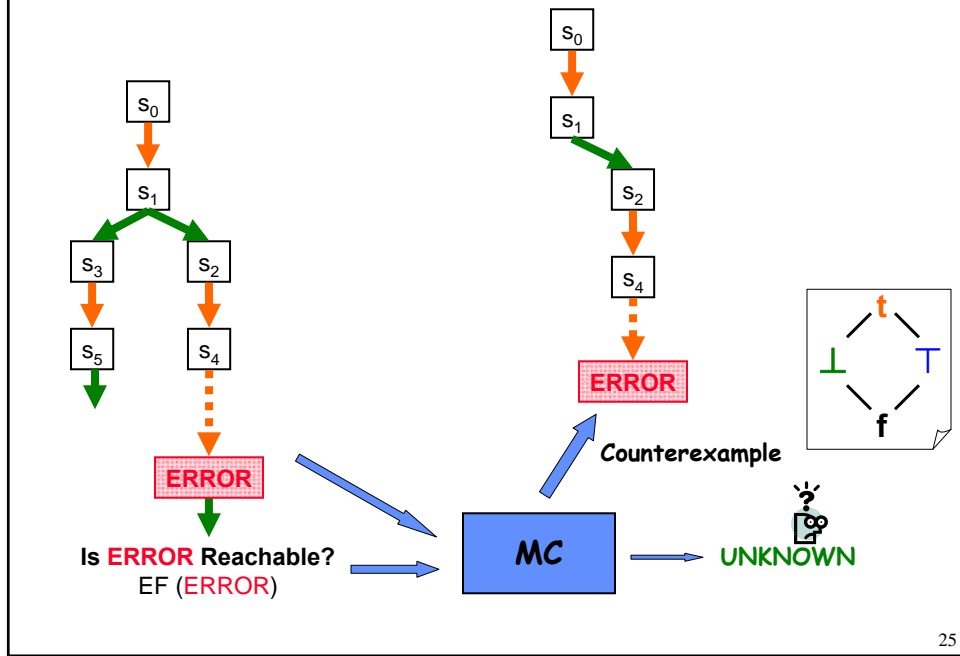
Example: Is ERROR Unreachable?



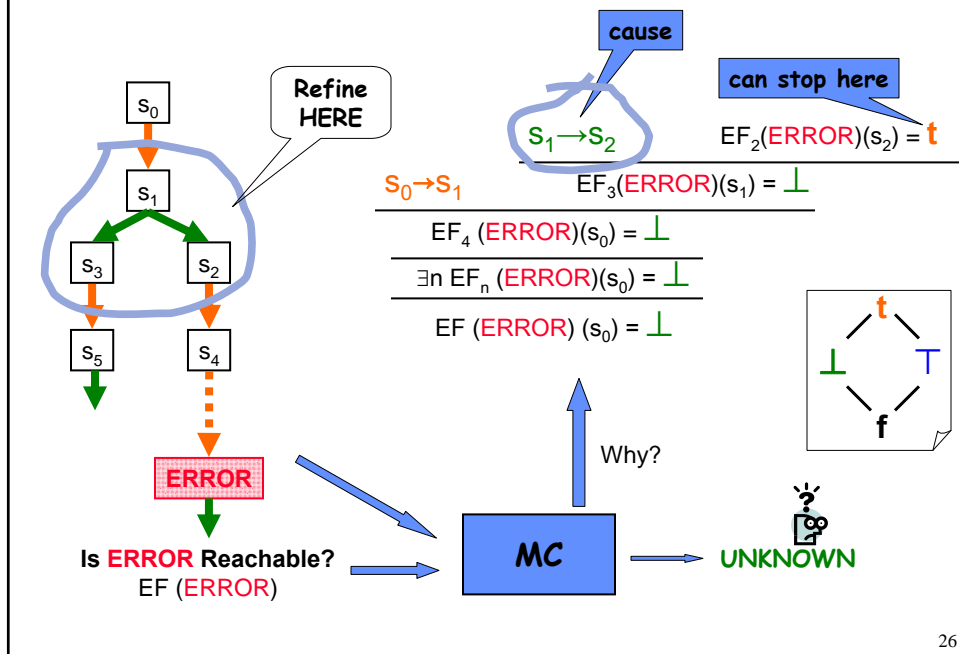
Example: Is ERROR Unreachable?



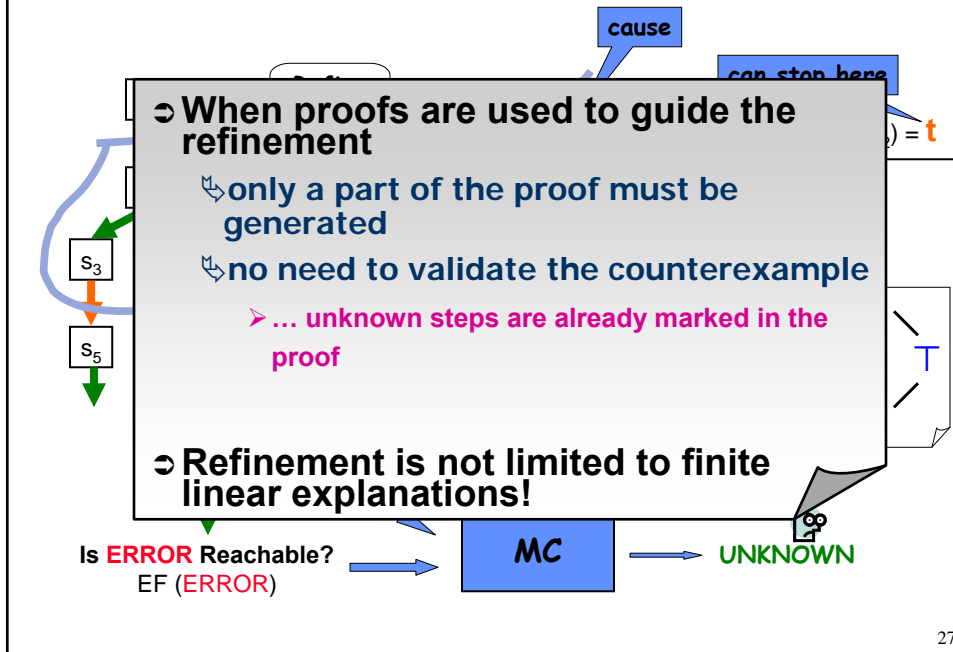
Using Cex for Refinement



Using Proofs for Refinement



Using Proofs for Refinement



Finding Refinement Predicates

Recall

- each abstract state is a conjunction of predicates
 - i.e., $y \leq 2 \wedge x = 2$ $y > 2 \wedge x \neq 2$ 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

Refinement via Weakest 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 $WP(C,p)$

An Example

s_1 is $y > 2 \wedge x == 2$

$\{s_1\}C\{x == 2\}$ is valid

s_2 is $y > 2 \wedge x == 2$

$\{s_1\}C\{y > 2\}$ is *not* valid

C is $y = y - 1$

add $WP(C, y > 2) = y > 3$

29

Summary: Software Model Checking

- ⇒ SoftMC is an effective technique for analyzing behavioral properties of software systems
- ⇒ Based on a combination of static analysis and traditional model-checking techniques
- ⇒ 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 enables to find the “right” abstraction incrementally

30

Overview of Software Model Checkers

Tools:

- ↳ YASM
- ↳ SLAM
- ↳ BLAST
- ↳ CBMC
- ↳ MAGIC
- ↳ Java PathFinder

Comparison parameters

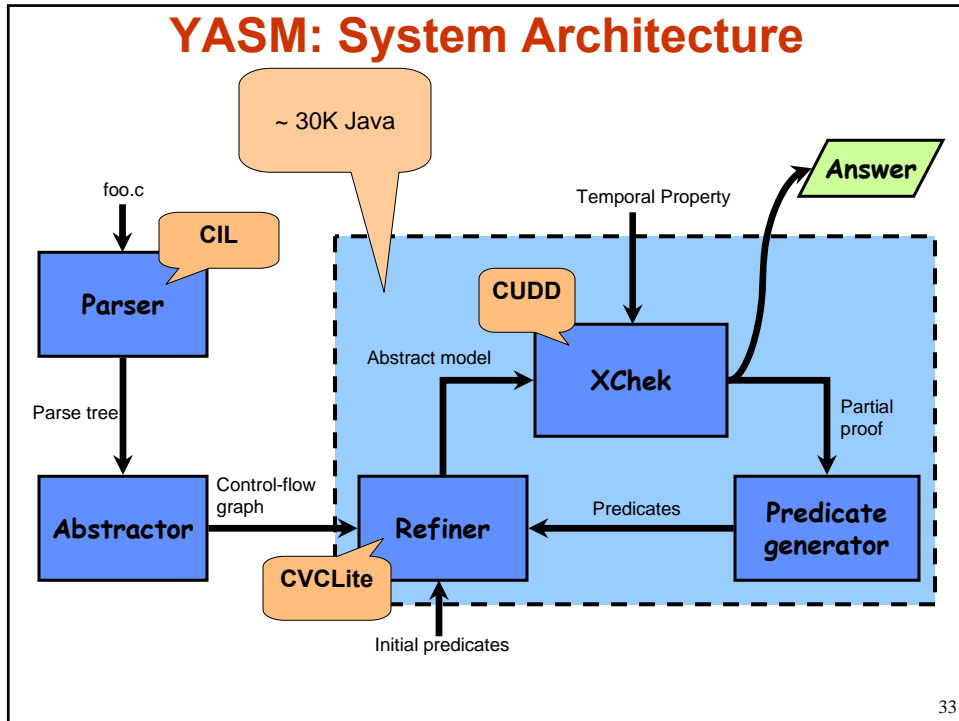
- ↳ Properties
- ↳ Types of abstraction
- ↳ Model-checking engine
- ↳ How refinement is done


31

YASM

- ↳ <http://www.cs.toronto.edu/~arie/yasm>
- ↳ Properties: CTL
- ↳ Abstraction: Predicate Over- and Under-
- ↳ MC Engine: Symbolic BDD-based
- ↳ Refinement: CTL Proof-based + WP

32



- ## Main Features of YASM
- ⊃ Checks real C programs
 - ⊃ Not biased towards verification or refutation
 - ⊃ Sound for both **True** and False answers
 - ⊃ Can check arbitrary CTL property
 - ↳ ... including liveness!
 - ⊃ Handles recursive programs 
- 34

Current Applications

- ⊃ **BLAST Benchmarks** [GC06]
 - ↳ Device drivers (4K-6K LOC)
 - ↳ Parts of OpenSSH (2K-3K LOC)
- ⊃ **Split OpenSSH (100K LOC)**
 - ↳ with UofT Security Group
- ⊃ **Detecting setuid/seteuid security flaws**
 - ↳ with UofT Security Group, in progress
- ⊃ **Concurrent “Toy” Programs**
 - ↳ Lamport’s Bakery Mutual Exclusion
 - ↳ Error detection in NASA RAX [PPV05]
- ⊃ **Finding livelock bugs**
 - ↳ “Can a library routine get stuck?”
 - ↳ with B. Cook at Microsoft Research, in progress

35

SLAM

- ⊃ Part of Windows DDK Static Driver Verifier
- ⊃ Properties: Reachability
- ⊃ Abstraction: Predicate over-approximation
- ⊃ MC Engine: Symbolic BDD-based
- ⊃ Refinement: Symbolic simulation of cexs
- ⊃ Key Features: robust, support for recursion, (almost) in production use

36

BLAST

- ⊃ <http://embedded.eecs.berkeley.edu/blast/>
- ⊃ Properties: Reachability
- ⊃ Abstraction: Predicate over-approximation
- ⊃ MC Engine: Symbolic BDD-based
 - ↳ MC and abstraction are interleaved
- ⊃ Refinement: Predicates from a proof of impossibility of a counter-example

37

SATABS & CBMC

- ⊃ <http://www.inf.ethz.ch/personal/daniekro/satabs/>
- ⊃ Properties: Bounded reachability
- ⊃ Abstraction: Predicate over-approximation
- ⊃ MC Engine: Symbolic SAT-based
- ⊃ Refinement: Symbolic simulation of cex + UNASTCORE
- ⊃ Key Features: supports precise machine arithmetic including bit-level operations

38

MAGIC

- ⊃ <http://www.cs.cmu.edu/~chaki/magic/>
- ⊃ Properties: Automata Simulation
- ⊃ Abstraction: Predicate over-approximation
- ⊃ MC Engine: SAT-based
- ⊃ Refinement: Symbolic simulation of cex
- ⊃ Key Features: support for concurrent C modules

39

Java Pathfinder

- ⊃ <http://javapathfinder.sourceforge.net/>
- ⊃ Properties: Reachability
- ⊃ Abstraction: user-provided data abstraction
- ⊃ MC Engine: Explicit state with symbolic execution
- ⊃ Refinement: None
- ⊃ Key Features: support for Java including Objects and Threads

40

References

- [BR01] T. Ball, S. Rajamani. “The SLAM Toolkit”. In CAV’01.
- [CGJLV00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. “Counterexample-Guided Abstraction Refinement”. In CAV’00.
- [GC06] A. Gurfinkel, M. Chechik. “Why Waste a Perfectly Good Abstraction”. In TACAS’06.
- [PPV05] C. Pasareanu, R. Pelanek, W. Visser. “Concrete Search with Abstract Matching and Refinement”. In CAV’05.

41

Acknowledgements

We thank the model-checking group at CMU (Ed Clarke) and the BANDERA project (Matt Dwyer, Corina Pasareanu) for the source of and the inspiration for some of our slides.

These slides were originally created for a model-checking tutorial at Formal Methods ’06.
Presenters: Marsha Chechik & Arie Gurfinkel

42