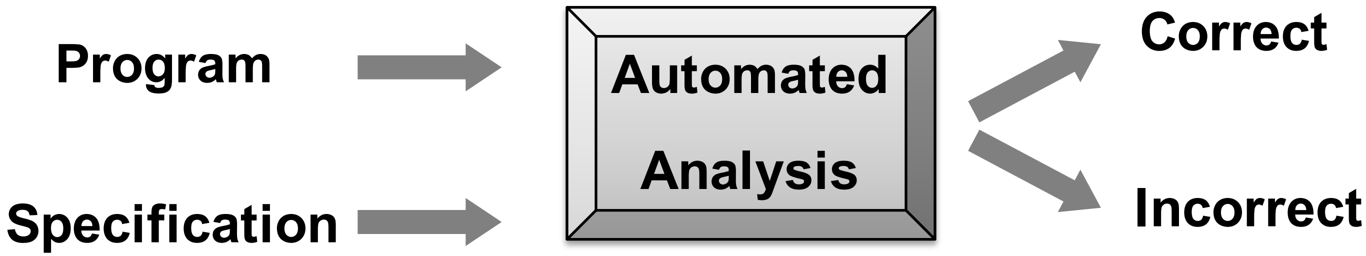
Automated Program Analysis with Software Model Checking

Arie Gurfinkel
Software Engineering Institute
Carnegie Mellon University

February, 2016

Static Program Analysis



Reasoning statically about behavior of a program without executing it

- compile-time analysis
- exhaustive, considers all possible executions under all possible environments and inputs

The *algorithmic* discovery of *properties* of program by *inspection* of the *source text*

Manna and Pnueli, “Algorithmic Verification”

Also known as static analysis, program verification, formal methods, etc.

Turing, 1936: “undecidable”

Undecidability

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

Living with Undecidability

"Algorithms" that occasionally diverge

Limit programs that can be analyzed

- finite-state, loop-free

Partial (unsound) verification

- analyze only some executions up-to a fixed number of steps

Incomplete verification / Abstraction

- analyze a superset of program executions

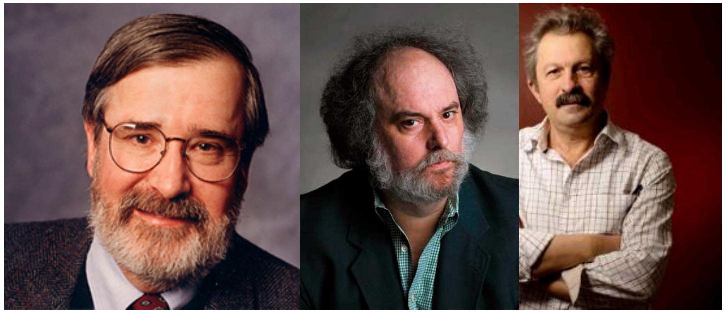
Programmer Assistance

- annotations, pre-, post-conditions, inductive invariants

(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

- Statespace explosion

Model Checking since 1981

1981 Clarke / Emerson: CTL Model Checking
Sifakis / Quielle

10^5

1982 EMC: Explicit Model Checker
Clarke, Emerson, Sistla

1990 Symbolic Model Checking
Burch, Clarke, Dill, McMillan

10^{100}

1992 SMV: Symbolic Model Verifier
McMillan

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

1998 Bounded Model Checking using SAT
Biere, Clarke, Zhu

10^{1000}

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

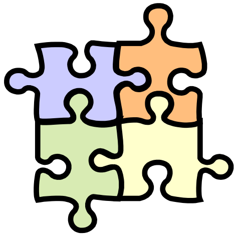
Model Checking since 1981

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

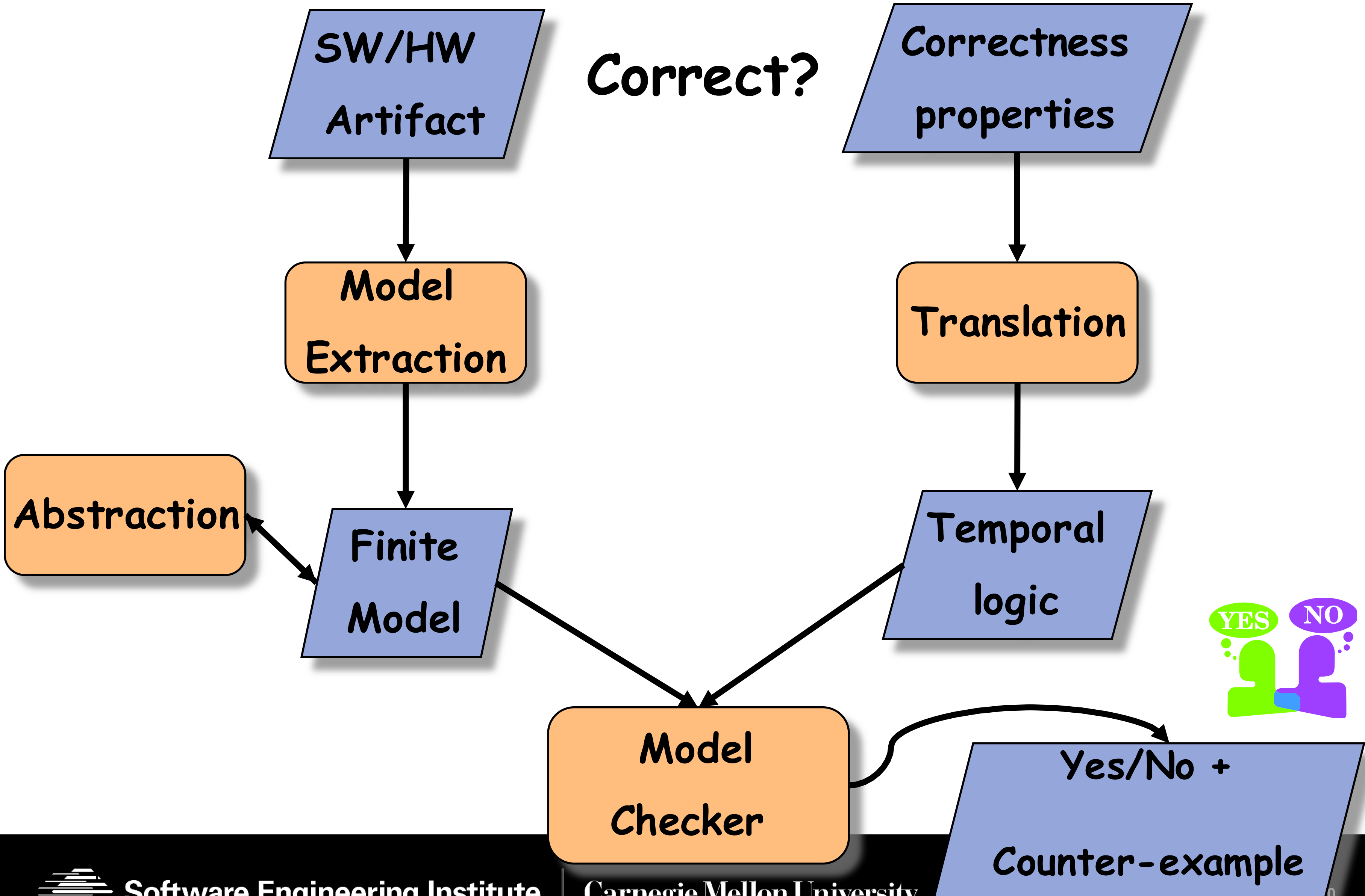
→ **CBMC**

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

→ **SLAM, MAGIC, BLAST, …**

Temporal Logic Model Checking

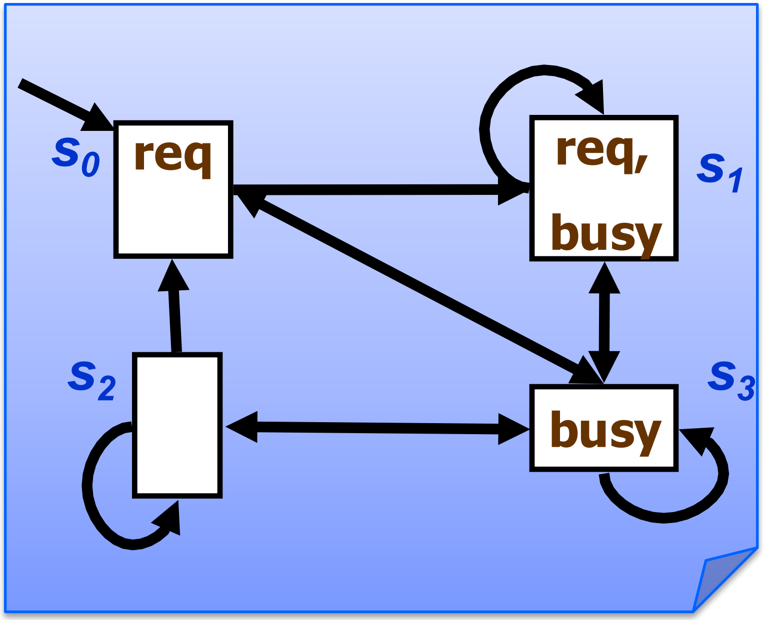
Temporal Logic Model Checking



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

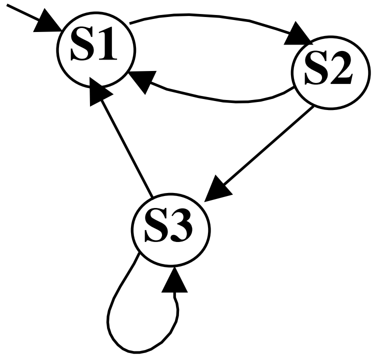
- *"it is raining"* – *rain*
- "it will rain tomorrow" – $\Box$*rain*
  - it is raining in all possible futures
- "it might rain tomorrow" – $\Diamond$*rain*
  - it is raining in some possible futures

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

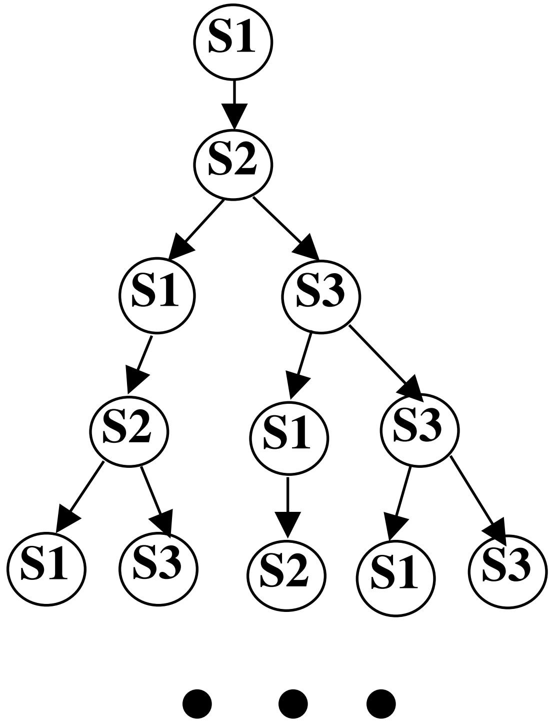
Temporal logic is a modal logic that adds temporal modalities: next, always, eventually, and until

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

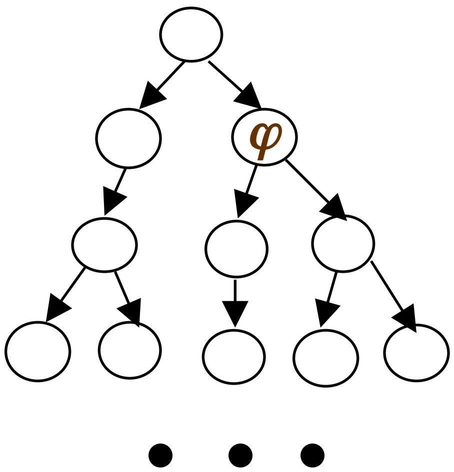
EF φ : along some path, φ holds in a future state

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

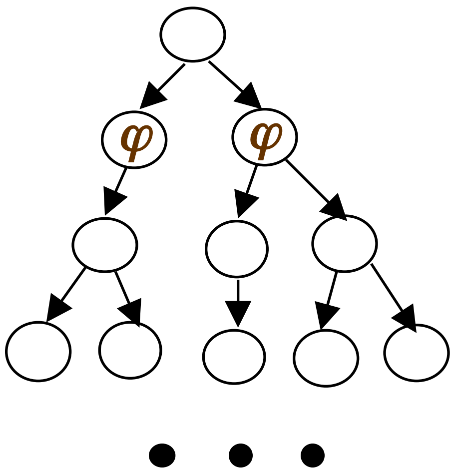
EG φ : along some path, φ holds in every state

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

Examples: EX and AX

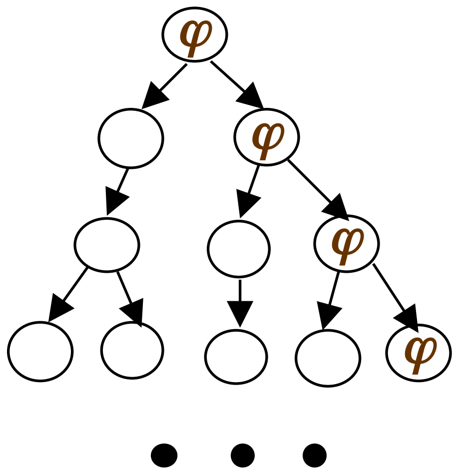


EX φ (exists next)

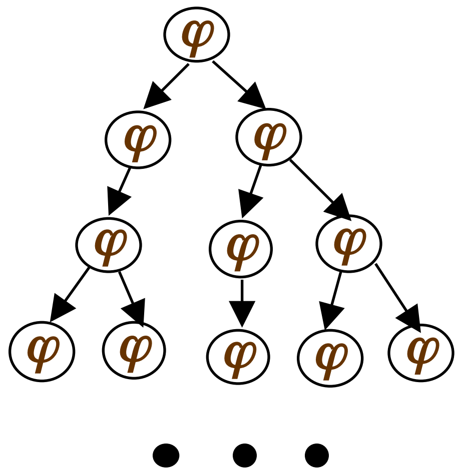


AX φ (all next)

Examples: EG and AG

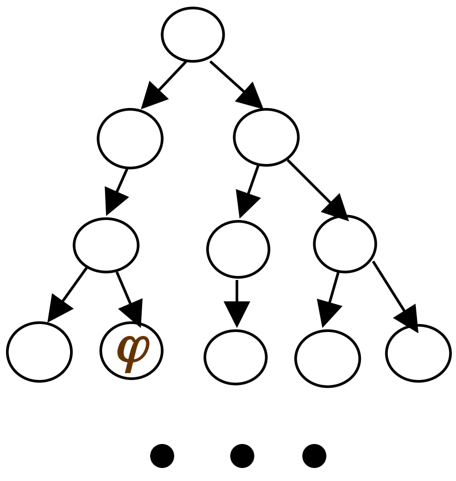


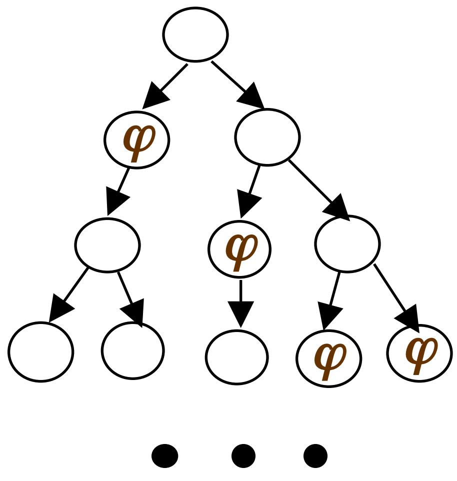
EG φ (exists global)



AG φ (all global)

Examples: EF and AF

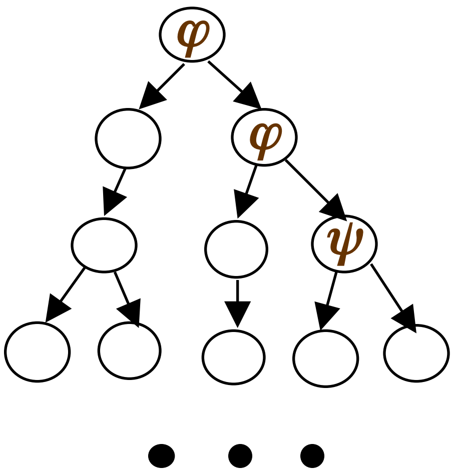




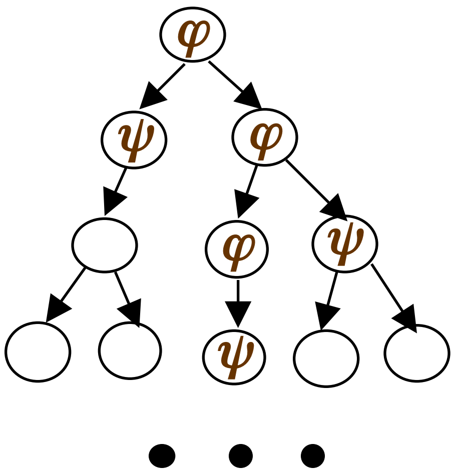
EF φ (exists future)

AF φ (all future)

Examples: EU and AU



E[φ U ψ] (exists until)



A[φ U ψ] (all until)

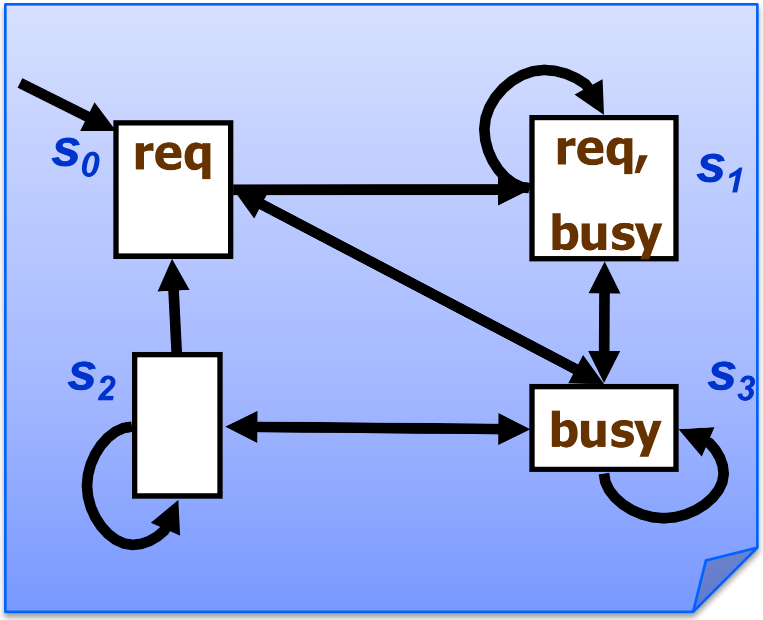
CTL Examples

Properties that hold:

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

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. 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,

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

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

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

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

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

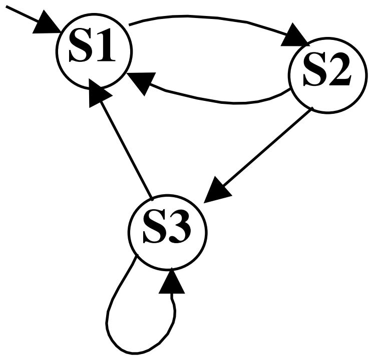
$EF\ \varphi = \exists \pi \cdot \exists i \cdot \pi^i \models \varphi$

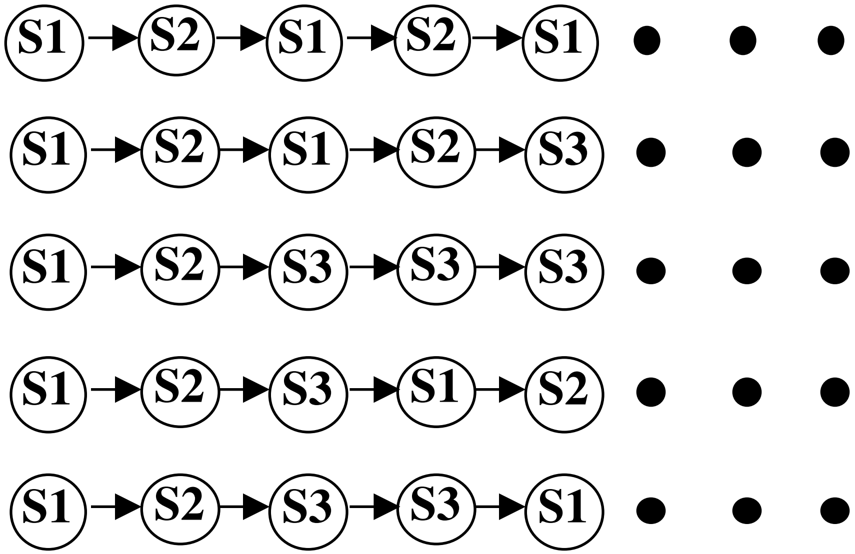
$A[\varphi\ U\ \psi] = \forall \pi \cdot \exists i \cdot \pi^i \models \psi \wedge \forall j \cdot 0 \leq j < i \Rightarrow \pi^j \models \varphi$

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

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

$\pi \models X\, \varphi$ iff $\pi^1 \models \varphi$

$\pi \models G\, \varphi$ iff $\forall i \cdot \pi^i \models \varphi$

$\pi \models F\varphi$ iff $\exists i \cdot \pi^i \models \varphi$

$\pi \models \varphi\, U\, \psi$ 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

Expressing Properties in LTL

Good for safety (G ¬) and liveness (F) properties

Express:

- When a request occurs, it will eventually be acknowledged
- Each path contains infinitely many *q*'s
- At most a finite number of states in each path satisfy ¬*q* (or property *q* eventually stabilizes)
- Action *s* precedes *p* after *q*
  – 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

State Explosion

How fast do Kripke structures grow?

- Composing linear number of structures yields exponential growth!

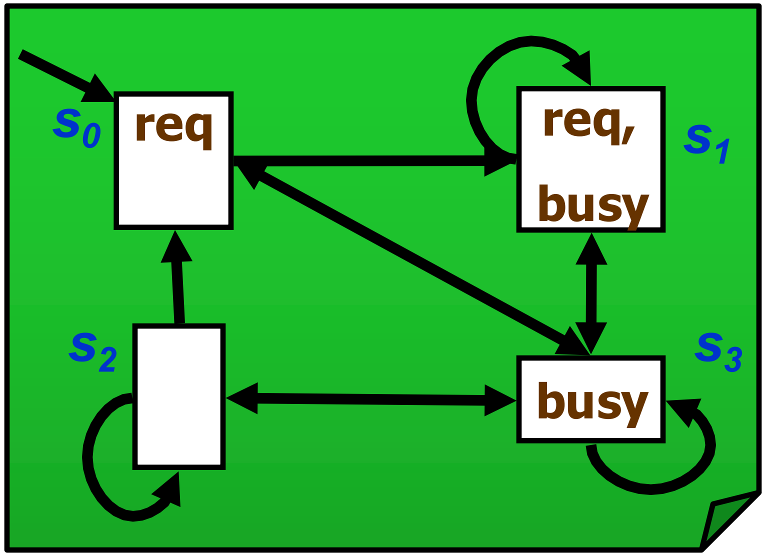
How to deal with this problem?

- 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 $\neg req \wedge \neg busy \wedge \neg req' \wedge busy'$
  - 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 behaviours of a downscaled system

  than by

  testing some behaviours of the full system

SAT and SMT

Boolean Satisfiability

Let V be a set of variables

A *literal* is either a variable v in V or its negation ~v

A *clause* is a disjunction of literals

- e.g., (v1 || ~v2 || v3)

A Boolean formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses

- e.g., (v1 || ~v2) && (v3 || v2)

An *assignment* *s* of Boolean values to variables *satisfies* a clause *c* if it evaluates at least one literal in *c* to true

An assignment *s* *satisfies* a formula *C* in CNF if it satisfies every clause in *C*

Boolean Satisfiability Problem (SAT):

- determine whether a given CNF C is satisfiable

CNF Examples

CNF 1

- ~b
- ~a || ~b || ~c
- a
- sat: s(a) = True; s(b) = False; s(c) = False

CNF 2

- ~b
- ~a || b || ~c
- a
- ~a || c
- unsat

Algorithms for SAT

SAT is NP-complete

DPLL (Davis-Putnam-Logemman-Loveland, ‘60)

- smart enumeration of all possible SAT assignments
- worst-case EXPTIME
- alternate between deciding and propagating variable assignments

CDCL (GRASP ‘96, Chaff ‘01)

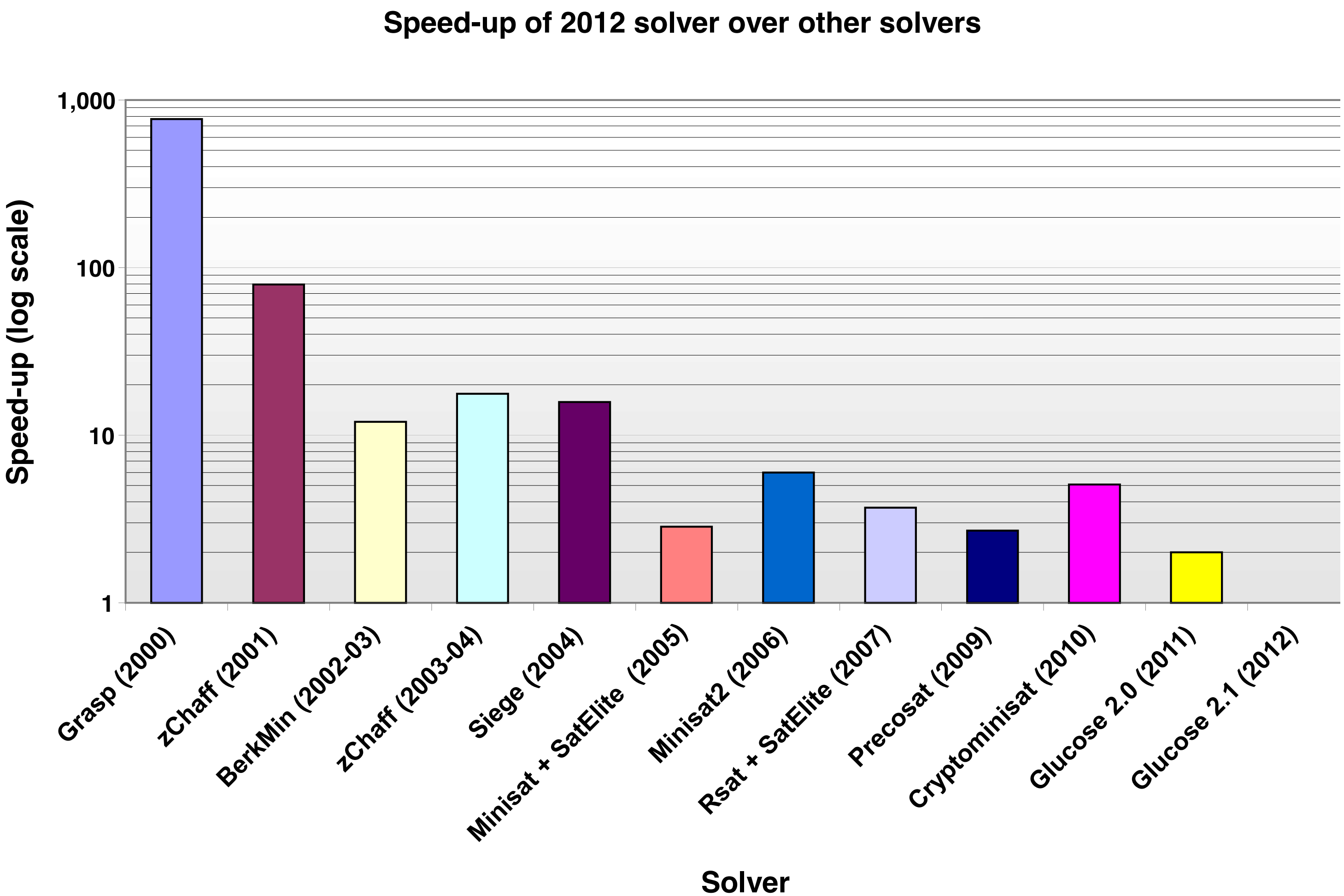
- conflict-driven clause learning
- extends DPLL with
  - smart data structures, backjumping, clause learning, heuristics, restarts…
- scales to millions of variables
- N. Een and N. Sörensson, “An Extensible SAT-solver”, in SAT 2013.

DPLL by Example

DPLL Example by Prof. Cesare Tinelli

From http://homepage.cs.uiowa.edu/~tinelli/classes/196/Fall09/notes/dpll.pdf

Some Experience with SAT Solving



from M. Vardi, https://www.cs.rice.edu/~vardi/papers/highlights15.pdf

SMT: Satisfiability Modulo Theory

Satisfiability of Boolean formulas over atoms in a theory

- e.g., (x < 0) && (x >= 0)

Extends syntax of Boolean formulas with functions and predicates

- +, -, div, select, store, bvadd, etc.

Existing solvers support many theories useful for program analysis

- Equality and Uninterpreted Functions: f(x)
- Real/Integer Linear Arithmetic: x + 2*y <= 3
- Unbounded Arrays: a[i], a[i := v]
- Bitvectors (a.k.a. machine integers): x >> 3, x/3
- Floating point: 3.0 * x
- …

SMT-LIB: http://smt-lib.org

International initiative for facilitating research and development in SMT

Provides rigorous definition of syntax and semantics for theories

SMT-LIB syntax

- based on s-expressions (LISP-like)
- common syntax for interpreted functions of different theories
  - e.g. (and (= x y) (<= (* 2 x) z))
- commands to interact with the solver
  - (declare-fun …) declares a constant/function symbol
  - (assert p) conjoins formula p to the curent context
  - (check-sat) checks satisfiability of the current context
  - (get-model) prints current model (if the context is satisfiable)

- see examples at http://rise4fun.com/z3

SMT Example

Is this formula satisfiable?

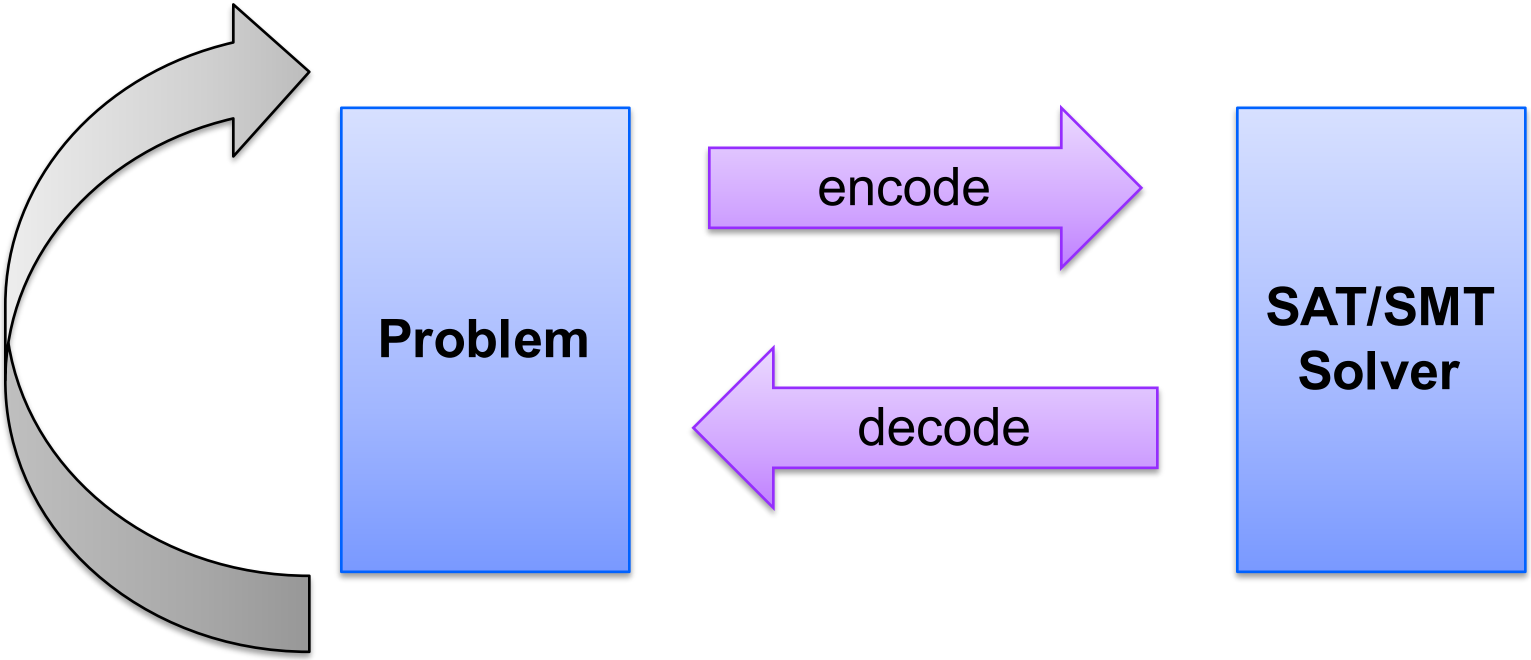
```
; This example illustrates basic arithmetic and
; uninterpreted functions

(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (>= (* 2 x) (+ y z)))
(declare-fun f (Int) Int)
(declare-fun g (Int Int) Int)
(assert (< (f x) (g x x)))
(assert (> (f y) (g x x)))
(check-sat)
(get-model)
(push)
(assert (= x y))
(check-sat)
(pop)
(exit)
```

http://rise4fun.com/z3

SAT/SMT Revolution

Solve any computational problem by effective reduction to SAT/SMT

- iterate as necessary

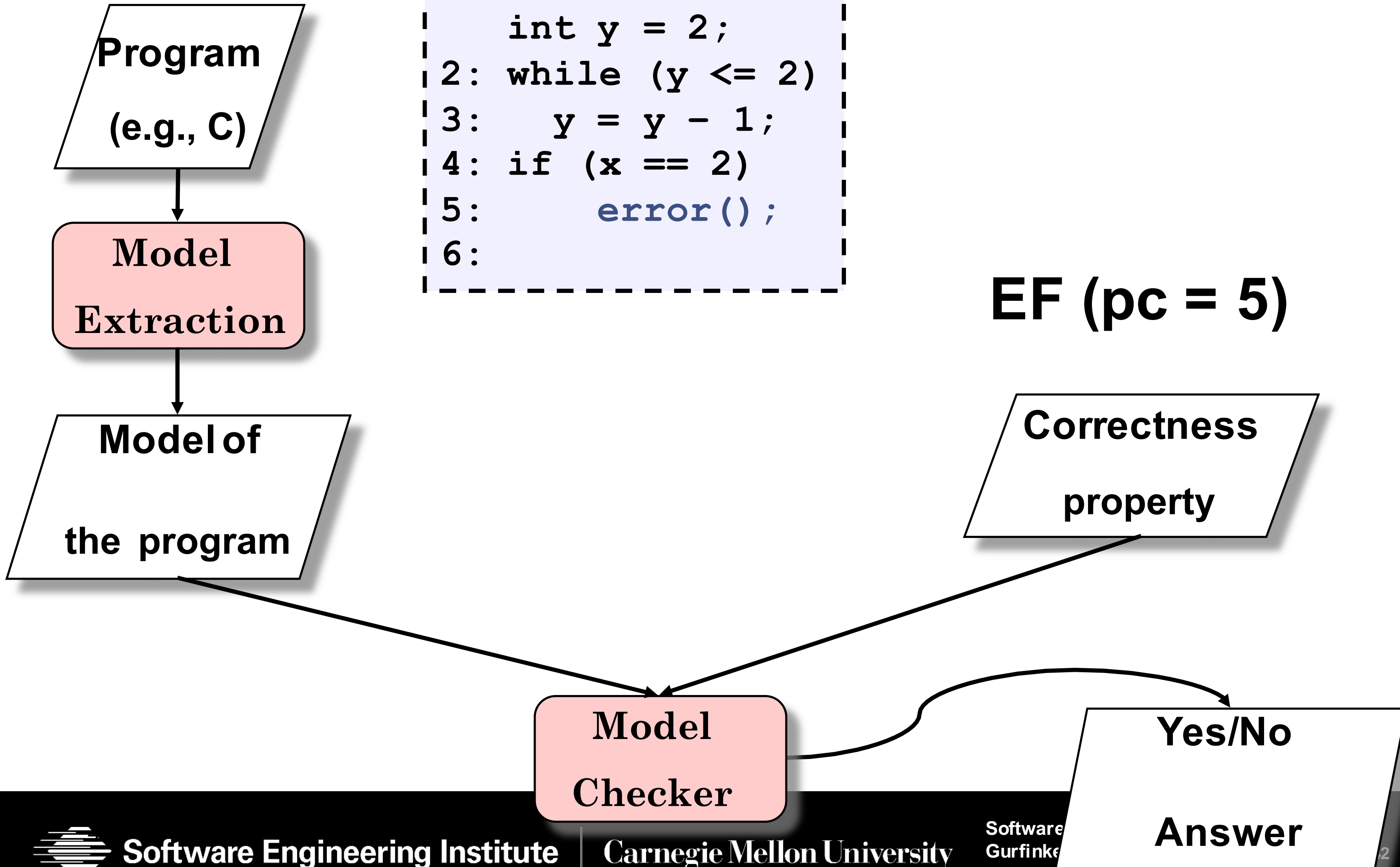


Software Model Checking

Software Model Checking

```
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)



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

pc	x	y	…
3	1	3	…

Step

pc	x	y	…
2	1	2	…

Property: EF (pc = 5)

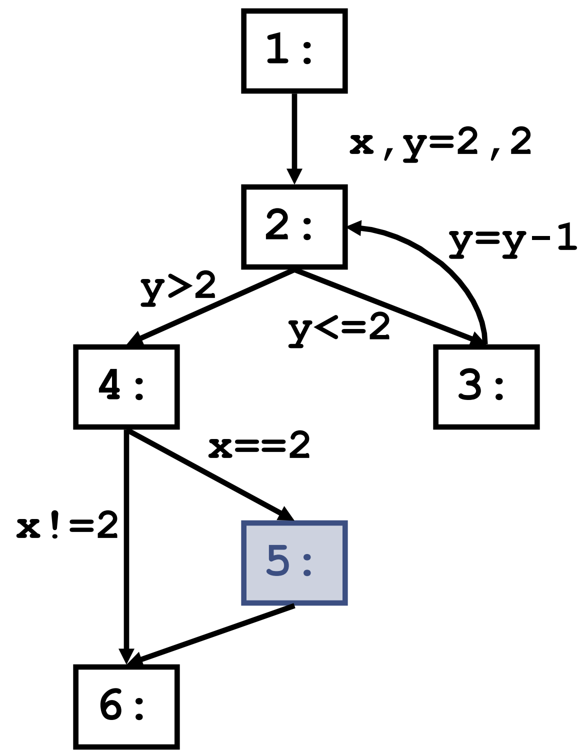
Programs as Control Flow Graphs

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

- but it is a 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, etc.

Programing languages already provide convenient primitives to describe behavior

- programming languages are extended to modeling and specification languages by adding three 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/enviroment

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);

}
```

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); }
```

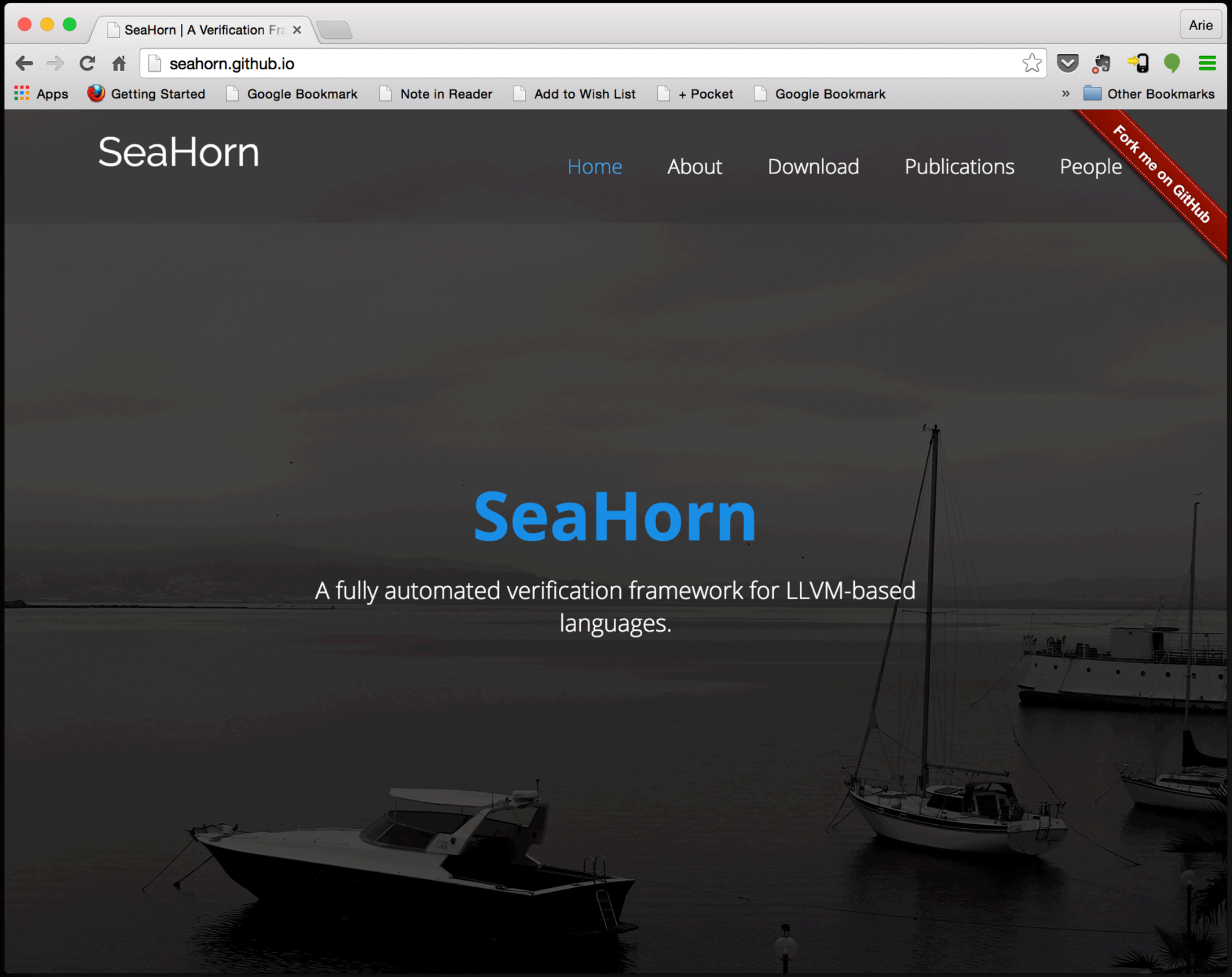
Assume must either be checked with assert or used as an idiom:

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

Software Model Checking Workflow

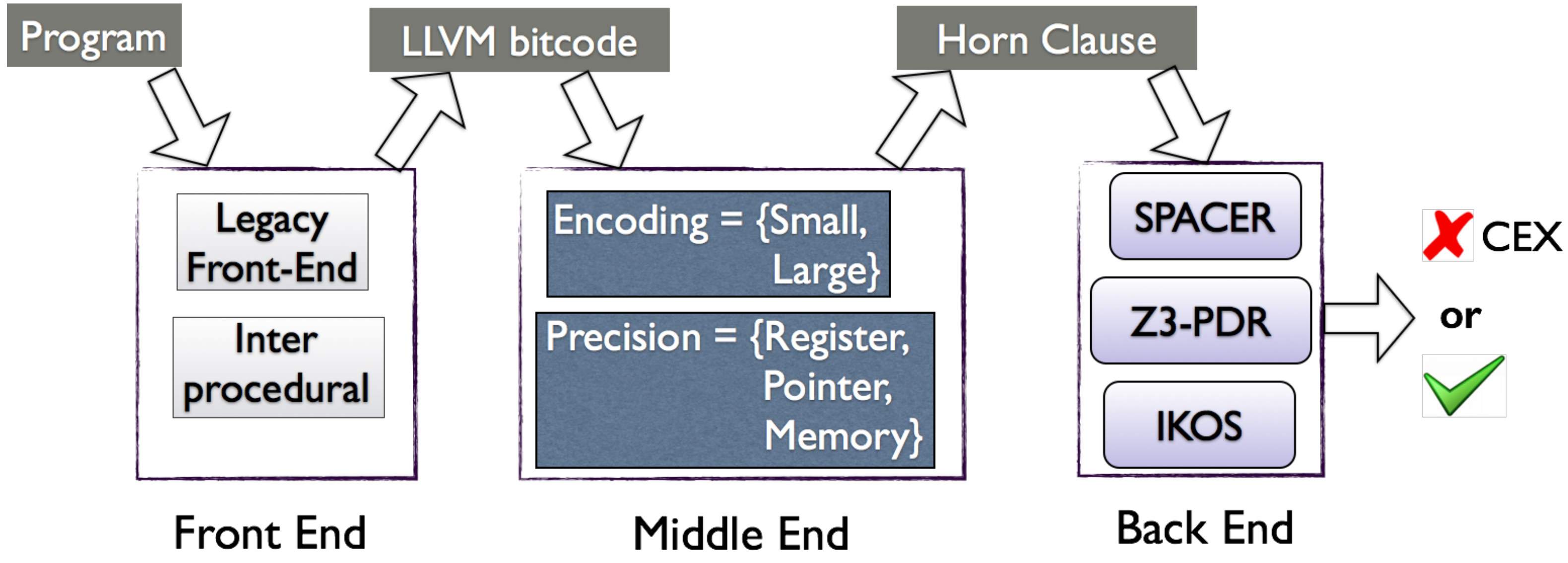
1. Identify module to be analyzed
   – e.g., function, component, device driver, library, etc.
2. Instrument with property assertions
   – e.g., buffer overflow, proper API usage, proper state change, etc.
   – might require significant changes in the program to insert necessary monitors
3. Model environment 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



http://seahorn.github.io

SeaHorn Verification Framework



Automated C program verifier for

- buffer- and integer-overflow, API usage rules, and user-specified assertions

Integrates with industrial-strength LLVM compiler framework

Based on our research in software model checking and abstract interpretation

Developed jointly by the SEI, CMU CyLab, and NASA Ames

SeaHorn Usage

`> sea pf FILE.c`

Outputs `sat` for unsafe (has counterexample); `unsat` for safe

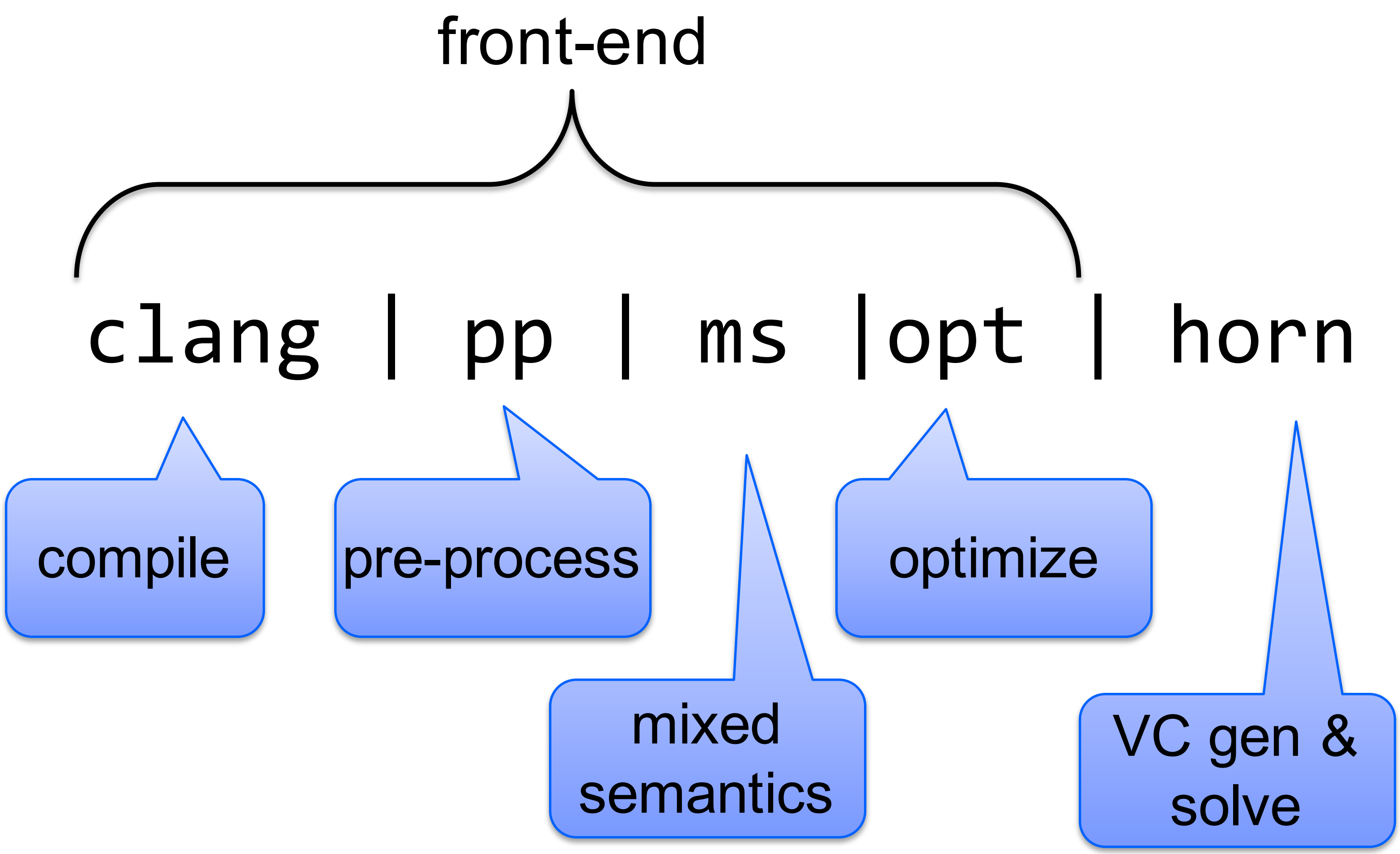
Additional options

- `--cex=trace.xml` outputs a counter-example in SV-COMP'15 format
- --show-invars displays computed invariants
- `--track={reg,ptr,mem}` track registers, pointers, memory content
- `--step={large,small}` verification condition step-semantics
  - *small* == basic block, *large* == loop-free control flow block
- `--inline` inline all functions in the front-end passes

Additional commands

- `sea smt` – generates CHC in extension of SMT-LIB2 format
- `sea clp` -- generates CHC in CLP format (under development)
- `sea lfe-smt` – generates CHC in SMT-LIB2 format using legacy front-end

Verification Pipeline



Current Application

Verification of resource usage rules in Linux device drivers

- e.g., locks are acquired and released, buffers are initialized, etc.
- specifications and verification environment provided by the Open-Source Linux Device Verification (LDV) project

NASA's Lunar Atmosphere and Dust Environment Explorer (LADEE)

- conformance of auto-generated code with Simulink models
- absence of buffer overflows

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

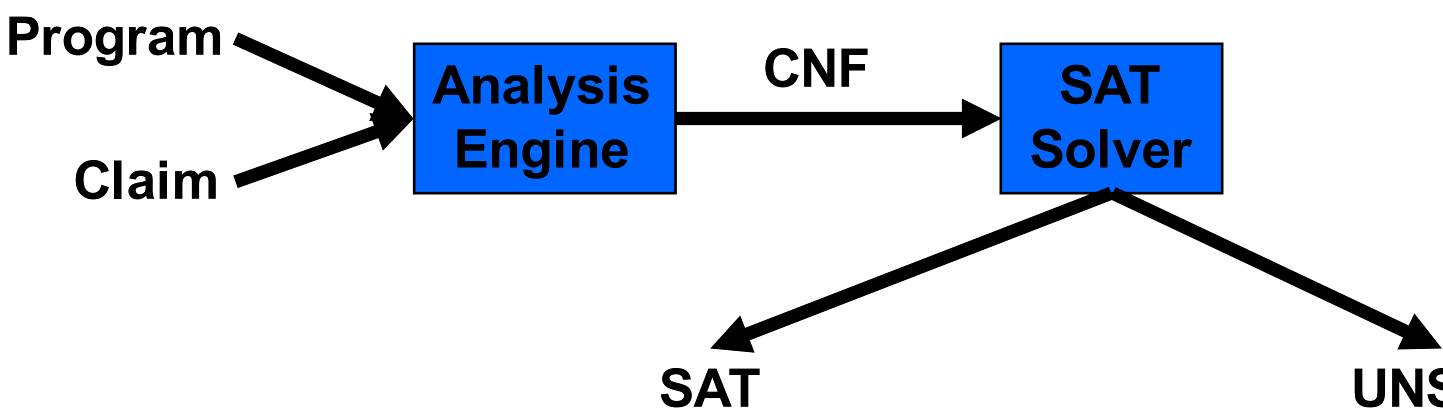
Interpolation-based Model Checking (IMC)

- 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

Bounded Model Checking

Bug Catching with SAT-Solvers

Main Idea: Given a program and a claim use a SAT-solver to find whether there exists an execution that violates the claim.



(counterexample exists) **(no counterexample found)**

Programs and Properties

Arbitrary ANSI-C programs

- With bitvector arithmetic, dynamic memory, pointers, …

Simple Safety Properties

- Array bound checks (i.e., buffer overflow)
- Division by zero
- Pointer checks (i.e., NULL pointer dereference)
- Arithmetic overflow
- User supplied assertions (i.e., `assert (i > j)`)

Why use a SAT Solver?

SAT Solvers are very efficient

Analysis is completely automated

Analysis as good as the underlying SAT solver

Allows support for many features of a programming language

- bitwise operations, pointer arithmetic, dynamic memory, type casts

A (very) simple example (1)

Program

```
int x;
int y=8,z=0,w=0;
if (x)
  z = y - 1;
else
  w = y + 1;
assert (z == 7 ||
        w == 9)
```

Constraints

y = 8,

z = x ? y – 1 : 0,

w = x ? 0 :y + 1,

z != 7,

w != 9

UNSAT

no counterexample

assertion always holds!

A (very) simple example (2)

Program

```
int x;
int y=8,z=0,w=0;
if (x)
  z = y - 1;
else
  w = y + 1;
assert (z == 5 ||
        w == 9)
```

Constraints

y = 8,

z = x ? y – 1 : 0,

w = x ? 0 :y + 1,

z != 5,

w != 9

SAT

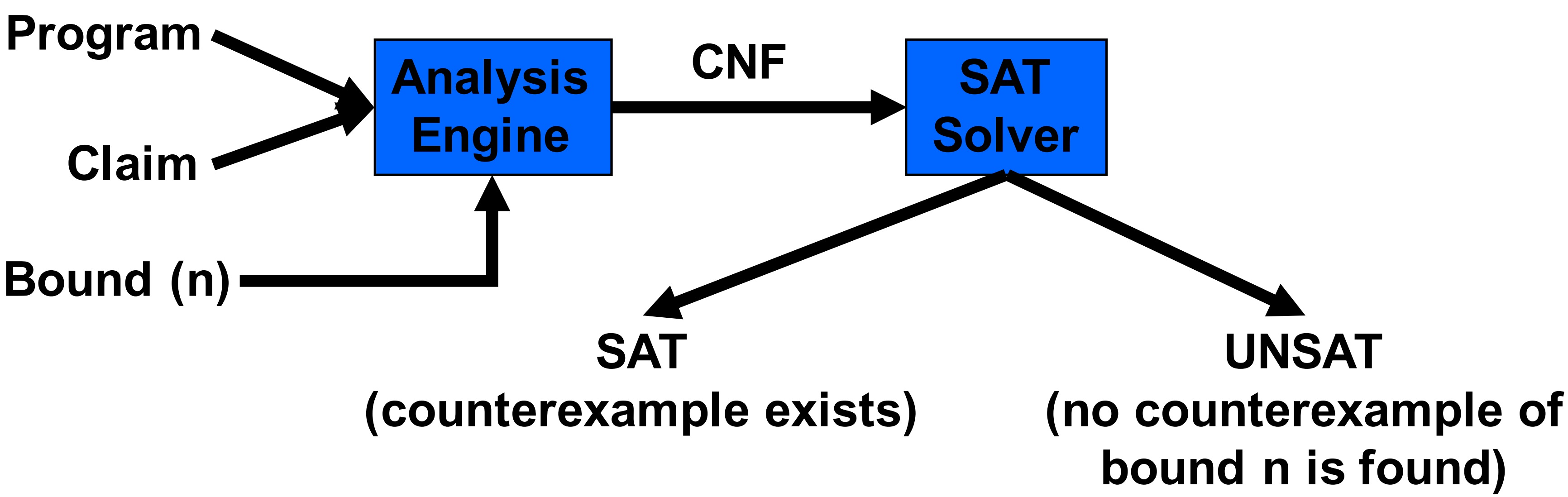
counterexample found!

y = 8, x = 1, w = 0, z = 7

What about loops?!

SAT Solver can only explore finite length executions!

Loops must be bounded (i.e., the analysis is unsound)



CBMC: C Bounded Model Checker

Started at CMU by Daniel Kroening and Ed Clarke

Available at: http://www.cprover.org/cbmc

- On Ubuntu: `apt-get install cbmc`

Supported platforms: Windows, Linux, OSX

Has a command line, Eclipse CDT, and Visual Studio interfaces

Scales to programs with over 30K LOC

Found previously unknown bugs in MS Windows device drivers

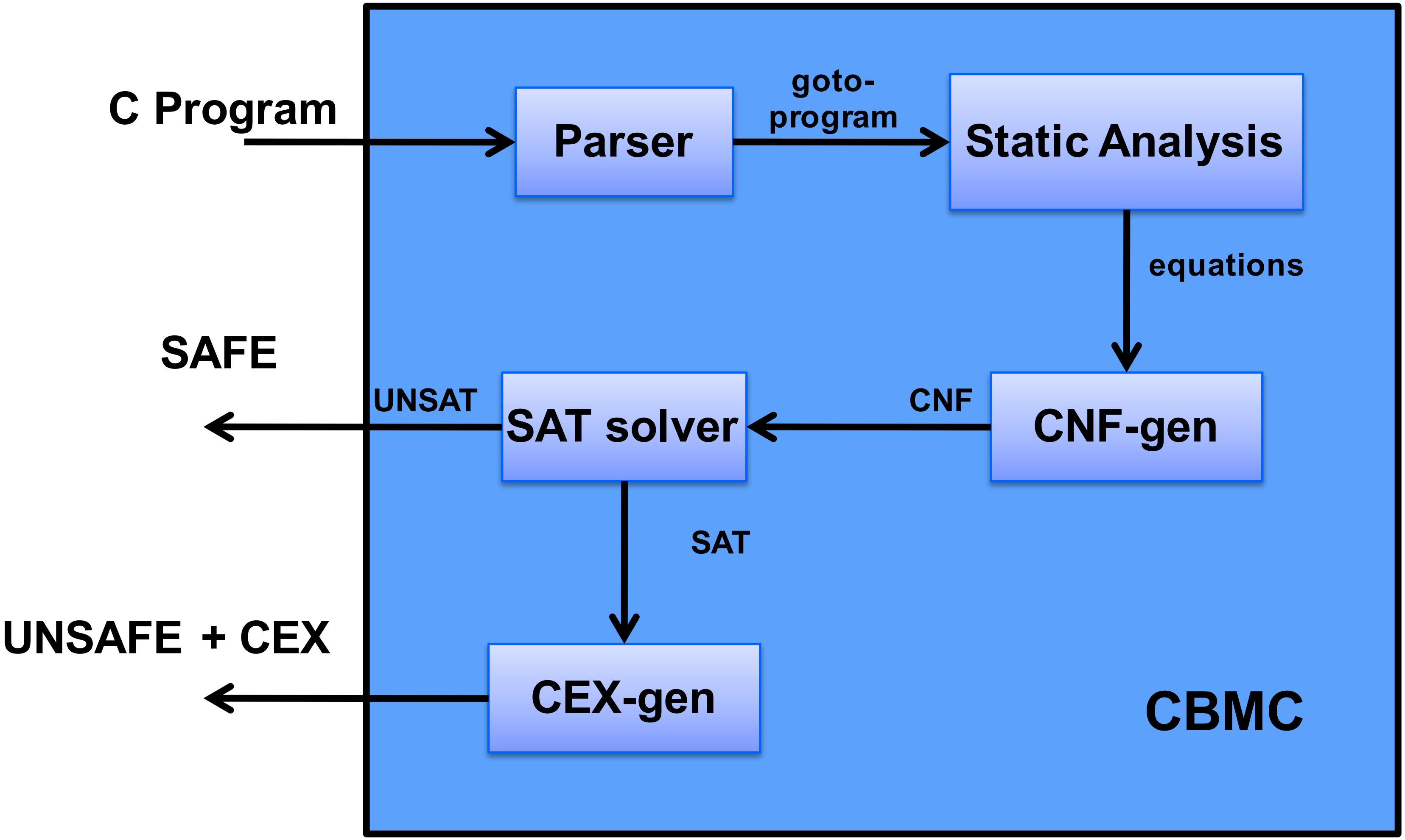
How does it work

Transform a programs into a set of equations

1. Simplify control flow
2. Unwind all of the loops
3. Convert into Single Static Assignment (SSA)
4. Convert into equations
5. Bit-blast
6. Solve with a SAT Solver
7. Convert SAT assignment into a counterexample

CBMC: Bounded Model Checker for C

A tool by D. Kroening/Oxford and Ed Clarke/CMU



Control Flow Simplifications

- **All side effect are removed**
  - `e.g., j=i++` **becomes** `j=i;i=i+1`

- **Control Flow is made explicit**
  - `continue`, `break` **replaced by** `goto`

- **All loops are simplified into one form**
  - `for`, `do while` **replaced by** `while`

Loop Unwinding

- **All loops are unwound**
  - can use different unwinding bounds for different loops
  - to check whether unwinding is sufficient special "unwinding assertion" claims are added

- **If a program satisfies all of its claims and all unwinding assertions then it is correct!**

- **Same for backward `goto` jumps and recursive functions**

Loop Unwinding

```
void f(...) {
  ...
  while(cond) {
    Body;
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Loop Unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    while(cond) {
      Body;
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Loop Unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond) {
        Body;
      }
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Unwinding assertion

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        while(cond) {
          Body;
        }
      }
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Assertion inserted after last iteration: violated if program runs longer than bound permits

Unwinding assertion

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        assert(!cond);

      }
    }
  }
  Remainder;
}
```

Unwinding assertion

while() loops are unwound iteratively

Break / continue replaced by goto

Assertion inserted after last iteration: violated if program runs longer than bound permits

Sound results!

Example: Sufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 2)
    j = j + 1;
  Remainder;
}
```

unwind = 3

```
void f(...) {
  j = 1
  if(j <= 2) {
    j = j + 1;
    if(j <= 2) {
      j = j + 1;
      if(j <= 2) {
        j = j + 1;
        assert(!(j <= 2));
      }
    }
  }
  Remainder;
}
```

Example: Insufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 10)
    j = j + 1;
  Remainder;
}
```

unwind = 3

```
void f(...) {
  j = 1
  if(j <= 10) {
    j = j + 1;
    if(j <= 10) {
      j = j + 1;
      if(j <= 10) {
        j = j + 1;
        assert(!(j <= 10));
        }
      }
    }
  }
  Remainder;
}
```

Transforming Loop-Free Programs Into Equations (1)

Easy to transform when every variable is only assigned once!

Program

```
x = a;
y = x + 1;
z = y - 1;
```

Constraints

x = a &&

y = x + 1 &&

z = y – 1 &&

Transforming Loop-Free Programs Into Equations (2)

When a variable is assigned multiple times,
use a new variable for the RHS of each assignment

Program

```
x=x+y;
x=x*2;
a[i]=100;
```

ρ

SSA Program

$x_1 = x_0 + y_0;$

$x_2 = x_1 * 2;$

$a_1[i_0] = 100;$

What about conditionals?

Program

```
if (v)
   x = y;
else
   x = z;

w = x;
```

ρ

SSA Program

if (v_0)

   x_0 = y_0;

else

   x_1 = z_0;

w_1 = x??;

What should 'x' be?

What about conditionals?

Program

```
if (v)
   x = y;
else
   x = z;

w = x;
```

ρ

SSA Program

$$
\begin{array}{l}
\texttt{if } (v_0) \\
\quad x_0 = y_0; \\
\texttt{else} \\
\quad x_1 = z_0; \\
x_2 = v_0 \; ? \; x_0 : x_1; \\
w_1 = x_2
\end{array}
$$

For each join point, add new variables with selectors

Adding Unbounded Arrays

$$v_\alpha[a] = e \quad \xrightarrow{\rho} \quad v_\alpha = \lambda i : \begin{cases} \rho(e) & : \; i = \rho(a) \\ v_{\alpha-1}[i] & : \; \text{otherwise} \end{cases}$$

Arrays are updated “whole array” at a time

A[1] = 5;	**$A_1 = \lambda$ i : i == 1 ? 5 : A_0[i]**
A[2] = 10;	**$A_2 = \lambda$ i : i == 2 ? 10 : A_1[i]**
A[k] = 20;	**$A_3 = \lambda$ i : i == k ? 20 : A_2[i]**

Examples:

A_2[2] == 10 **A_2[1]==5** **A_2[3] == A_0[3]**

A_3[2] == (k==2 ? 20 : 10)

Uses only as much space as there are uses of the array!

Example

```
int main() {
   int x, y;
   y=8;
   if(x)
     y--;
   else
     y++;

   assert
       (y==7 ||
       y==9);
}
```

ρ

```
int main() {
   int x, y;
   y1=8;
   if(x0)
     y2=y1-1;
   else
     y3=y1+1;
   y4= x0?y2:y3;
   assert
       (y4==7 ||
       y4==9);
}
```

$$
\begin{aligned}
(\quad & y_1 = 8 \\
\wedge \quad & y_2 = y_1 - 1 \\
\wedge \quad & y_3 = y_1 + 1 \\
\wedge \quad & y_4 = x_0 \,?\, y_2 : y_3 \ ) \\
\Longrightarrow \quad & (y_4 = 7 \vee y_4 = 9)
\end{aligned}
$$

Pointers

While unwinding, record right hand side of assignments to pointers

This results in very precise points-to information

- Separate for each pointer
- Separate for each instance of each program location

Dereferencing operations are expanded into case-split on pointer object (not: offset)

- Generate assertions on offset and on type

Pointer data type assumed to be part of bit-vector logic

- Consists of pair <object, offset>

BMC: Summary

An effective way to look for bugs

- reduce analysis to SAT/SMT
- creating effective and precise encoding is very hard

Mature tools available from several academic groups

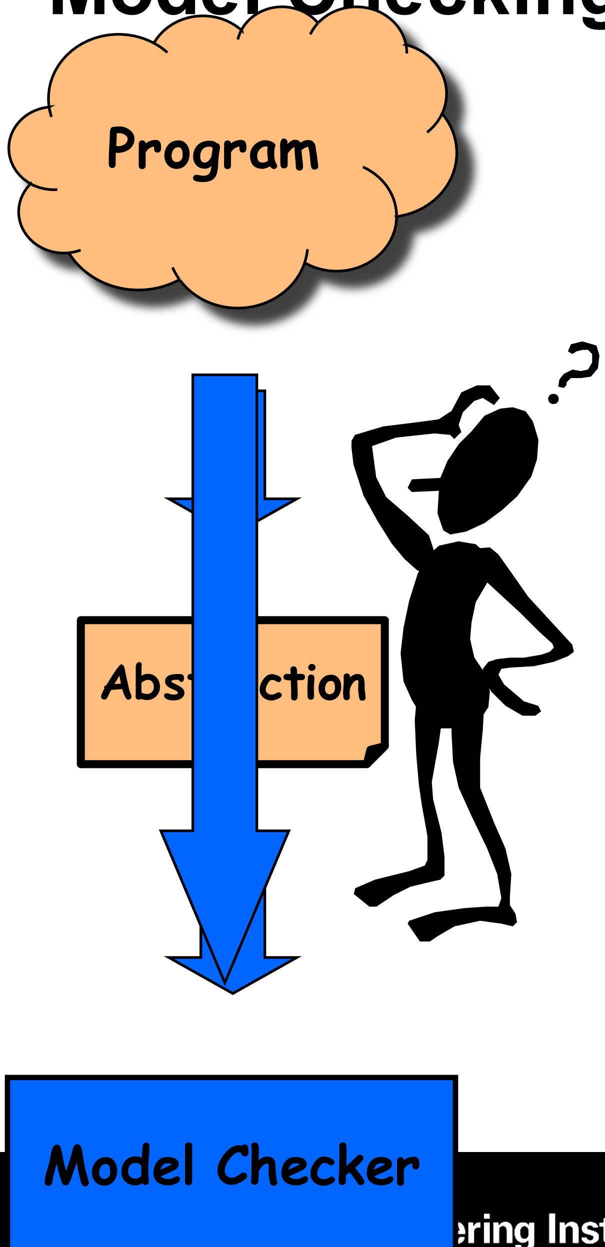
- CBMC: http://www.cprover.org/cbmc/
- LLBMC: http://llbmc.org/

Starting point for many other approaches

- deductive verification: user provides inductive invariants for loops
- Interpolation-based Model Checking (later in the lecture)
- (dynamic) symbolic execution

Predicate Abstraction and CounterExample Guided Abstraction-Refinement

Model Checking Software by Abstraction



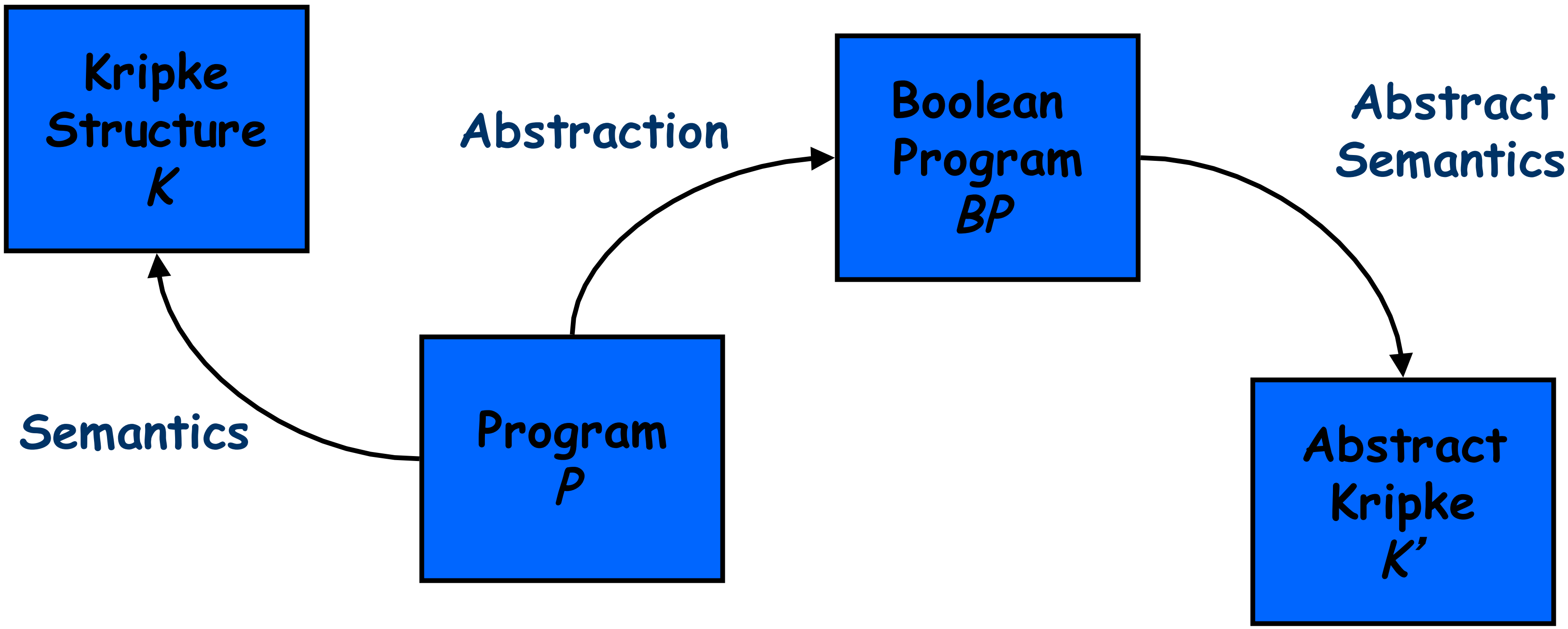
Programs are not finite state

- 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**

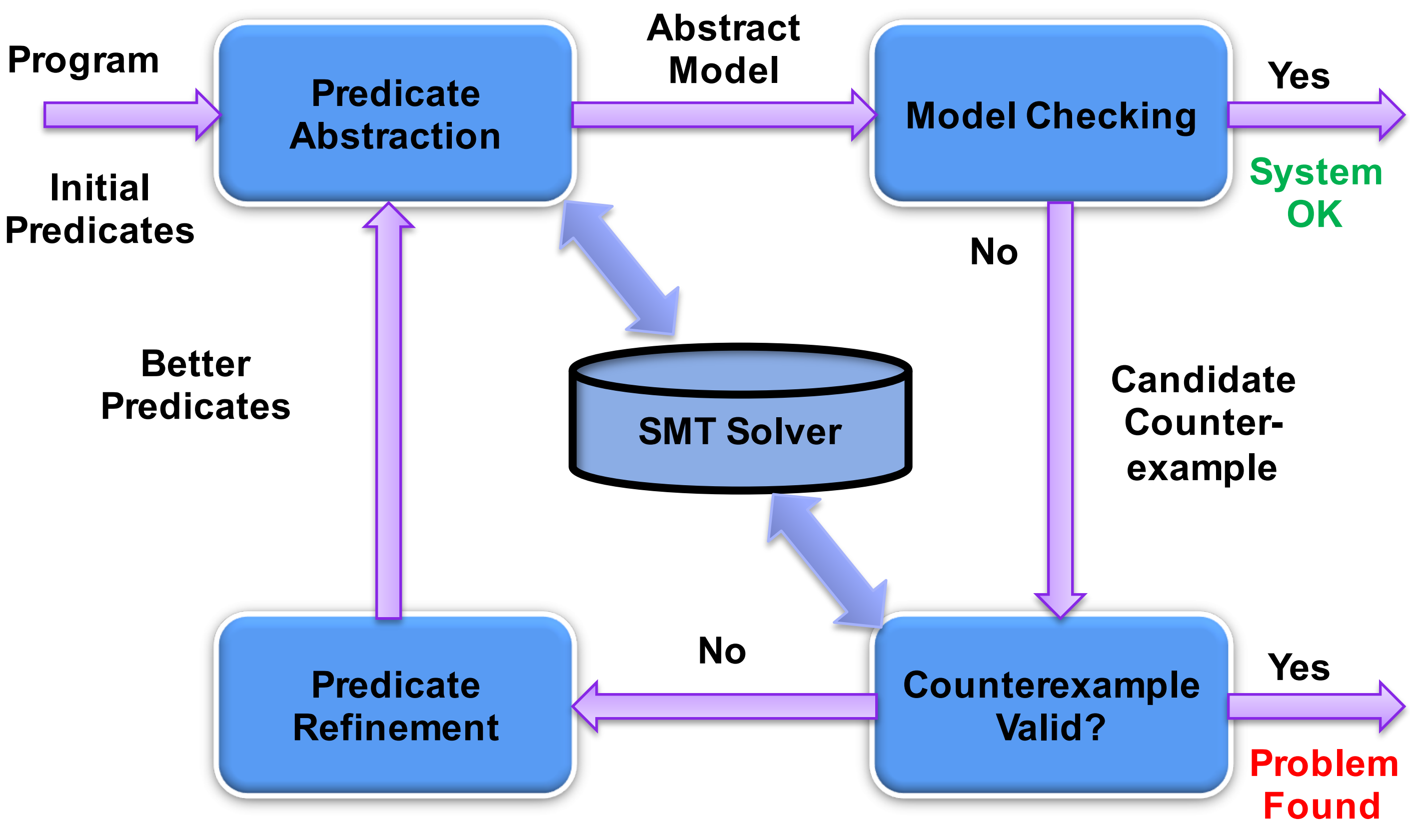
Software Model Checking and Abstraction



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 *
```

Parallel Assignment

$p_1 = ch(a_1, b_1), \quad p_2 = ch(a_2, b_2), \quad \ldots$

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

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').

Detour: Weakest Liberal Preconditions

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

{x>0} x = 2+y {y>0}              ✘

{*x>3 ∨ 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)"
}
```

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	✘

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?

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

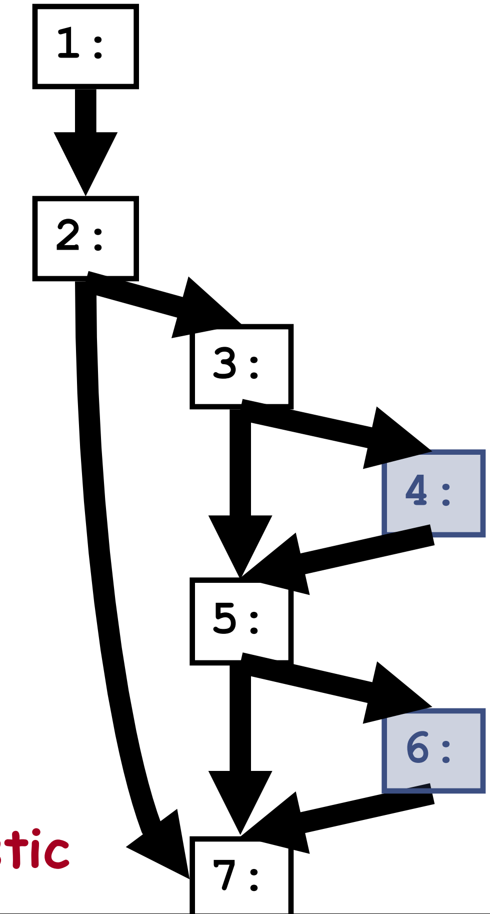
- Treat "unknown" as a special unknown value
- good for verification and refutation
- good for universal, existential, and mixed properties

BP Semantics: Over-Approximation

Abstraction

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

Over-Approximation



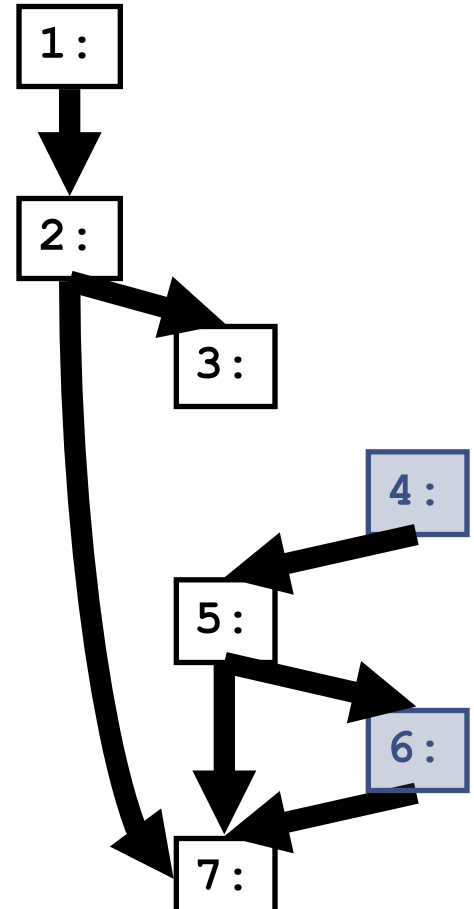
Unknown is treated as non-deterministic

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

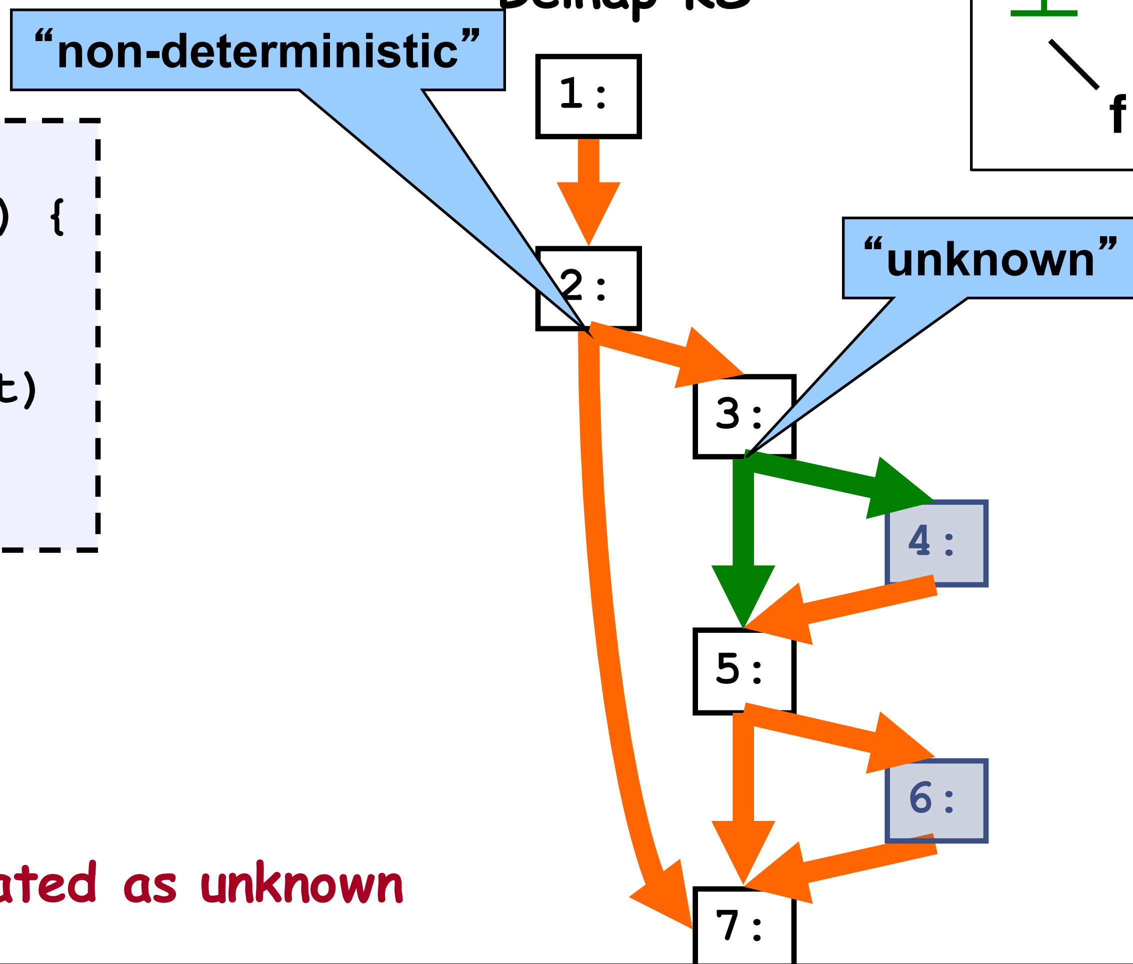
BP Semantics: Exact Approximation

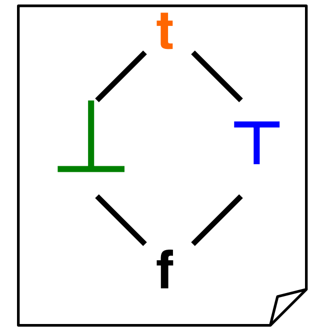
Abstraction

```
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

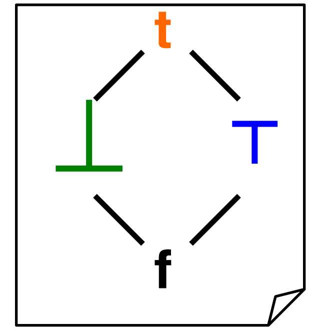
Concrete

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

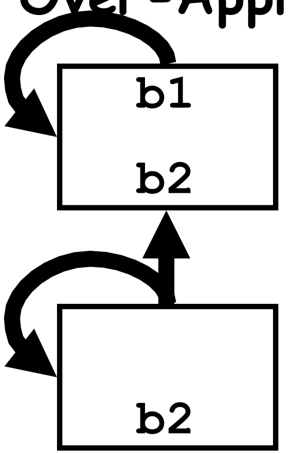
Abstract

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

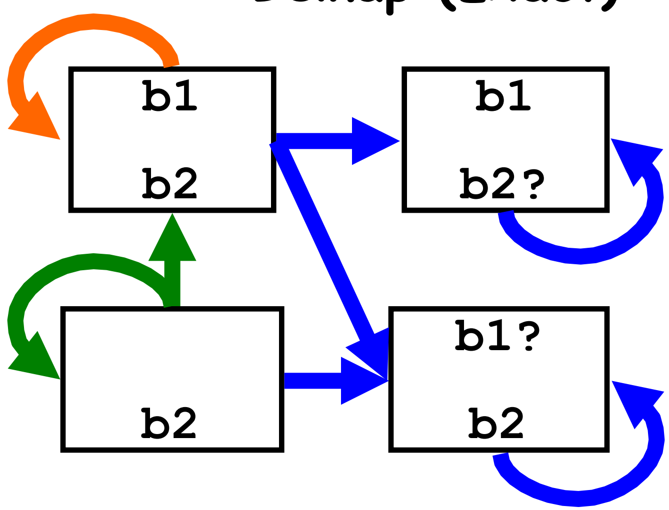
```
b1 = ch(b1,f);
b2 = ch(b2,¬b2)
```



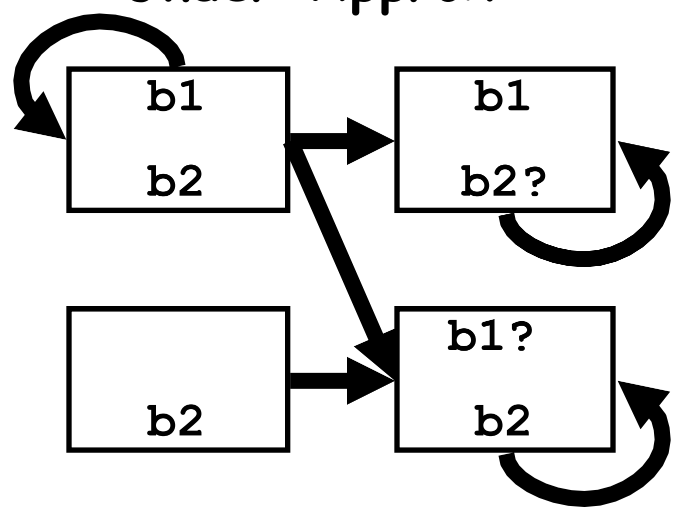
Over-Approx



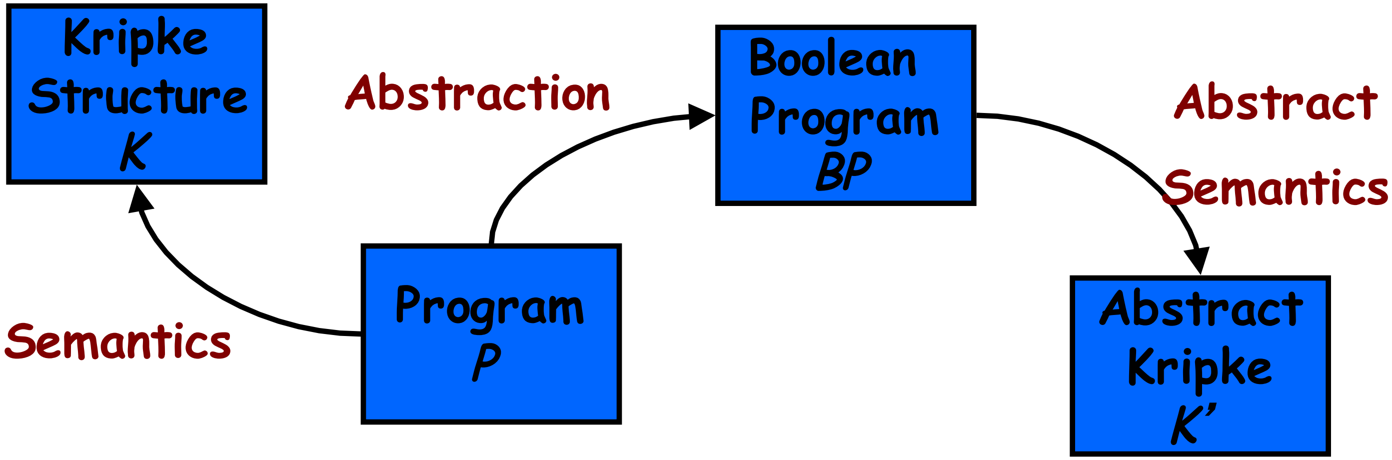
Belnap (Exact)



Under-Approx



Summary: Program Abstraction



Abstract a program P by a Boolean program BP

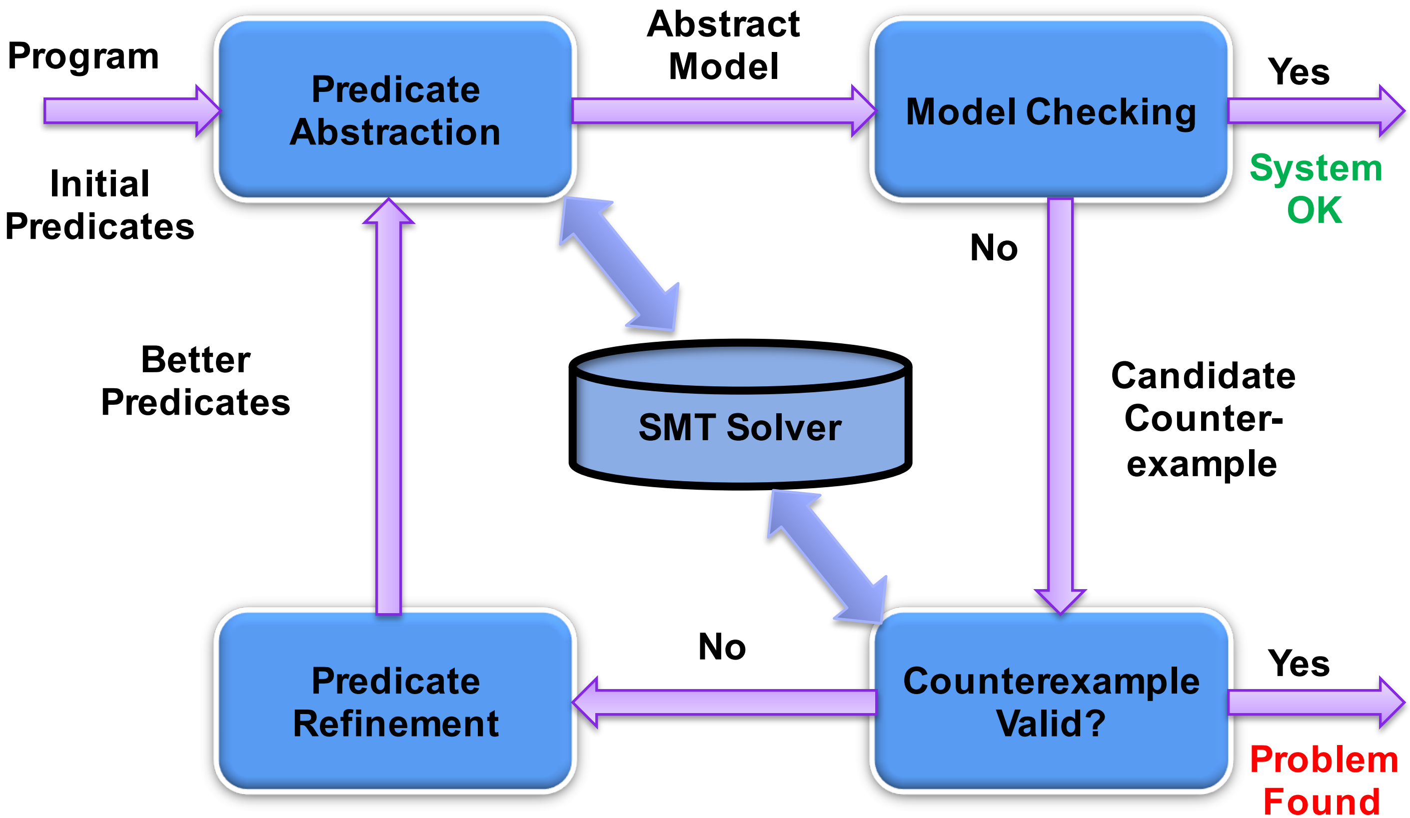
Pick an abstract semantics for this BP:

- 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

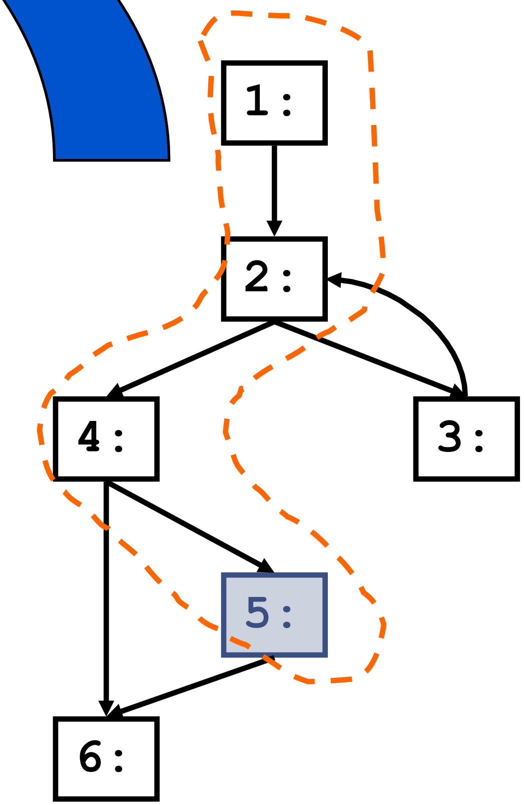
Example: Is ERROR Unreachable?

Program | **Abstraction** | **Over-Approximation**

Need This!

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

```
1: ;

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



CEGAR steps

Abstract ⟹ Translate ⟹ Check ⟹ Validate ⟹ Repeat

Example: Is ERROR Unreachable?

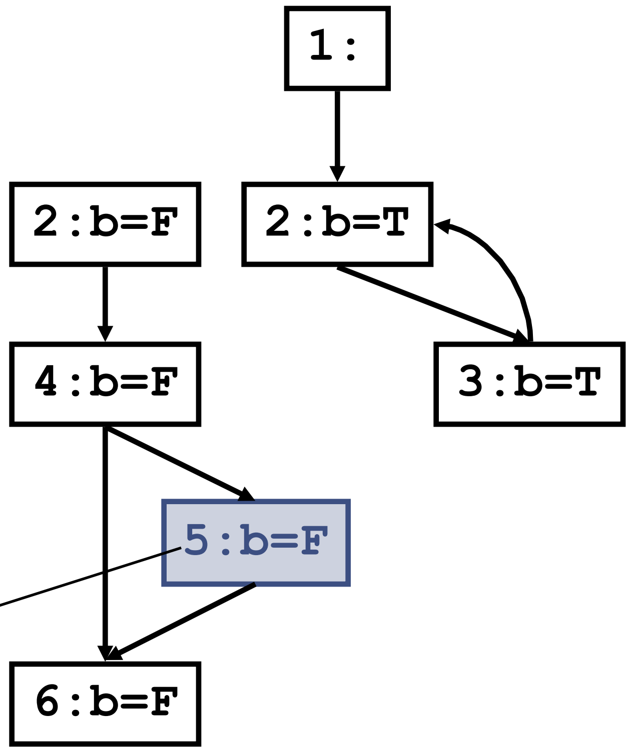
Program

```
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

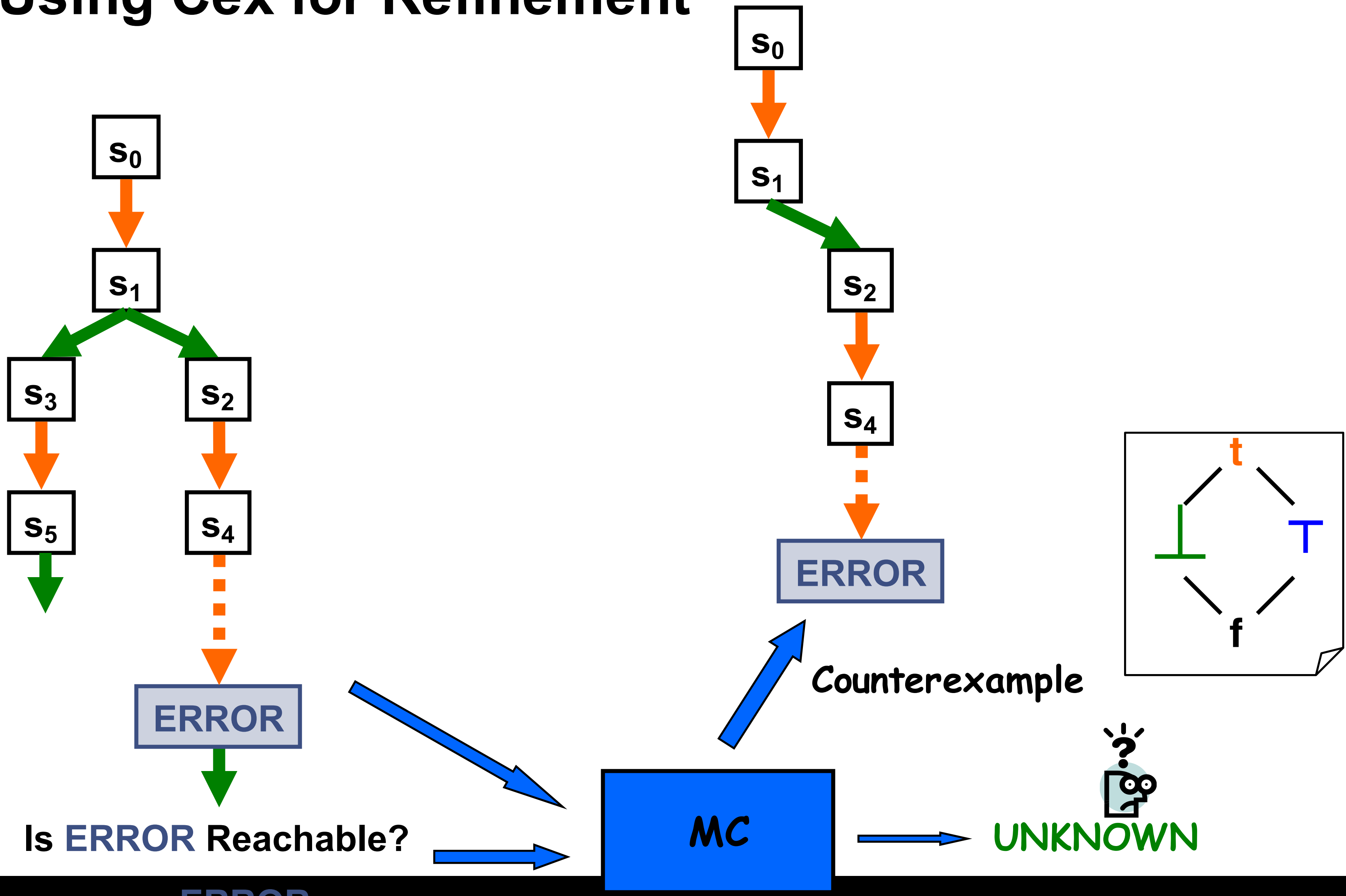


UNREACHABLE

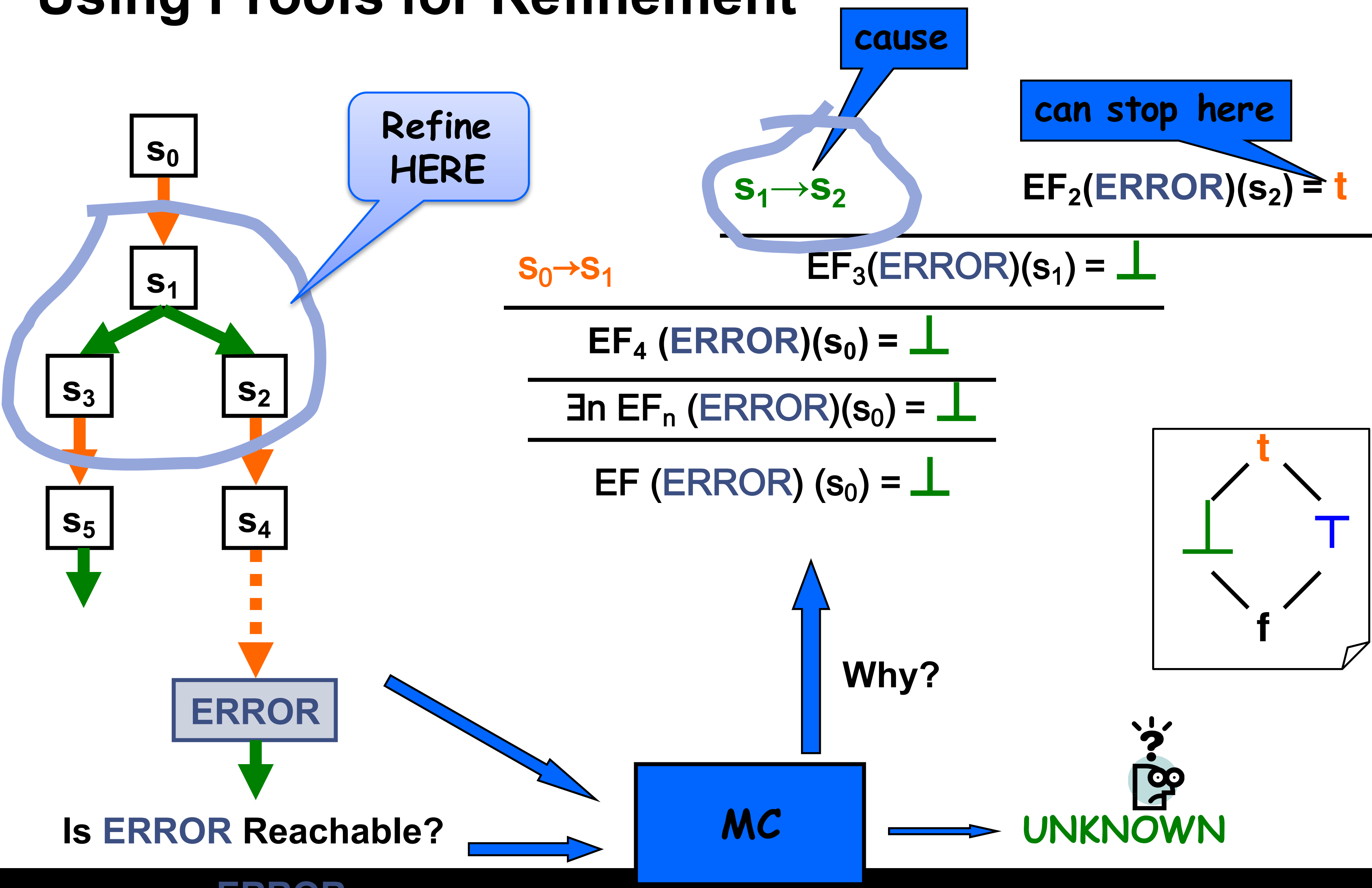
CEGAR steps

Abstract ⟹ Translate ⟹ Check ⟹ NO ERROR

Using Cex for Refinement



Using Proofs for Refinement



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

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)

An Example

$s_1 \longrightarrow s_2$ 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
```

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

Interpolation-based Model Checking

Programs, Safety, Cexs, Invariants

A transition system *P = (V, Init, $\mathcal{T}r$, Bad)*

P is UNSAFE if and only if there exists a number *N* s.t.

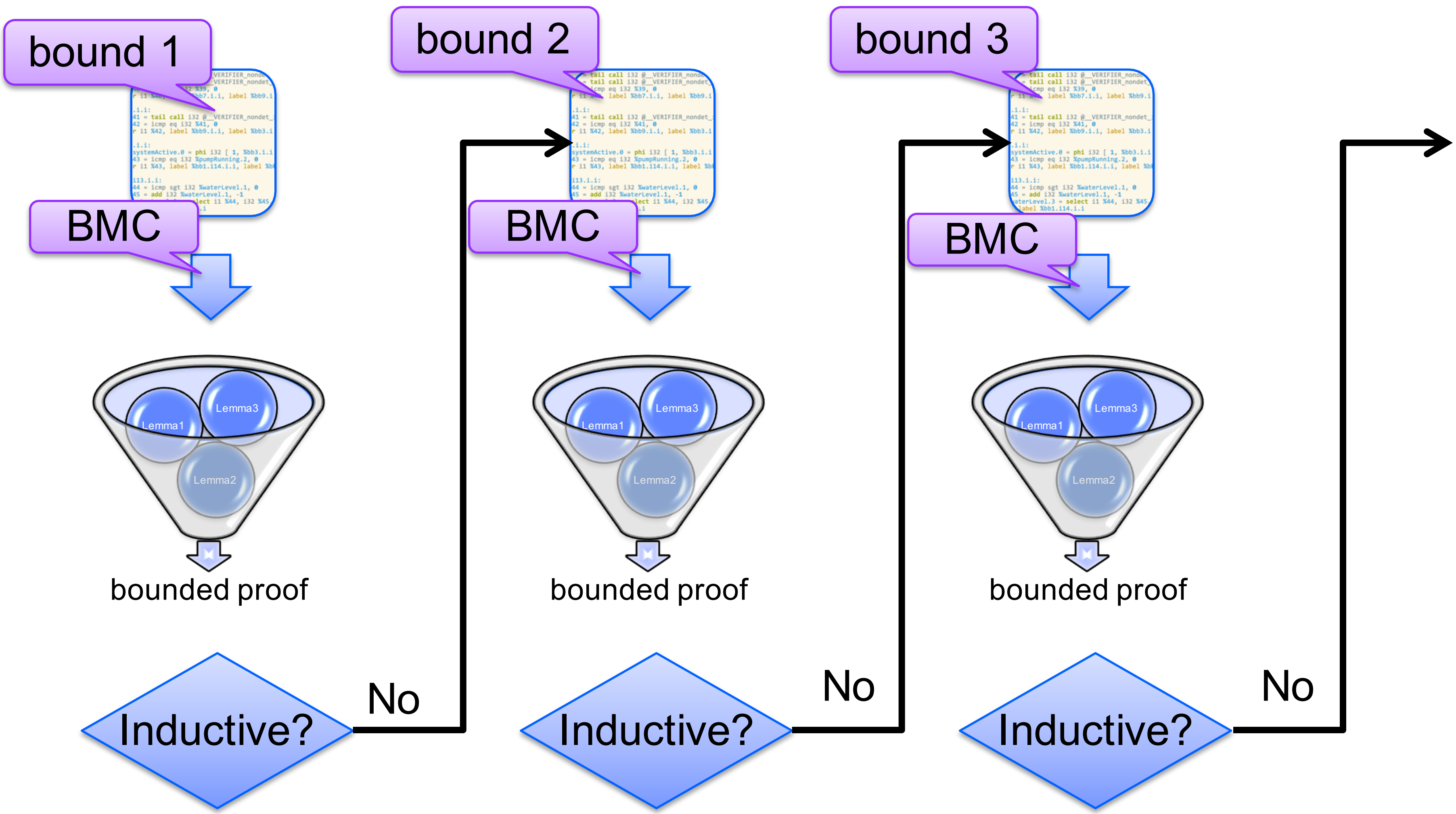
$$Init(X_0) \land \left(\bigwedge_{i=0}^{N-1} Tr(X_i, X_{i+1}) \right) \land Bad(X_N) \not\Rightarrow \bot$$

P is SAFE if and only if there exists a *safe inductive invariant Inv s.t.*

$$\left.\begin{array}{r} Init \Rightarrow Inv \\ Inv(X) \land Tr(X, X') \Rightarrow Inv(X') \end{array}\right\} \quad \text{Inductive}$$

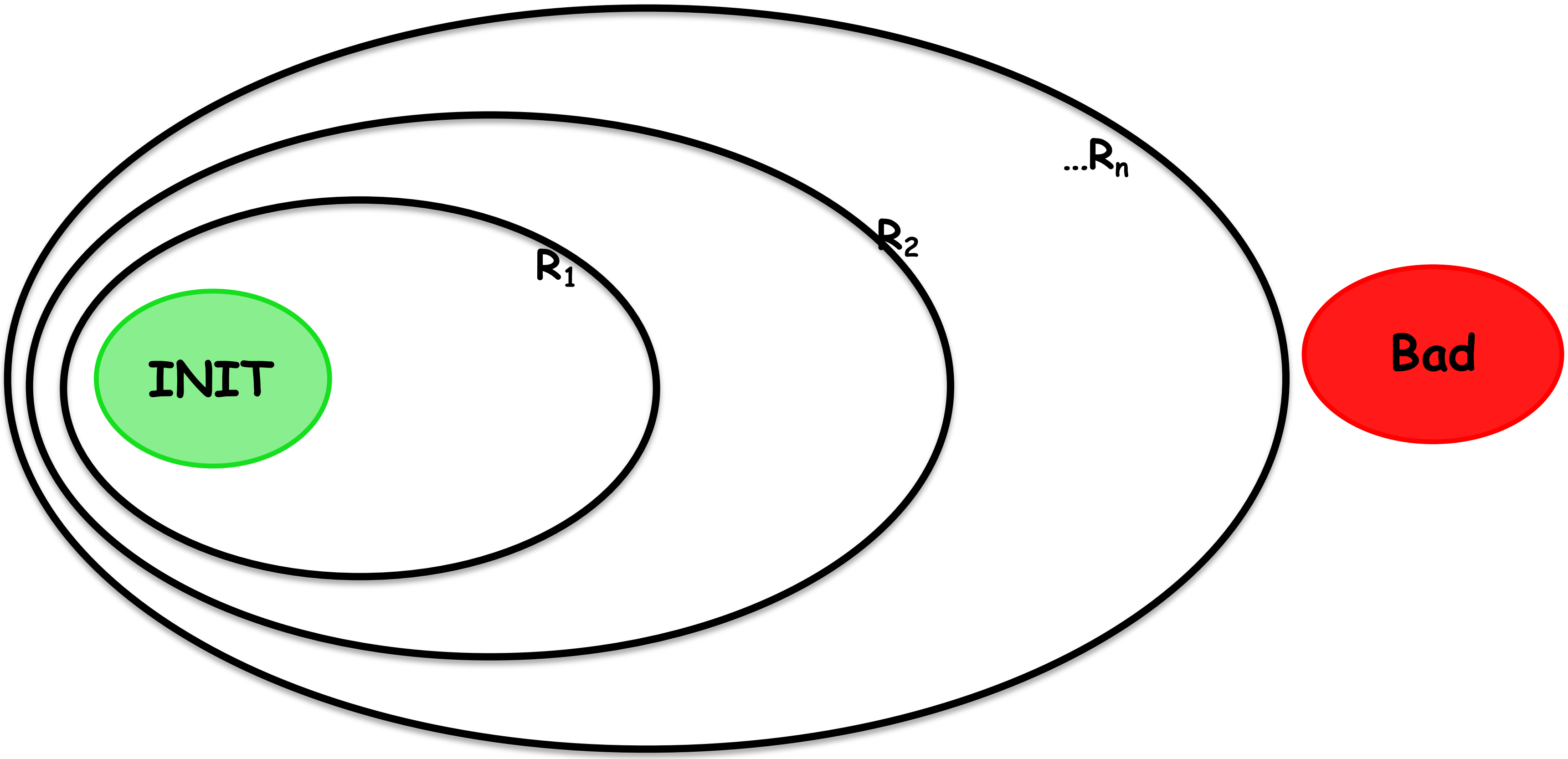
$$Inv \Rightarrow \neg Bad \quad \text{Safe}$$

Verification by Successive Under-Approximation



Reachability Analysis

Is Bad reachable?



Interpolating Model Checking

Key Idea

- turn SAT/SMT proofs of bounded safety to inductive traces
- repeat forever until a counterexample or inductive invariant are found

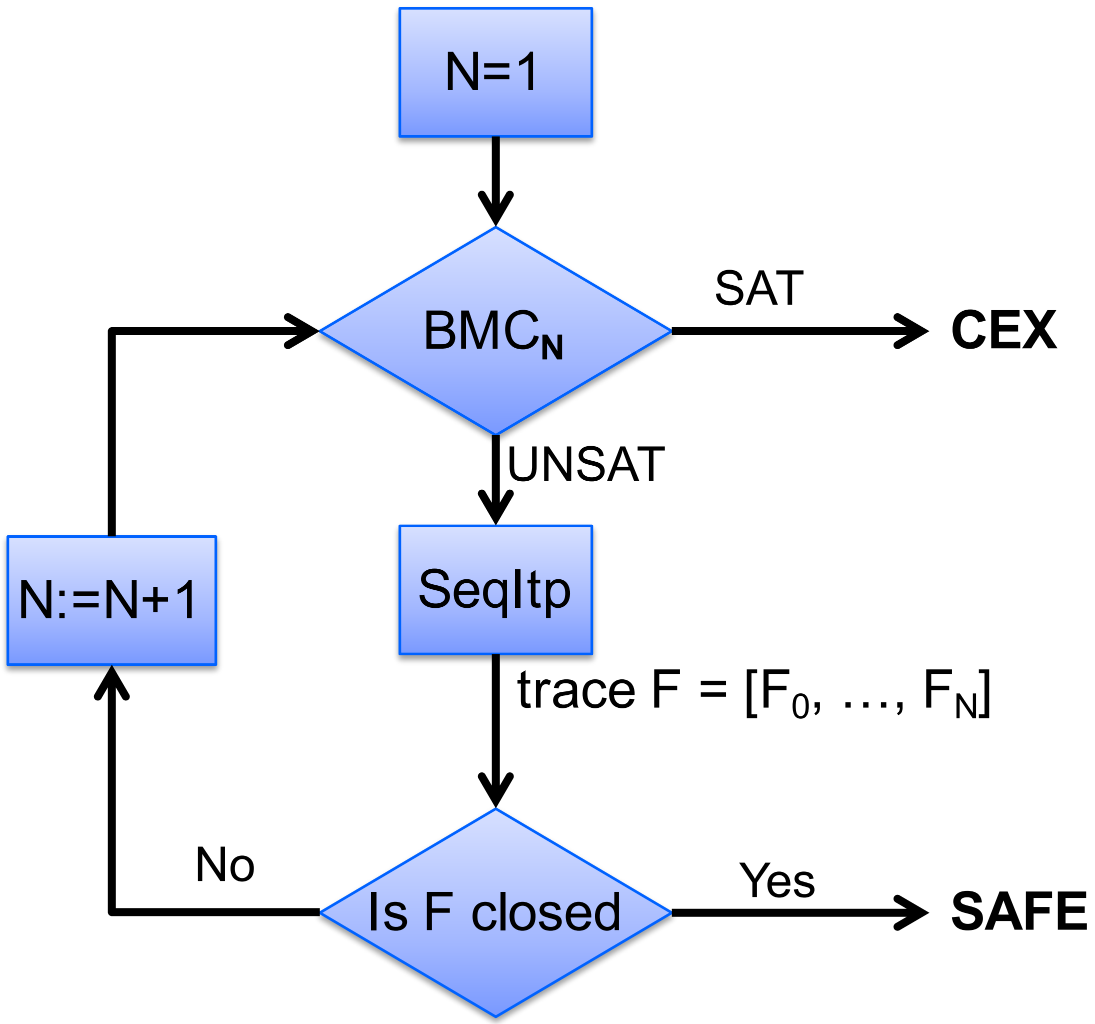
Introduced by McMillan in 2003

- Kenneth L. McMillan: Interpolation and SAT-Based Model Checking. CAV2003: 1-13
- based on pairwise Craig interpolation

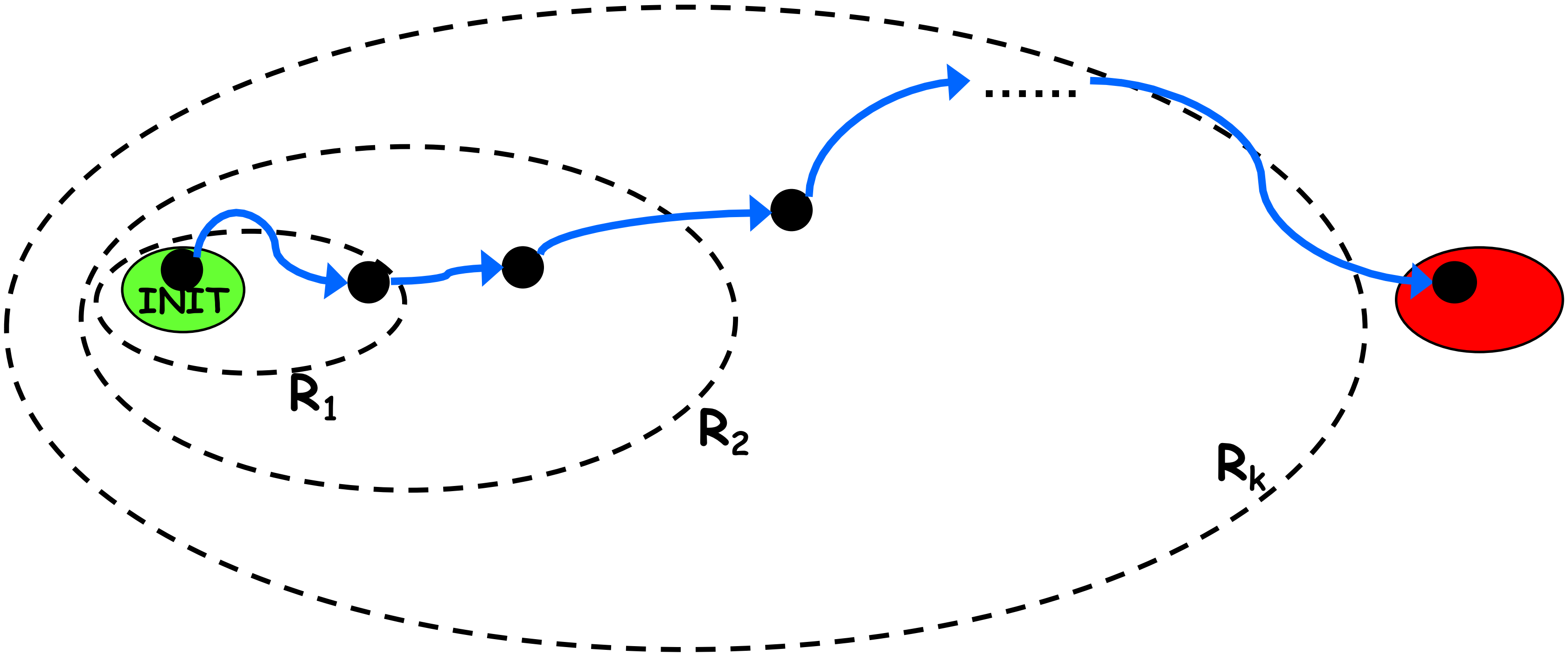
Extended to sequences and DAGs

- Yakir Vizel, Orna Grumberg: Interpolation-sequence based model checking. FMCAD 2009: 1-8
  – uses interpolation sequence
- Kenneth L. McMillan: Lazy Abstraction with Interpolants. CAV 2006: 123-136
  – IMPACT: interpolation sequence on each program path
- Aws Albarghouthi, Arie Gurfinkel, Marsha Chechik: From Under-Approximations to Over-Approximations and Back. TACAS 2012: 157-172
  – UFO: interpolation sequence on the DAG of program paths

IMC: Interpolating Model Checking



Bounded Model Checking



$$INIT(V^0) \wedge Tr(V^0,V^1) \wedge \ldots \wedge Tr(V^{k-1},V^k) \wedge Bad(V^k)$$

Inductive Trace

An *inductive trace* of a transition system $P = (V, Init, Tr, Bad)$ is a sequence of formulas $[F_0, \ldots, F_N]$ such that

- $Init \rightarrow F_0$
- $\forall\ 0 \leq i < N$, $F_i(v) \wedge Tr\ (v, u) \rightarrow F_{i+1}\ (u)$, or, in Hoare Logic $\{F_i\}\ Tr\ \{F_{i+1}\}$

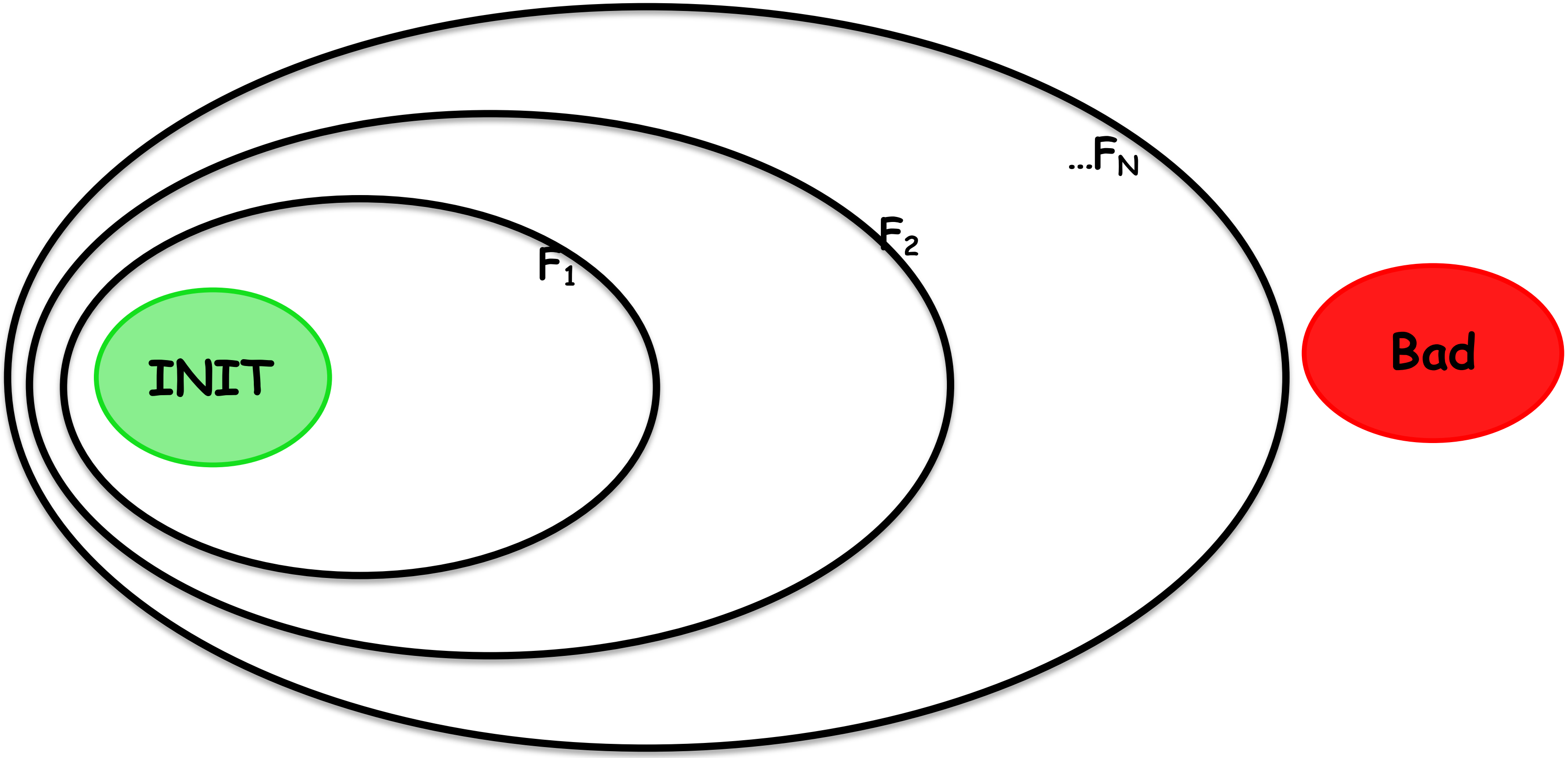
A trace is *safe* iff $\forall\ 0 \leq i \leq N$, $F_i \rightarrow \neg Bad$

A trace is *monotone* iff $\forall\ 0 \leq i < N$, $F_i \rightarrow F_{i+1}$

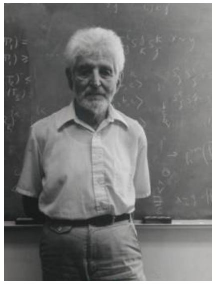
A trace is *closed* iff $\exists\ 1 \leq i \leq N$, $F_i \rightarrow (F_0 \vee \ldots \vee F_{i-1})$

A transition system P is SAFE iff it admits a safe closed trace

Inductive Trace in Pictures



Craig Interpolation Theorem



Theorem (Craig 1957)

Let A and B be two First Order (FO) formulae such that $A \Rightarrow \neg B$, then there exists a FO formula I, denoted ITP(A, B), such that

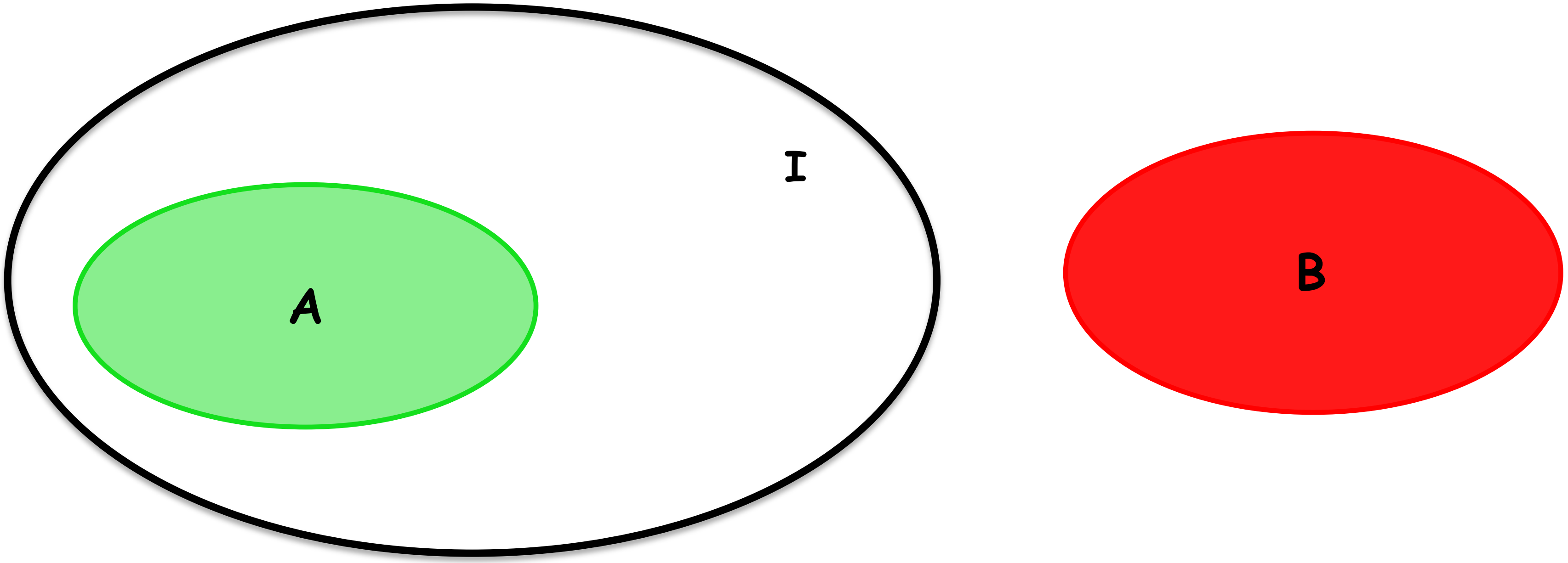
$$A \Rightarrow I \qquad I \Rightarrow \neg B$$

$$\mathit{atoms}(I) \in \mathit{atoms}(A) \cap \mathit{atoms}(B)$$

A Craig interpolant ITP(A, B) can be effectively constructed from a resolution proof of unsatisfiability of $A \wedge B$

In Model Cheching, Craig Interpolation Theorem is used to safely over-approximate the set of (finitely) reachable states

Craig Interpolant



Craig Interpolant Examples

Boolean logic

- A is {!b, (!a || b || c), a}   B is !a || !c
- Itp is a && c

EUF (equality with uninterpreted functions)

- A is {f(a) = b, p(f(a))}   B is {b=c, !p(c)}
- Itp is p(b)

Linear Arithmetic

- A is {z+x+y > 10, z < 5}   B is {x < -5, y < -3}
- Itp is x+y>5

Craig Interpolant as a Circuit

Let $F = A(x, z) \wedge B(z, y)$ be UNSAT, where x and y are distinct

- Note that for any assignment v to z either
  - $A(x, v)$ is UNSAT, or
  - $B(v, y)$ is UNSAT

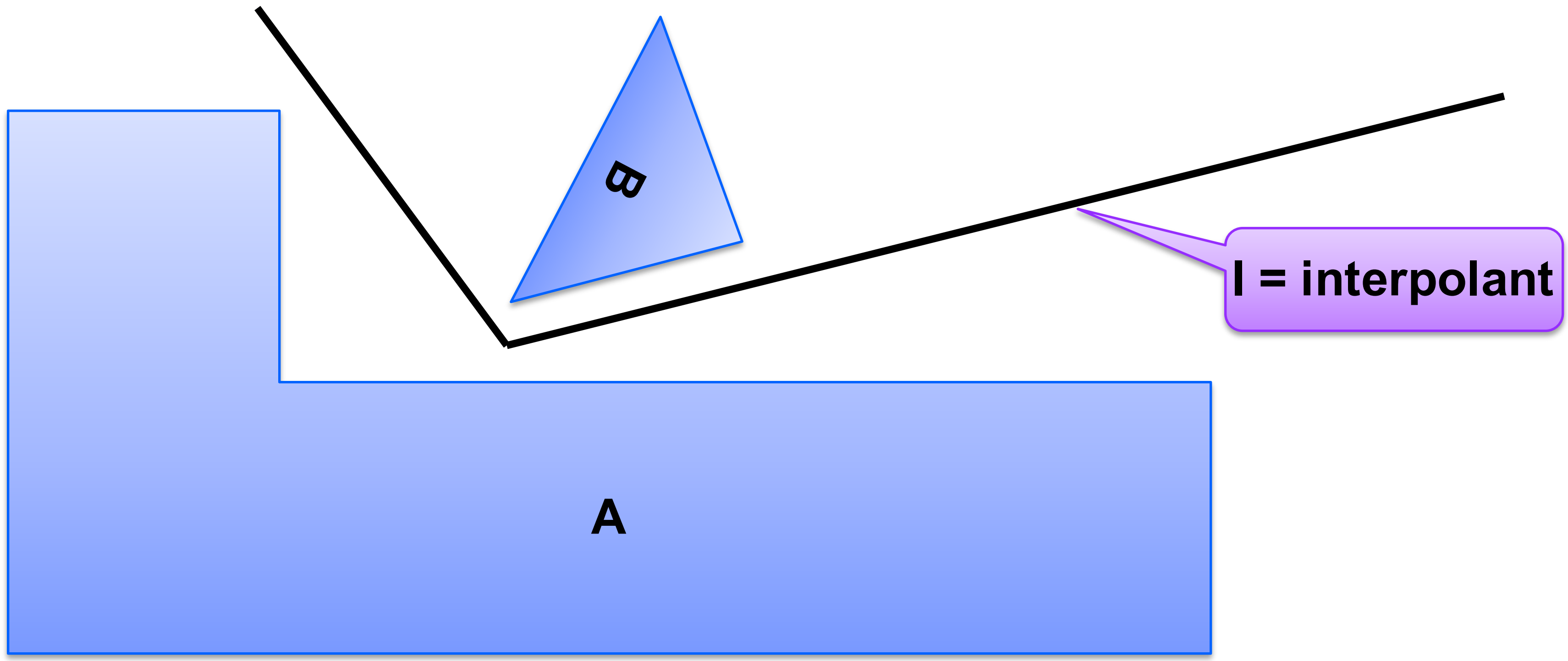
An interpolant is a circuit $I(z)$ such that for every assignment v to z

- $I(v) = A$ only if $A(x, v)$ is UNSAT
- $I(v) = B$ only if $B(v, y)$ is UNSAT

A proof system S has a *feasible interpolation* if for every refutation π of F in S, F has an interpolant polynomial in the size of π

- propositional resolution has feasible interpolation
- extended resolution does not have feasible interpolation

Craig Interpolation for Linear Arithmetic



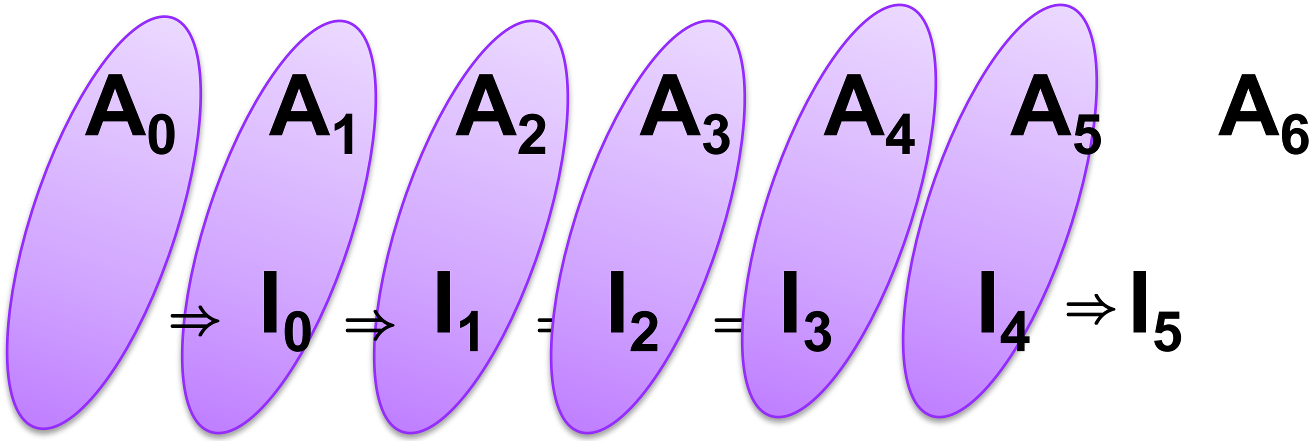
Useful properties of existing interpolation algorithms [CGS10] [HB12]

- $I \in ITP(A, B)$ then $\neg I \in ITP(B, A)$
- if A is syntactically convex (a monomial), then I is convex
- if B is syntactically convex, then I is co-convex (a clause)
- if A and B are syntactically convex, then I is a half-space

Interpolation Sequence

Given a sequence of formulas $\mathbf{A = \{A_i\}_{i=0}^{n}}$, an ***interpolation sequence*** $\mathbf{ItpSeq(A) = \{I_1, \ldots, I_{n-1}\}}$ is a sequence of formulas such that

- I_k is an **ITP** $(A_0 \wedge \ldots \wedge A_{k-1},\quad A_k \wedge \ldots \wedge A_n)$, and
- $\forall\, k<n \,.\, I_k \wedge A_{k+1} \Rightarrow I_{k+1}$



Can compute by pairwise interpolation applied to different cuts of a fixed resolution proof (very robust property of interpolation)

From Interpolants to Traces

A Sequence Interpolant of a BMC instance is an inductive trace

BMC$_N$

$$(\mathit{Init}(v_0))_0 \land (\mathit{Tr}(v_0,v_1))_1 \land \ldots \land (\mathit{Tr}(v_{N-1}, v_N))_N \land \mathit{Bad}(v_N)$$

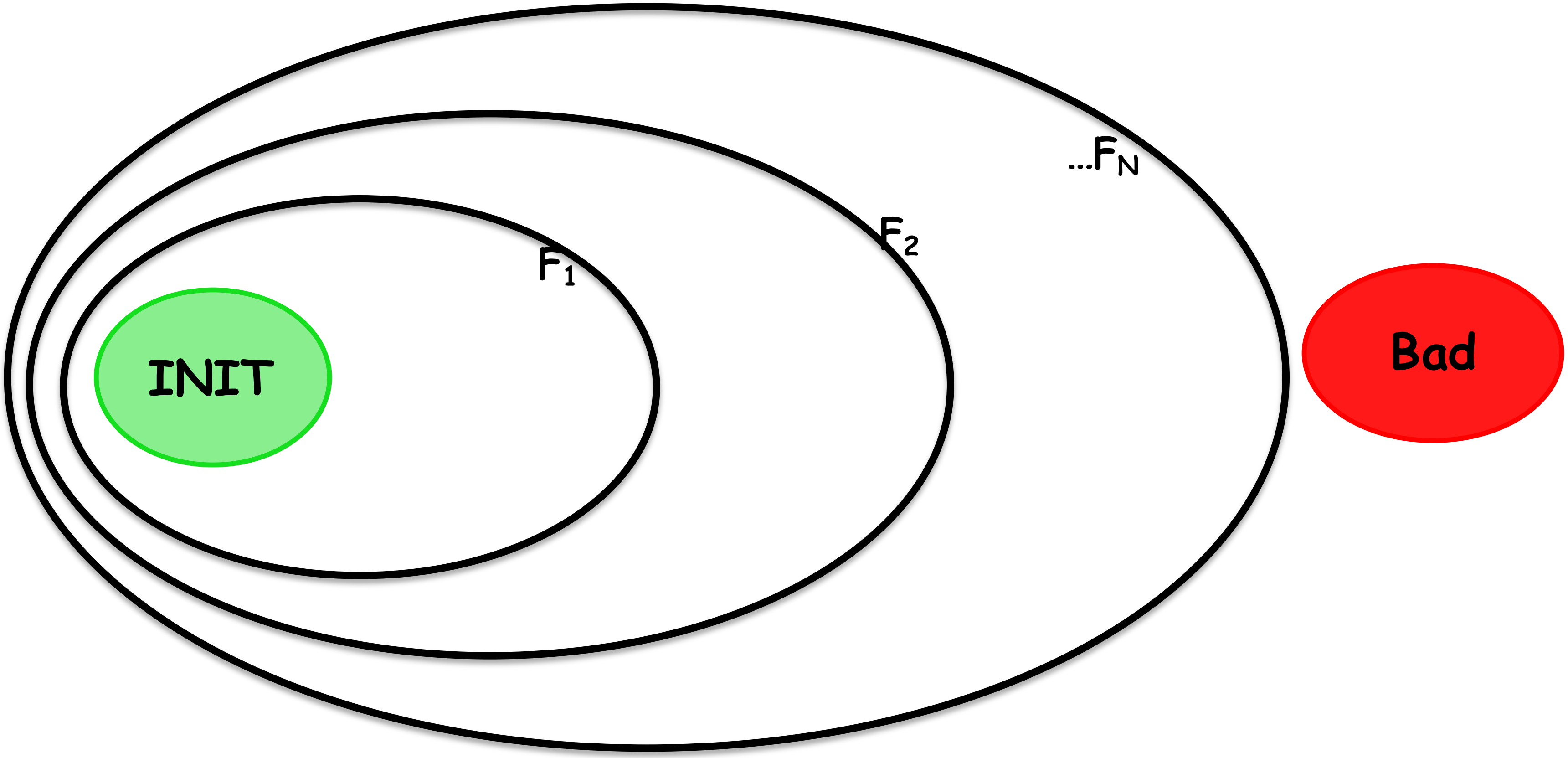
$$F_0(v_0) \qquad F_1(v_1) \qquad \qquad F_N(v_N)$$

trace

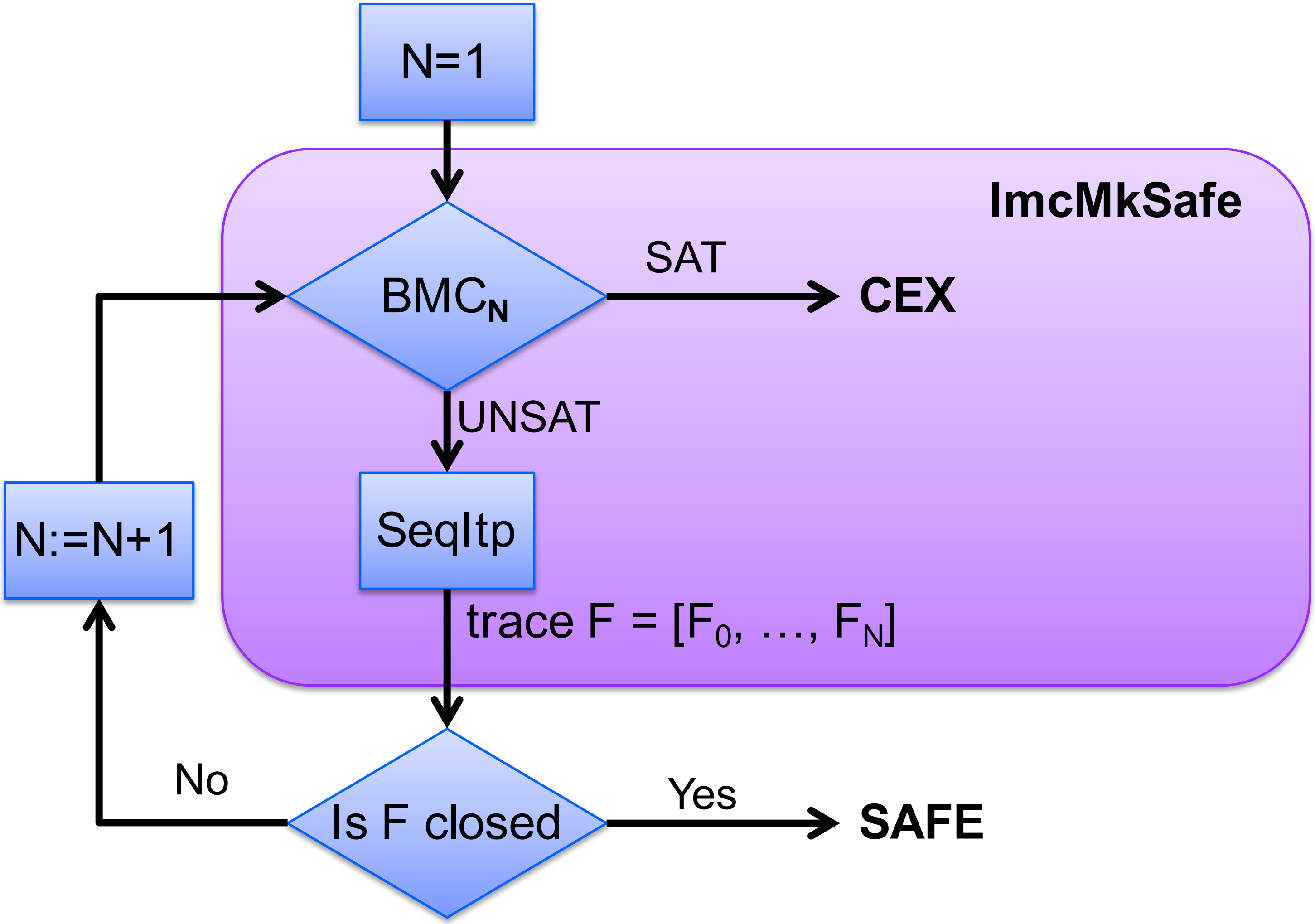
A trace computed by a sequence interpolant is

- safe
- NOT necessarily monotone
- NOT necessarily closed

Inductive Trace in Pictures



IMC: Interpolating Model Checking



IMC: Strength and Weaknesses

Strength

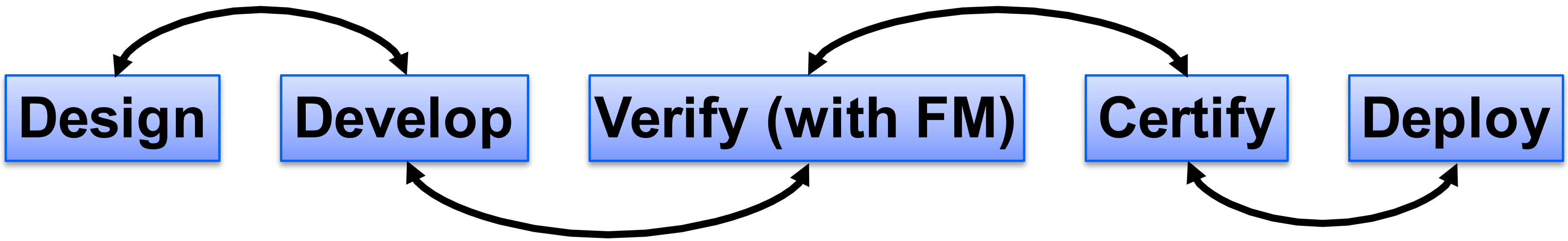
- elegant
- global bounded safety proof
- many different interpolation algorithms available
- easy to extend to SMT theories

Weaknesses

- the naïve version does not converge easily
  - interpolants are weaker towards the end of the sequence
- not incremental
  - no information is reused between BMC queries
- size of interpolants
- hard to guide

Trust in Formal Methods

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?

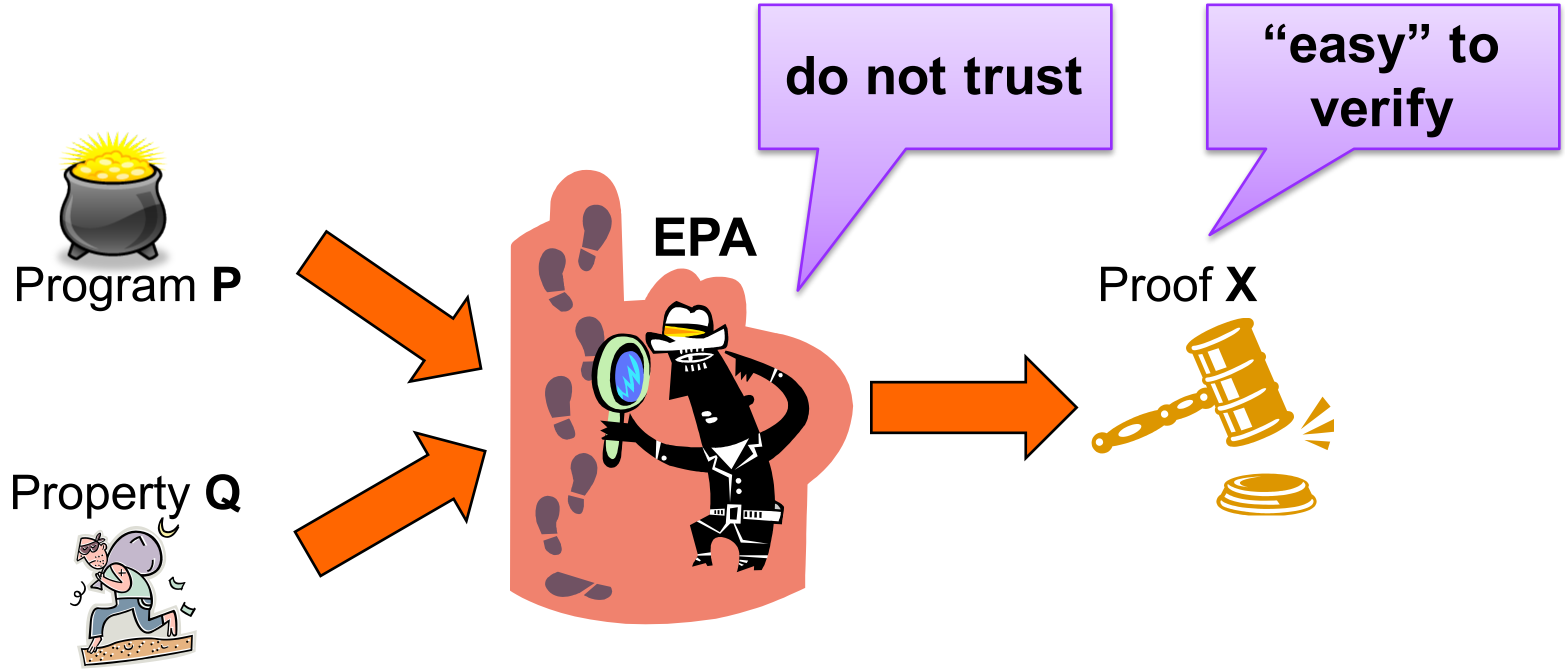
Verify the verifier

- (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

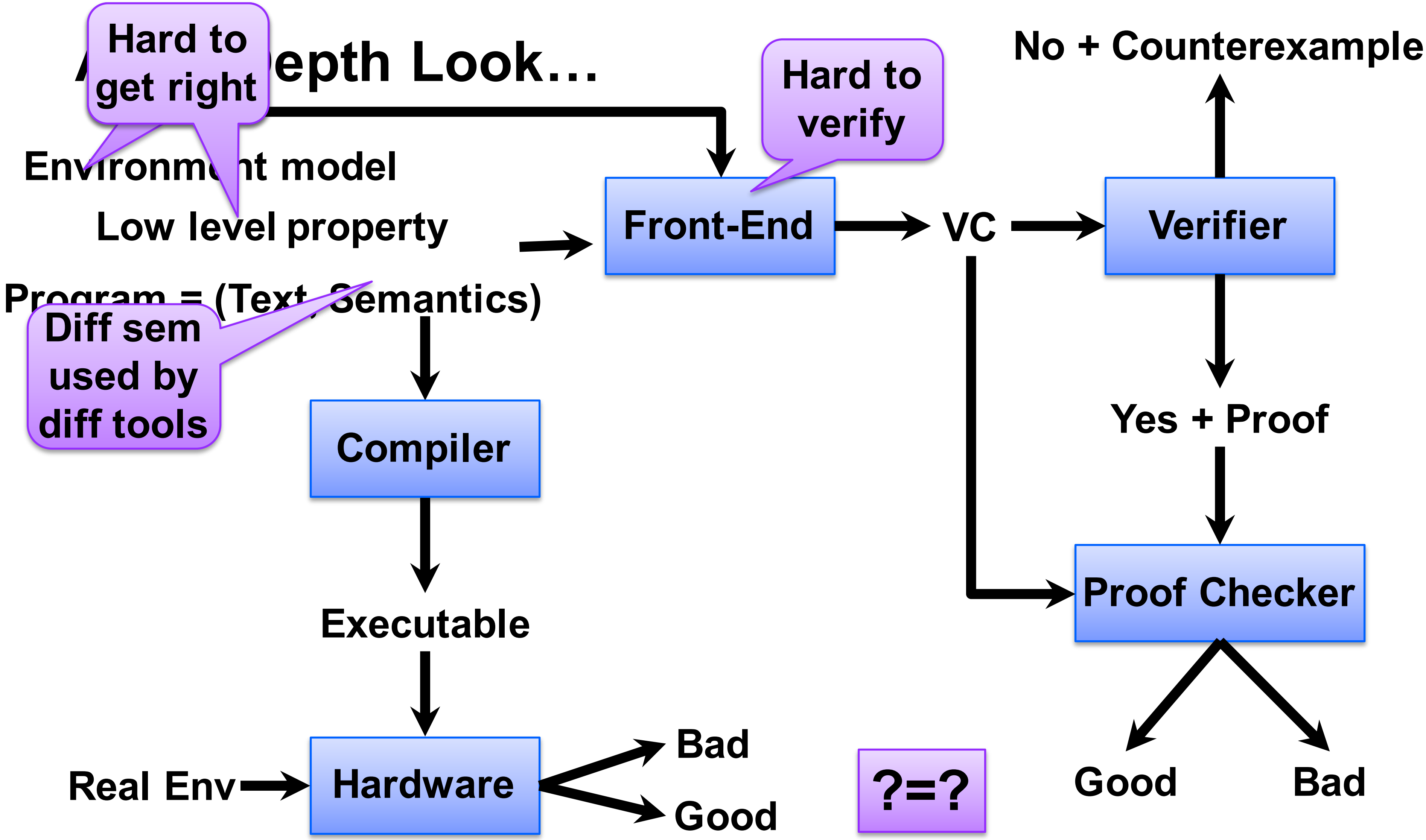


X witnesses that **P** satisfies **Q**. **X** can be objectively and independently verified.
Therefore, **EPA** is outside the Trusted Computing Base (**TCB**).

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 !!!



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

Verification Competitions

Multitude of events where solvers and analysis engines compete

SAT-RACE

- competitive event for SAT solvers
- http://baldur.iti.kit.edu/sat-race-2015/

SMT-COMP

- competitive event for SMT solvers
- http://www.smtcomp.org

SV-COMP

- Software Verification Competition
  - open to all, but most tools are based on Model Checking
- http://sv-comp.sosy-lab.org/2016/

CASC

- competitive event for Automated Theorem Proving
- http://www.cs.miami.edu/~tptp/CASC/

References

Software Model Checking and Program Analysis

- Vijay D'Silva, Daniel Kroening, Georg Weissenbacher: *A Survey of Automated Techniques for Formal Software Verification*. IEEE Trans. on CAD of Integrated Circuits and Systems 27(7): 1165-1178 (2008)
- Ranjit Jhala, Rupak Majumdar: *Software model checking*. ACM Comput. Surv. 41(4) (2009)

Symbolic Execution

- Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, Willem Visser: *Symbolic execution for software testing in practice: preliminary assessment*. ICSE 2011: 1066-1071

SMT and Decision Procedures

- Daniel Kroening, Ofer Strichman: *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series, Springer 2008, ISBN 978-3-540-74104-6, pp. 1-304
- *The SMT-LIB v2 Language and Tools: A Tutorial*, by David R. Cokk

Extra Slides

Hoare Triples

A Hoare triple {Pre} P {Post} is valid iff every terminating execution of *P* that starts in a state that satisfies *Pre* ends in a state that satisfies *Post*

Inductive Loop Invariant

$$\frac{Pre \Rightarrow Inv \qquad \{Inv \wedge C\}\ Body\ \{Inv\} \qquad Inv \wedge \neg C \Rightarrow Post}{\{Pre\}\ \textbf{while}\ C\ \textbf{do}\ Body\ \{Post\}}$$

Function Application

$$\frac{(Pre \wedge p=a) \Rightarrow P \qquad \{P\}\ Body_F\ \{Q\} \qquad (Q \wedge p,r=a,b) \Rightarrow Post}{\{Pre\}\ b = F(a)\ \{Post\}}$$

Recursion

$$\frac{\{Pre\}\ b = F(a)\ \{Post\} \vdash \{Pre\}\ Body_F\ \{Post\}}{\{Pre\}\ b = F(a)\ \{Post\}}$$

Weakest Liberal Pre-Condition

Validity of Hoare triples is reduced to FOL validity by applying a **predicate transformer**

Dijkstra's weakest liberal pre-condition calculus [Dijkstra'75]

wlp (P, Post)

weakest pre-condition ensuring that executing P ends in Post

{Pre} P {Post} is valid $\Leftrightarrow$ Pre $\Rightarrow$ **wlp** (P, Post)

A Simple Programming Language

```
Prog  ::= def Main(x) { body_M }, …, def P (x) { body_P }

body  ::= stmt (; stmt)*

stmt  ::= x = E | assert (E) | assume (E) |
          while E do S | y = P(E) |
          L:stmt | goto L              (optional)

E     := expression over program variables
```

Horn Clauses by Weakest Liberal Precondition

`Prog ::=` **`def`** `Main(x) {` $\texttt{body}_M$ `}, …,` **`def`** `P (x) {` $\texttt{body}_P$ `}`

wlp (`x=E`, Q) = **let** x=E **in** Q

wlp (**`assert`**`(E)` , Q) = E $\wedge$ Q

wlp (**`assume`**`(E)`, Q) = E $\rightarrow$ Q

wlp (**`while`** `E` **`do`** `S`, Q) = I(w) $\wedge$

$\forall w \,.\, ((I(w) \wedge E) \rightarrow \text{wlp}\,(S, I(w))) \wedge ((I(w) \wedge \neg E) \rightarrow Q))$

wlp (`y = P(E)`, Q) = $p_{pre}(E) \wedge (\forall\, r.\ p(E, r) \rightarrow Q[r/y])$

ToHorn (**`def`** `P(x) {S}`) = wlp (`x0=x;`**`assume`**`(`p_{pre}`(x))`; S, p(x0, ret))

ToHorn (Prog) = wlp (`Main()`, true) $\wedge$ $\forall\{P \in \text{Prog}\}$. ToHorn (`P`)

Example of a WLP Horn Encoding

{Pre: $y \geq 0$**}**

$x_o = x;$

$y_o = y;$

while $y > 0$ **do**

$\quad x = x+1;$

$\quad y = y-1;$

{Post: $x = x_o + y_o$**}**

ToHorn

C1: $I(x,y,x,y) \leftarrow y >= 0.$

C2: $I(x+1,y-1,x_o,y_o) \leftarrow I(x,y,x_o,y_o),\ y>0.$

C3: $false \leftarrow I(x,y,x_o,y_o),\ y \leq 0,\ x \neq x_o + y_o$

$\{y \geq 0\}\ P\ \{x = x_{old} + y_{old}\}$ is **true** iff the query C_3 is **satisfiable**

Single Static Assignment

SSA == every value has a unique assignment (a *definition*)

A procedure is in SSA form if every variable has exactly one definition

SSA form is used by many compilers

- explicit def-use chains
- simplifies optimizations and improves analyses

PHI-function are necessary to maintain unique definitions in branching control flow

x = PHI (v_0:bb_0, …, v_n:bb_n)) (phi-assignment)

"x gets v_i if previously executed block was bb_i"

Single Static Assignment: An Example

```
int x, y, n;

x = 0;
while (x < N) {
  if (y > 0)
    x = x + y;
  else
    x = x - y;
  y = -1 * y;
}
```

val:bb

```
0: goto 1
1: x_0 = PHI(0:0, x_3:5);
   y_0 = PHI(y:0, y_1:5);
   if (x_0 < N) goto 2 else goto 6

2: if (y_0 > 0) goto 3 else goto 4

3: x_1 = x_0 + y_0; goto 5

4: x_2 = x_0 - y_0; goto 5

5: x_3 = PHI(x_1:3, x_2:4);
   y_1 = -1 * y_0;
   goto 1
6:
```