

Constrained Horn Clauses (CHC)

Automated Program Verification (APV)
Fall 2018

Prof. Arie Gurfinkel



PREDICATE ABSTRACTION

Predicate Abstraction

Extends Boolean reasoning methods to non-Boolean domains

Given a set of predicates P , abstract transition relation by restricting its effects to the set P

- Each step of Tr sets some predicates in P to true and some to false
- Computing abstraction requires theory reasoning
- Abstract transition relation is Boolean, so Boolean methods can be applied

Predicate abstraction is an over-approximation

- May introduce spurious counterexamples that cannot be replayed in the real system

Abstraction-Refinement: replay counterexamples using theory reasoner

- Use BMC to replay
- Use Interpolation to learn new predicates

Implicit Predicate Abstraction with IC3

Idea: do not compute abstract transition relation upfront!

IC3 only requires computing one predecessor at a time

- Use theory reasoning to compute a predecessor
- Each POB/CTI/state is a Boolean valuations to all predicates

The rest is exactly like Boolean IC3

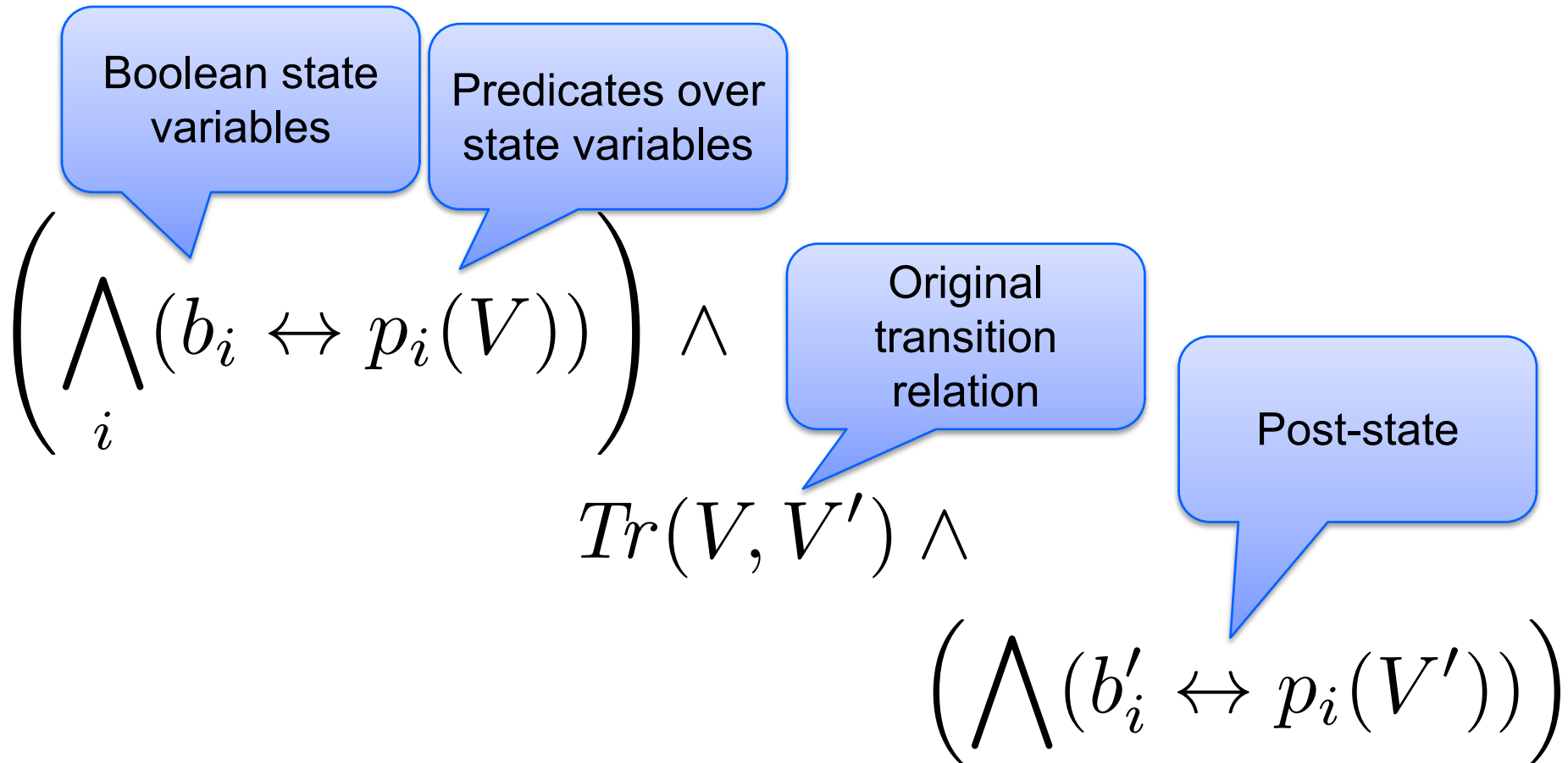
- Except that predecessor generalization does not work

To refine, replay the counterexamples using theory solver

- use interpolation to learn new predicates

Interesting idea to implement in Z3 using Spacer/CHC for refinement

Implicit Predicate Abstraction Construction



The diagram illustrates the formula for Implicit Predicate Abstraction Construction. It consists of three main parts connected by logical AND operators ($\wedge$). The first part is a large left parenthesis followed by a summation over index i of the expression $(b_i \leftrightarrow p_i(V))$, followed by a right parenthesis. A callout bubble labeled "Boolean state variables" points to b_i , and another bubble labeled "Predicates over state variables" points to $p_i(V)$. This is followed by an AND operator and the expression $Tr(V, V')$. A callout bubble labeled "Original transition relation" points to $Tr(V, V')$. This is followed by another AND operator and a large right parenthesis preceded by a summation over index i of the expression $(b'_i \leftrightarrow p_i(V'))$. A callout bubble labeled "Post-state" points to V' in the final expression.

$$\left(\bigwedge_i (b_i \leftrightarrow p_i(V)) \right) \wedge Tr(V, V') \wedge \left(\bigwedge_i (b'_i \leftrightarrow p_i(V')) \right)$$

There is a counter-example over b_i variables iff there are no lemmas over p_i predicates that can block the counter-example

Precise Logic-based Program Verification

Low-Level Bounded Model Checking (BMC)

- decide whether a low level program/circuit has an execution of a given length that violates a safety property
- effective decision procedure via encoding to propositional SAT

High-Level (Word-Level) Bounded Model Checking

- decide whether a program has an execution of a given length that violates a safety property
- efficient decision procedure via encoding to SMT

What is an SMT-like equivalent for Safety Verification?

- Logic: SMT-Constrained Horn Clauses
- Decision Procedure: Spacer / GPDR
 - extend IC3/PDR algorithms from Hardware Model Checking

CONSTRAINED HORN CLAUSES

Constrained Horn Clauses (CHCs)

A Constrained Horn Clause (CHC) is a FOL formula

$$\forall V \cdot (\varphi \wedge p_1[X_1] \wedge \cdots \wedge p_n[X_n]) \rightarrow h[X]$$

where

- $\mathcal{T}$ is a background theory (e.g., Linear Arithmetic, Arrays, Bit-Vectors, or combinations of the above)
- V are variables, and X_i are terms over V
- φ is a constraint in the background theory $\mathcal{T}$
- $p_1, \dots, p_n, h$ are n -ary predicates
- $p_i[X]$ is an application of a predicate to first-order terms

CHC Satisfiability

A $\mathcal{T}$ -**model** of a set of CHCs Π is an extension of the model M of $\mathcal{T}$ with a first-order interpretation of each predicate p_i that makes all clauses in Π true in M

A set of clauses is **satisfiable** if and only if it has a model

- This is the usual FOL satisfiability

A $\mathcal{T}$ -**solution** of a set of CHCs Π is a substitution σ from predicates p_i to $\mathcal{T}$ -formulas such that $\Pi\sigma$ is $\mathcal{T}$ -valid

In the context of program verification

- a program satisfies a property iff corresponding CHCs are satisfiable
- solutions are inductive invariants
- refutation proofs are counterexample traces

CHC Notation and Terminology

head

body

constraint

Rule

$$h[X] \leftarrow p_1[X_1], \dots, p_n[X_n], \phi$$

Query

$$\text{false} \leftarrow p_1[X_1], \dots, p_n[X_n], \phi.$$

Fact

$$h[X] \leftarrow \phi.$$

Linear CHC

$$h[X] \leftarrow p[X_1], \phi.$$

Non-Linear CHC

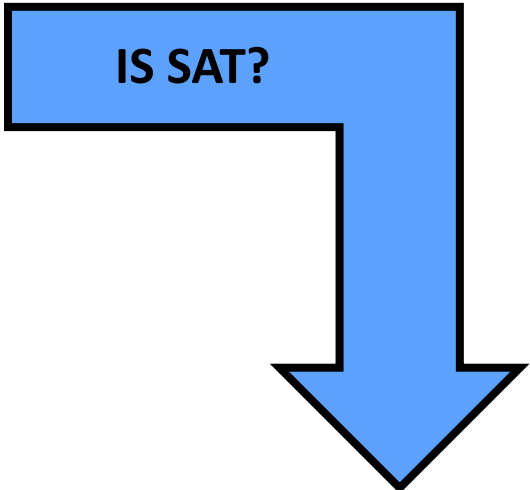
$$h[X] \leftarrow p_1[X_1], \dots, p_n[X_n], \phi.$$

for $n > 1$

Program Verification with HORN(LIA)

```
z = x; i = 0;  
assume (y > 0);  
while (i < y) {  
    z = z + 1;  
    i = i + 1;  
}  
assert(z == x + y);
```

IS SAT?



$z = x \ \& \ i = 0 \ \& \ y > 0$	$\rightarrow$	$\text{Inv}(x, y, z, i)$
$\text{Inv}(x, y, z, i) \ \& \ i < y \ \& \ z1=z+1 \ \& \ i1=i+1$	$\rightarrow$	$\text{Inv}(x, y, z1, i1)$
$\text{Inv}(x, y, z, i) \ \& \ i \geq y \ \& \ z \neq x+y$	$\rightarrow$	false

In SMT-LIB

```
(set-logic HORN)

;; Inv(x, y, z, i)
(declare-fun Inv ( Int Int Int Int) Bool)

(assert
  (forall ( (A Int) (B Int) (C Int) (D Int))
    (=> (and (> B 0) (= C A) (= D 0))
      (Inv A B C D)))
)
(assert
  (forall ( (A Int) (B Int) (C Int) (D Int) (C1 Int) (D1 Int) )
    (=>
      (and (Inv A B C D) (< D B) (= C1 (+ C 1)) (= D1 (+ D
1))))
    (Inv A B C1 D1)
  )
)
(assert
  (forall ( (A Int) (B Int) (C Int) (D Int))
    (=> (and (Inv A B C D) (>= D B) (not (= C (+ A B))))
      false
    )
  )
)

(check-sat)
(get-model)
```

```
$ z3 add-by-one.smt2
```

```
sat
(model
  (define-fun Inv ((x!0 Int) (x!1 Int) (x!2 Int) (x!3 Int)) Bool
    (and (<= (+ x!2 (* (- 1) x!0) (* (- 1) x!3)) 0)
      (<= (+ x!2 (* (- 1) x!0) (* (- 1) x!1)) 0)
      (<= (+ x!0 x!3 (* (- 1) x!2)) 0)))
  )
```

$\text{Inv}(x, y, z, i)$

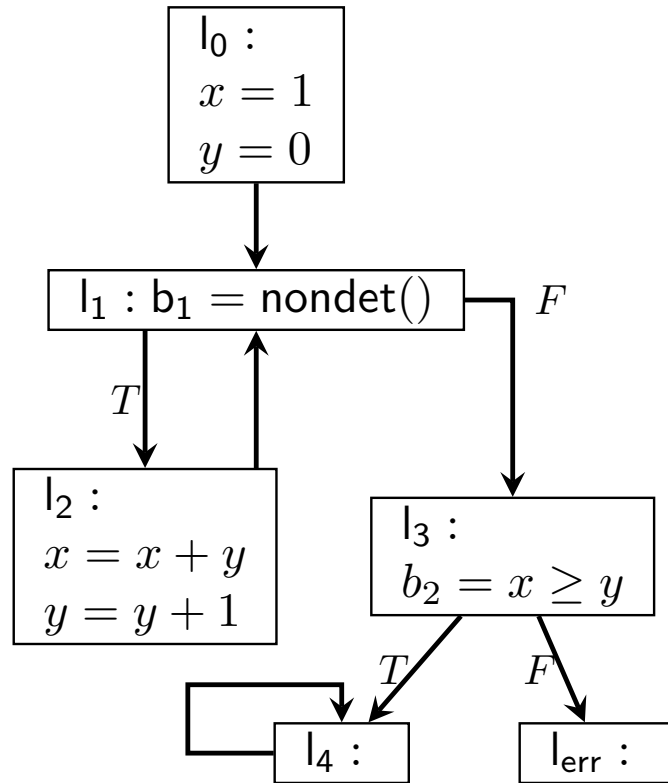
$z = x + i$

$z \leq x + y$

Programs, CFG, Horn Clauses

```

int x = 1;
int y = 0;
while (*) {
    x = x + y;
    y = y + 1;
}
assert(x ≥ y);
    
```



- ⟨1⟩ $p_0.$
- ⟨2⟩ $p_1(x, y) \leftarrow p_0, x = 1, y = 0.$
- ⟨3⟩ $p_2(x, y) \leftarrow p_1(x, y).$
- ⟨4⟩ $p_3(x, y) \leftarrow p_1(x, y).$
- ⟨5⟩ $p_1(x', y') \leftarrow p_2(x, y), x' = x + y, y' = y + 1.$
- ⟨6⟩ $p_4 \leftarrow (x \geq y), p_3(x, y).$
- ⟨7⟩ $p_{\text{err}} \leftarrow (x < y), p_3(x, y).$
- ⟨8⟩ $p_4 \leftarrow p_4.$
- ⟨9⟩ $\perp \leftarrow p_{\text{err}}.$

Horn Clauses for Program Verification

$e_{out}(x_0, w, e_o)$, which is an entry point into successor edges. with the edges are formulated as follows:

$$\begin{aligned} p_{init}(x_0, w, \perp) &\leftarrow x = x_0 && \text{where } x \text{ occurs in } w \\ p_{exit}(x_0, ret, \top) &\leftarrow \ell(x_0, w, \top) && \text{for each label } \ell, \text{ and re} \\ p(x, ret, \perp, \perp) &\leftarrow p_{exit}(x, ret, \perp) \\ p(x, ret, \perp, \top) &\leftarrow p_{exit}(x, ret, \top) \\ \ell_{out}(x_0, w', e_o) &\leftarrow \ell_{in}(x_0, w, e_i) \wedge \neg e_i \wedge \neg wlp(S, \neg(e_i = \end{aligned}$$

5. incorrect :- Z=W+1, W ≥ 0, W+1 < read(A, W, U), read(A, 2
6. p(I1, N, B) :- 1 ≤ I, I < N, D = I - 1, I1 = I + 1. V = U + 1. read(A, D, U), write(A
7. p(I, N, A) :- I = 1. N > 1.

De Angelis et al. Verifying Array Programs by Transforming Verification Conditions. VMCAI'14

Weakest Preconditions If we apply Boogie directly we obtain a translation from programs to Horn logic using a weakest liberal pre-condition calculus [26]:

$$\begin{aligned} \text{ToHorn}(\text{program}) &:= wlp(\text{Main}(), \top) \wedge \bigwedge_{\text{decl} \in \text{program}} \text{ToHorn}(\text{decl}) \\ \text{ToHorn}(\text{def } p(x) \{S\}) &:= wlp \left(\begin{array}{l} \text{havoc } x_0; \text{ assume } x_0 = x; \\ \text{assume } p_{pre}(x); S, \end{array} p(x_0, ret) \right) \\ wlp(x := E, Q) &:= \text{let } x = E \text{ in } Q \\ wlp(\text{if } E \text{ then } S_1 \text{ else } S_2, Q) &:= wlp(((\text{assume } E; S_1) \square (\text{assume } \neg E; S_2)), Q) \\ wlp((S_1 \square S_2), Q) &:= wlp(S_1, Q) \wedge wlp(S_2, Q) \\ wlp(S_1; S_2, Q) &:= wlp(S_1, wlp(S_2, Q)) \\ wlp(\text{havoc } x, Q) &:= \forall x. Q \\ wlp(\text{assert } \varphi, Q) &:= \varphi \wedge Q \\ wlp(\text{assume } \varphi, Q) &:= \varphi \rightarrow Q \\ wlp(\text{while } E \text{ do } S, Q) &:= \text{inv}(w) \wedge \\ &\quad \forall w. \left(\begin{array}{l} ((\text{inv}(w) \wedge E) \rightarrow wlp(S, \text{inv}(w))) \\ \wedge ((\text{inv}(w) \wedge \neg E) \rightarrow Q) \end{array} \right) \end{aligned}$$

To translate a procedure call $\ell : y := q(E); \ell'$ within a procedure p , create the clauses:

$$\begin{aligned} p(w_0, w_4) &\leftarrow p(w_0, w_1), \text{call}(w_1, w_2), q(w_2, w_3), \text{return}(w_1, w_3, w_4) \\ q(w_2, w_2) &\leftarrow p(w_0, w_1), \text{call}(w_1, w_2) \\ \text{call}(w, w') &\leftarrow \pi = \ell, x' = E, \pi' = \ell_{q_{init}} \\ \text{return}(w, w', w'') &\leftarrow \pi' = \ell_{q_{exit}}, w'' = w[\text{ret}'/y, \ell'/\pi] \end{aligned}$$

Bjørner, Gurfinkel, McMillan, and Rybalchenko:
Horn Clause Solvers for Program Verification

Horn Clauses for Concurrent / Distributed / Parameterized Systems

For assertions $R_1, \dots, R_N$ over V and $E_1, \dots, E_N$ over V, V' ,

- CM1 : $init(V) \rightarrow R_i(V)$
 CM2 : $R_i(V) \wedge \rho_i(V, V') \rightarrow R_i(V')$
 CM3 : $(\bigvee_{i \in 1..N \setminus \{j\}} R_i(V) \wedge \rho_i(V, V')) \rightarrow E_j(V, V')$
 CM4 : $R_i(V) \wedge E_i(V, V') \wedge \rho_i^{\bar{}}(V, V') \rightarrow R_i(V')$
 CM5 : $R_1(V) \wedge \dots \wedge R_N(V) \wedge error(V) \rightarrow false$

multi-threaded program P is safe

Rybalchenko et al. Synthesizing Software Verifiers from Proof Rules. PLDI'12

- (initial) $init(g, x_1) \wedge \dots \wedge init(g, x_n) \rightarrow Inv(g, \ell_{init}, x_1, \dots, \ell_{init}, x_k)$
 (inductive) $Inv(g, \ell_1, x_1, \dots, \ell_i, x_i, \dots, \ell_k, x_k) \wedge s(g, x_i, g', x'_i) \rightarrow Inv(g', \ell_1, x_1, \dots, \ell'_i, x'_i, \dots, \ell_k, x_k)$
 (non-interference) $Inv(g, \ell_1, x_1, \dots, \ell_k, x_k) \wedge$
 $Inv(g, \ell^\dagger, x^\dagger, \ell_2, x_2, \dots, \ell_k, x_k) \wedge$
 $\vdots$
 $Inv(g, \ell_1, x_1, \dots, \ell_{k-1}, x_{k-1}, \ell^\dagger, x^\dagger) \wedge s(g, x^\dagger, g', \cdot) \rightarrow Inv(g', \ell_1, x_1, \dots, \ell_k, x_k)$
 (safe) $Inv(g, \ell_1, x_1, \dots, \ell_k, x_k) \wedge err(g, \ell_1, x_1, \dots, \ell_m, x_m) \rightarrow false$

Figure 6. Horn clause encoding for thread modularity at level k (where (ℓ_i, s, ℓ'_i) and $(\ell^\dagger, s, \cdot)$ refer to statement s on a thread from ℓ_i to ℓ'_i and, respectively, from $\ell^\dagger$ to some other location in the control flow graph)

Hoenicke et al. Thread Modularity at Many Levels. POPL'17

$$\left\{ R(g, p_{\sigma(1)}, l_{\sigma(1)}, \dots, p_{\sigma(k)}, l_{\sigma(k)}) \leftarrow dist(p_1, \dots, p_k) \wedge R(g, p_1, l_1, \dots, p_k, l_k) \right\}_{\sigma \in S_k} \quad (6)$$

$$R(g, p_1, l_1, \dots, p_k, l_k) \leftarrow dist(p_1, \dots, p_k) \wedge Init(g, l_1) \wedge \dots \wedge Init(g, l_k) \quad (7)$$

$$R(g', p_1, l'_1, \dots, p_k, l_k) \leftarrow dist(p_1, \dots, p_k) \wedge ((g, l_1) \xrightarrow{p_1} (g', l'_1)) \wedge R(g, p_1, l_1, \dots, p_k, l_k) \quad (8)$$

$$R(g', p_1, l_1, \dots, p_k, l_k) \leftarrow dist(p_0, p_1, \dots, p_k) \wedge ((g, l_0) \xrightarrow{p_0} (g', l'_0)) \wedge RConj(0, \dots, k) \quad (9)$$

$$false \leftarrow dist(p_1, \dots, p_r) \wedge \left(\bigwedge_{j=1, \dots, m} (p_j = p_j \wedge (g, l_j) \in E_j) \right) \wedge RConj(1, \dots, r) \quad (10)$$

Figure 4: Horn constraints encoding a homogeneous infinite system with the help of a k -indexed invariant. S_k is the symmetric group on $\{1, \dots, k\}$, i.e., the group of all permutations of k numbers; as an optimisation, any generating subset of S_k , for instance transpositions, can be used instead of S_k . In (10), we define $r = \max\{m, k\}$.

Hojjat et al. Horn Clauses for Communicating Timed Systems. HCVS'14

$$Init(i, j, \bar{v}) \wedge Init(j, i, \bar{v}) \wedge$$

$$Init(i, i, \bar{v}) \wedge Init(j, j, \bar{v}) \Rightarrow I_2(i, j, \bar{v})$$

$$I_2(i, j, \bar{v}) \wedge Tr(i, \bar{v}, \bar{v}') \Rightarrow I_2(i, j, \bar{v}') \quad (3)$$

$$I_2(i, j, \bar{v}) \wedge Tr(j, \bar{v}, \bar{v}') \Rightarrow I_2(i, j, \bar{v}') \quad (4)$$

$$I_2(i, j, \bar{v}) \wedge I_2(i, k, \bar{v}) \wedge I_2(j, k, \bar{v}) \wedge$$

$$Tr(k, \bar{v}, \bar{v}') \wedge k \neq i \wedge k \neq j \Rightarrow I_2(i, j, \bar{v}') \quad (5)$$

$$I_2(i, j, \bar{v}) \Rightarrow \neg Bad(i, j, \bar{v})$$

Figure 3: $VC_2(T)$ for two-quantifier invariants.

Gurfinkel et al. SMT-Based Verification of Parameterized Systems. FSE 2016

Relationship between CHC and Verification

A program satisfies a property iff corresponding CHCs are satisfiable

- satisfiability-preserving transformations == safety preserving

Models for CHC correspond to verification certificates

- inductive invariants and procedure summaries

Unsatisfiability (or derivation of FALSE) corresponds to counterexample

- the resolution derivation (a path or a tree) is the counterexample

CAVEAT: In SeaHorn the terminology is reversed

- SAT means there exists a counterexample – a BMC at some depth is SAT
- UNSAT means the program is safe – BMC at all depths are UNSAT

Semantics of Programming Languages

Denotational Semantics

- Meaning of a program is defined as the mathematical object it computes (e.g., partial functions).
- example: Abstract Interpretation

Axiomatic Semantics

- Meaning of a program is defined in terms of its effect on the truth of logical assertions.
- example: Hoare Logic, Weakest precondition calculus

Operational Semantics

- Meaning of a program is defined by formalizing the individual computation steps of the program.
- example: Natural (Big-Step) Semantics, Structural (Small-Step) Semantics

A Simple Programming Language (WHILE or IMP)

$\text{Prog} ::= \text{def Main}(x) \{ \text{body}_M \}, \dots, \text{def } P(x) \{ \text{body}_P \}$

$\text{body} ::= \text{stmt} (; \text{stmt})^*$

$\text{stmt} ::= x = E \mid \text{assert } (E) \mid \text{assume } (E) \mid$
 $\quad \text{while } E \text{ do } S \mid y = P(E) \mid$
 $\quad L:\text{stmt} \mid \text{goto } L \quad (\text{optional})$

$E := \text{expression over program variables}$

Axiomatic Semantics

An axiomatic semantics consists of:

- a language for stating assertions about programs;
- rules for establishing the truth of assertions.

Some typical kinds of assertions:

- This program terminates.
- If this program terminates, the variables x and y have the same value throughout the execution of the program.
- The array accesses are within the array bounds.

Some typical languages of assertions

- First-order logic
- Other logics (temporal, linear, separation)
- Special-purpose specification languages (Z, Larch, JML)

Assertions for WHILE

The assertions we make about WHILE programs are of the form:

$$\{A\} c \{B\}$$

with the meaning that:

- If A holds in state q and $q \rightarrow q'$
- then B holds in q'

A is the precondition and B is the post-condition

For example:

$$\{y \leq x\} z := x; z := z + 1 \{y < z\}$$

is a valid assertion

These are called **Hoare triples** or **Hoare assertions**

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 IFF $Pre \Rightarrow \mathbf{wlp} (P, Post)$

Horn Clauses by Weakest Liberal Precondition

$\text{Prog} ::= \text{def Main}(x) \{ \text{body}_M \}, \dots, \text{def } P(x) \{ \text{body}_P \}$

$\text{wlp}(x=E, Q) = \text{let } x=E \text{ in } Q$

$\text{wlp}(\text{assert}(E), Q) = E \wedge Q$

$\text{wlp}(\text{assume}(E), Q) = E \Rightarrow Q$

$\text{wlp}(\text{while } E \text{ do } S, Q) = I(w) \wedge$

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

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

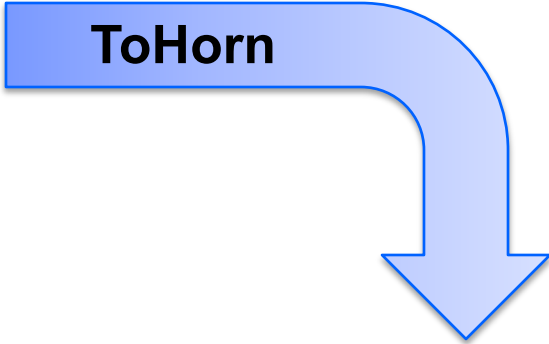
ToHorn ($\text{def } P(x) \{ S \}$) = $\text{wlp}(x_0=x; \text{assume}(p_{\text{pre}}(x)); S, p(x_0, \text{ret}))$

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

Example of a WLP Horn Encoding

```
{Pre:  $y \geq 0$ }  
   $x_o = x$ ;  
   $y_o = y$ ;  
  while  $y > 0$  do  
     $x = x+1$ ;  
     $y = y-1$ ;  
{Post:  $x = x_o + y_o$ }
```

ToHorn



```
C1:  $I(x, y, x, y) \leftarrow y \geq 0$ .  
C2:  $I(x+1, y-1, x_o, y_o) \leftarrow I(x, y, x_o, y_o), y > 0$ .  
C3:  $\text{false} \leftarrow I(x, y, x_o, y_o), y \leq 0, x \neq x_o + y_o$ 
```

$\{y \geq 0\} P \{x = x_{\text{old}} + y_{\text{old}}\}$ is **valid** IFF the $C_1 \wedge C_2 \wedge C_3$ is **satisfiable**

EXAMPLE

Control Flow Graph

basic block

A CFG is a graph of basic blocks

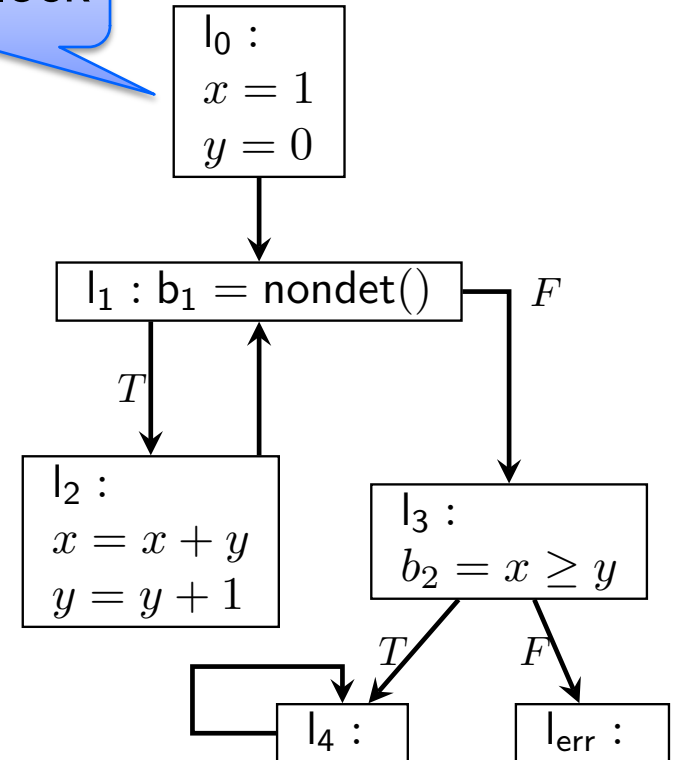
- edges represent different control flow

A CFG corresponds to a program syntax

- where statements are restricted to the form

$L_i : S ; \text{goto } L_j$

and S is control-free (i.e., assignments and procedure calls)



Dual WLP

Dual weakest liberal pre-condition

$$\mathbf{dual-wlp} (P, \text{Post}) = \neg \mathbf{wlp} (P, \neg \text{Post})$$

$s \in \mathbf{dual-wlp} (P, \text{Post})$ IFF there exists an execution of P that starts in s and ends in Post

$\mathbf{dual-wlp} (P, \text{Post})$ is the weakest condition ensuring that an execution of P can reach a state in Post

Examples of dual-wlp

$$\text{dual-wlp}(\text{assume}(E), Q) = \neg \text{wlp}(\text{assume}(E), \neg Q) = \neg(E \Rightarrow \neg Q) = E \wedge Q$$

$$\text{dual-wlp}(x := x+y; y := y+1, x=x' \wedge y=y') = y+1=y' \wedge x+y=x'$$

$$\begin{aligned} & \text{wlp}(x := x + y, \neg(y+1=y \wedge x=x')) \\ &= \text{let } x = x+y \text{ in } \neg(y+1=y' \wedge x=x') \\ &= \neg(y+1=y' \wedge x+y=x') \end{aligned}$$

$$\begin{aligned} & \text{wlp}(y:=y+1, \neg(x=x' \wedge y=y')) \\ &= \text{let } y = y+1 \text{ in } \neg(y=y' \wedge x=x') \\ &= \neg(y+1=y \wedge x=x') \end{aligned}$$

Horn Clauses by Dual WLP

Assumptions

- each procedure is represent by a control flow graph
 - i.e., statements of the form $l_i : S ; \text{ goto } l_j$, where S is loop-free
- program is unsafe iff the last statement of $\text{Main}()$ is reachable
 - i.e., no explicit assertions. All assertions are top-level.

For each procedure $P(x)$, create predicates

- $l(w)$ for each label (i.e., basic block)
 - $p_{\text{en}}(x_\theta, x)$ for entry location of procedure $p()$
 - $p_{\text{ex}}(x_\theta, r)$ for exit location of procedure $p()$
- $p(x, r)$ for each procedure $P(x):r$

Horn Clauses by Dual WLP

The verification condition is a conjunction of clauses:

$$p_{\text{en}}(x_0, x) \leftarrow x_0 = x$$

$$l_j(x_0, w') \leftarrow l_i(x_0, w) \wedge \neg \text{wlp}(S, \neg(w = w'))$$

- for each statement $l_i: S; \text{ goto } l_j$

$$p(x_0, r) \leftarrow p_{\text{ex}}(x_0, r)$$

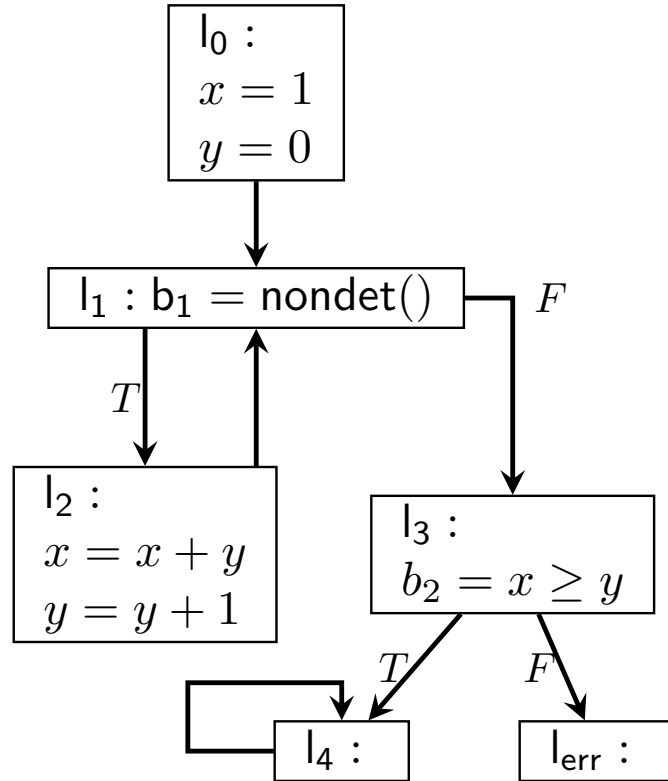
$$\text{false} \leftarrow \text{Main}_{\text{ex}}(x, \text{ret})$$

Example Horn Encoding

```

int x = 1;
int y = 0;
while (*) {
    x = x + y;
    y = y + 1;
}
assert(x ≥ y);

```



- ⟨1⟩ $p_0.$
- ⟨2⟩ $p_1(x, y) \leftarrow p_0, x = 1, y = 0.$
- ⟨3⟩ $p_2(x, y) \leftarrow p_1(x, y).$
- ⟨4⟩ $p_3(x, y) \leftarrow p_1(x, y).$
- ⟨5⟩ $p_1(x', y') \leftarrow p_2(x, y), x' = x + y, y' = y + 1.$
- ⟨6⟩ $p_4 \leftarrow (x \geq y), p_3(x, y).$
- ⟨7⟩ $p_{\text{err}} \leftarrow (x < y), p_3(x, y).$
- ⟨8⟩ $p_4 \leftarrow p_4.$
- ⟨9⟩ $\perp \leftarrow p_{\text{err}}.$

From CFG to Cut Point Graph

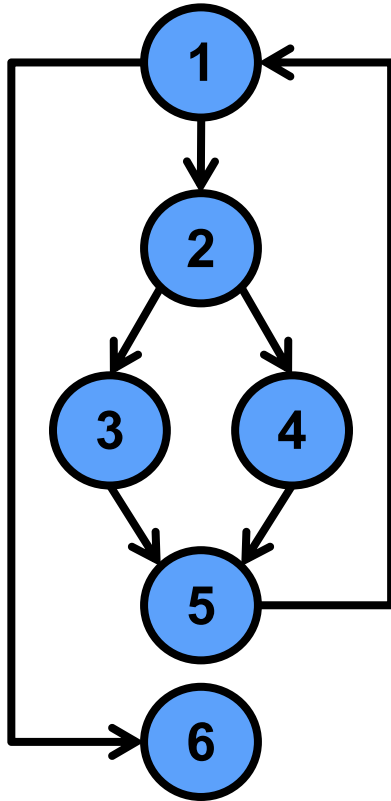
A *Cut Point Graph* hides (summarizes) fragments of a control flow graph by (summary) edges

Vertices (called, *cut points*) correspond to *some* basic blocks

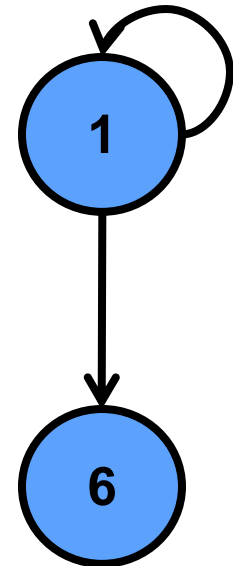
An edge between cut-points *c* and *d* summarizes all finite (loop-free) executions from *c* to *d* that do not pass through any other cut-points

Cut Point Graph Example

CFG



CPG



From CFG to Cut Point Graph

A *Cut Point Graph* hides (summarizes) fragments of a control flow graph by (summary) edges

Cut Point Graph preserves reachability of (not-summarized) control location.

Summarizing loops is undecidable! (Halting program)

A *cutset summary* summarizes all location except for a *cycle cutset* of a CFG. Computing minimal cutset summary is NP-hard (minimal feedback vertex set).

A reasonable compromise is to summarize everything but heads of loops. (Polynomial-time computable).

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

$$x = \text{PHI} (v_0:\text{bb}_0, \dots, v_n:\text{bb}_n) \quad (\text{phi-assignment})$$

“x gets v_i if previously executed block was bb_i ”

Single Static Assignment: An Example

val:bb

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

```
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:
```

Large Step Encoding

Problem: Generate a compact verification condition for a loop-free block of code

```
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:
```

Large Step Encoding: Extract all Actions

$$x_1 = x_0 + y_0$$

$$x_2 = x_0 - y_0$$

$$y_1 = -1 * y_0$$

```
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
```

Example: Encode Control Flow

$$x_1 = x_0 + y_0$$

$$x_2 = x_0 - y_0$$

$$y_1 = -1 * y_0$$

$$B_2 \rightarrow x_0 < N$$

$$B_3 \rightarrow B_2 \wedge y_0 > 0$$

$$B_4 \rightarrow B_2 \wedge y_0 \leq 0$$

$$B_5 \rightarrow (B_3 \wedge x_3 = x_1) \vee \\ (B_4 \wedge x_3 = x_2)$$

$$B_5 \wedge x'_0 = x_3 \wedge y'_0 = y_1$$

$$p_1(x'_0, y'_0) \leftarrow p_1(x_0, y_0), \phi.$$

```
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
```

Summary

Convert body of each procedure into SSA

For each procedure, compute a Cut Point Graph (CPG)

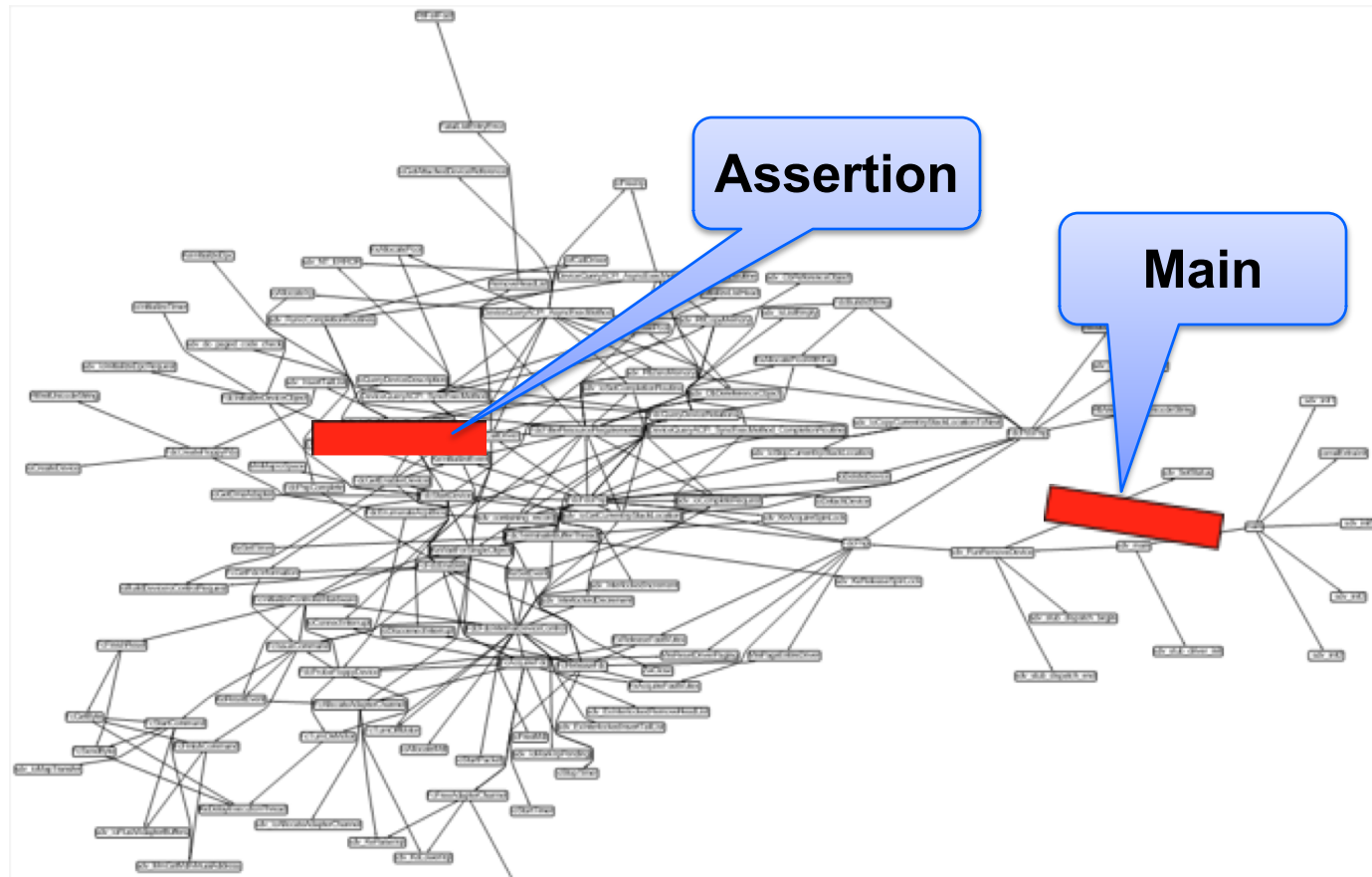
For each edge (s, t) in CPG use dual-wlp to construct the constraint for an execution to flow from s to t

Procedure summary is determined by constraints at the exit point of a procedure

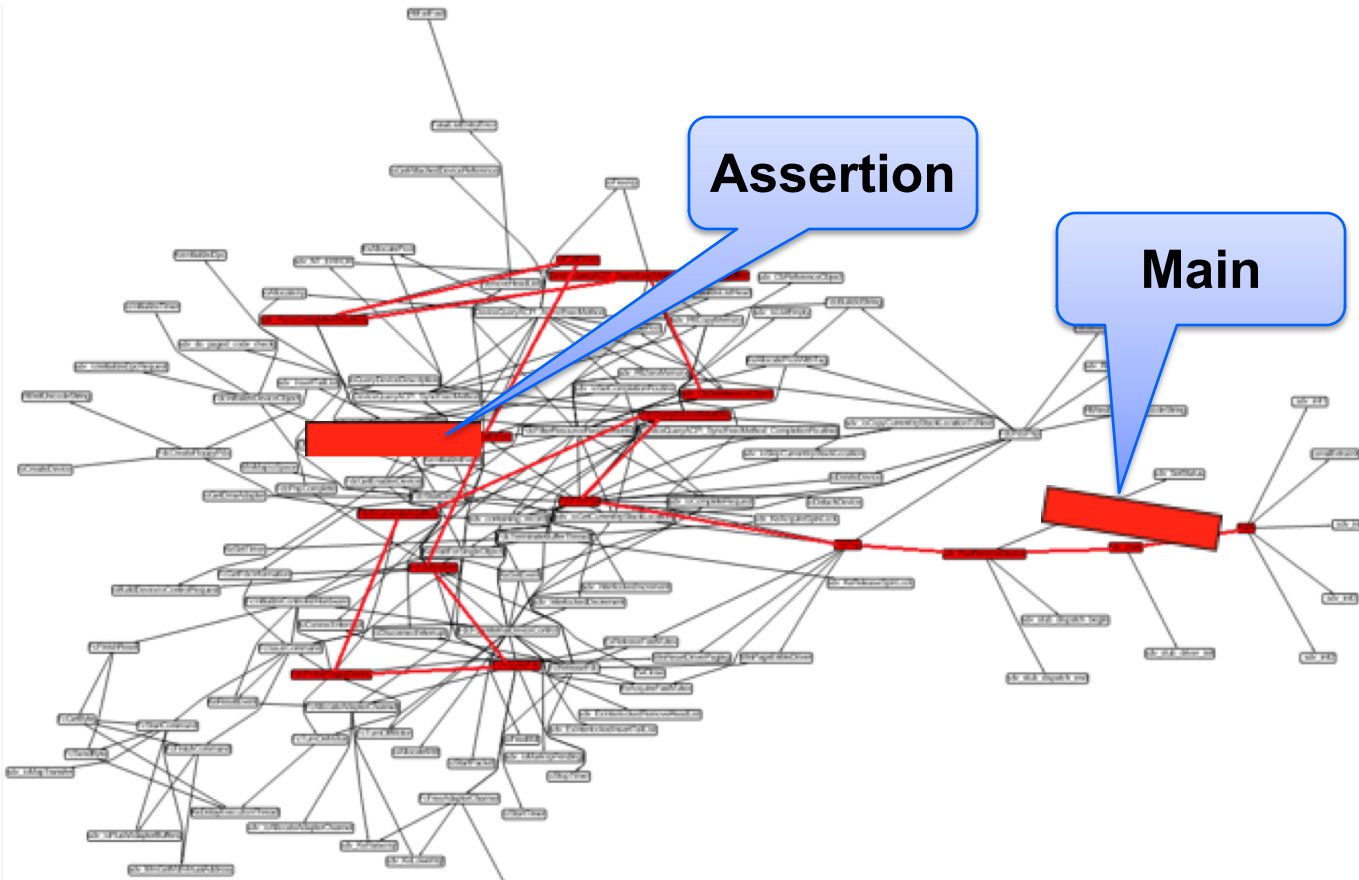
Mixed Semantics

PROGRAM TRANSFORMATION

Deeply nested assertions



Deeply nested assertions



Counter-examples are long

Hard to determine (from main) what is relevant

Mixed Semantics

Stack-free program semantics combining:

- operational (or small-step) semantics
 - i.e., usual execution semantics
- natural (or big-step) semantics: function summary [Sharir-Pnueli 81]
 - $(\sigma, \sigma') \in ||f||$ iff the execution of f on input state σ terminates and results in state σ'
- some execution steps are big, some are small

Non-deterministic executions of function calls

- update top activation record using function summary, or
- enter function body, forgetting history records (i.e., no return!)

Preserves reachability and non-termination

Theorem: Let K be the operational semantics, K^m the stack-free semantics, and L a program location. Then,

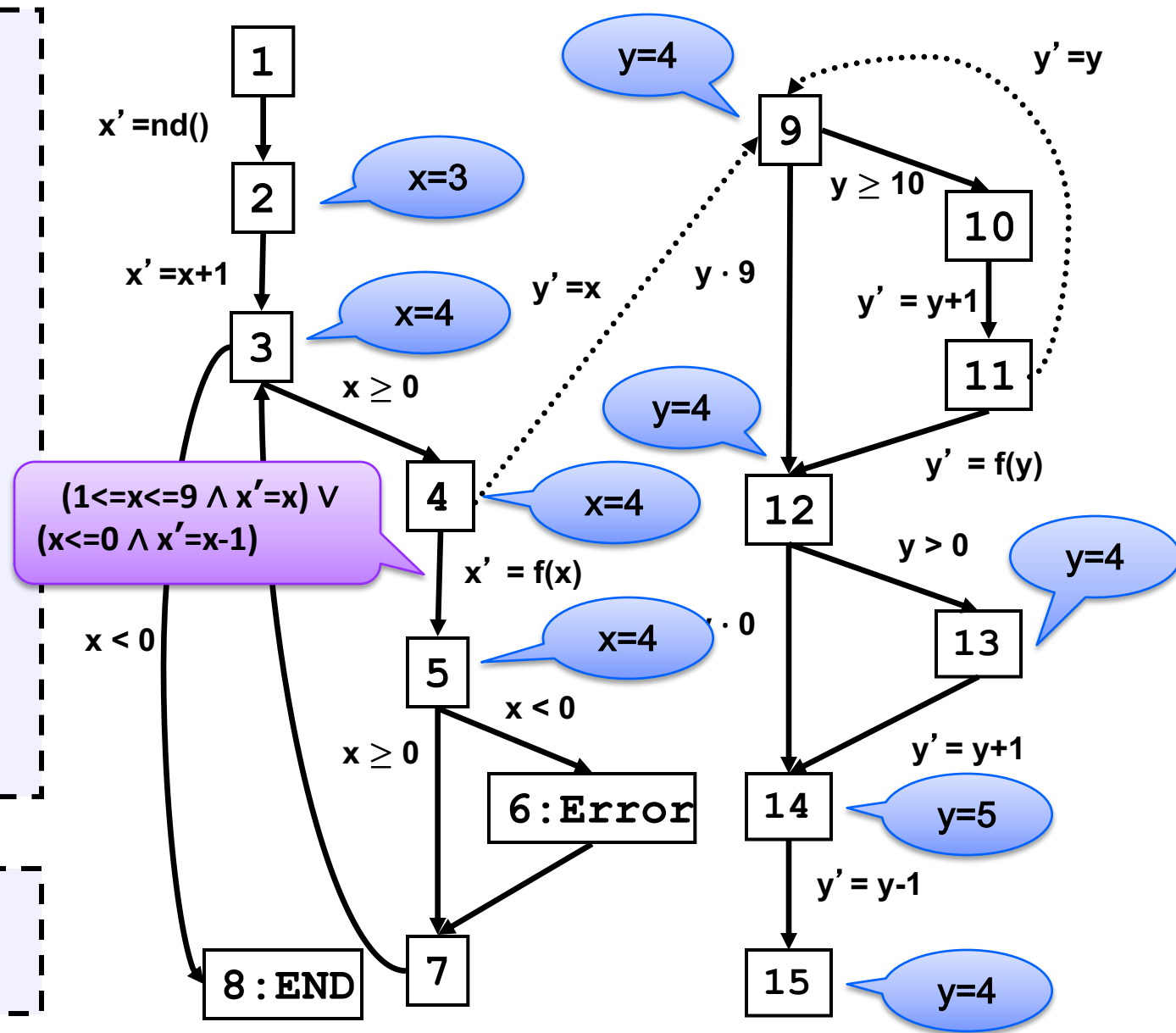
$$K \models EF(pc=L) \Leftrightarrow K^m \models EF(pc=L) \quad \text{and} \quad K \models EG(pc \neq L) \Leftrightarrow K^m \models EG(pc \neq L)$$

```

def main()
1: int x = nd();
2: x = x+1;
3: while(x>=0)
4:   x=f(x);
5:   if(x<0)
6:     Error;
7:
8: END;

def f(int y): ret y
9: if(y>=10){
10:   y=y+1;
11:   y=f(y);
12: } else if(y>0)
13:   y=y+1;
14: y=y-1
15:

```



Mixed Semantics Transformation via Inlining

```
void main() {  
    p1(); p2();  
    assert(c1);  
}  
void p1() {  
    p2();  
    assert(c2);  
}  
void p2() {  
    assert(c3);  
}
```

```
void main() {  
    if(nd()) p1(); else goto p1;  
    if(nd()) p2(); else goto p2;  
    assert(c1);  
    assume(false);  
p1: if (nd) p2(); else goto p2;  
    assume(!c2);  
    assert(false);  
p2: assume(!c3);  
    assert(false);  
}  
void p1() {p2(); assume(c2);}  
void p2() {assume(c3);}
```

Mixed Semantics: Summary

Every procedure is inlined at most once

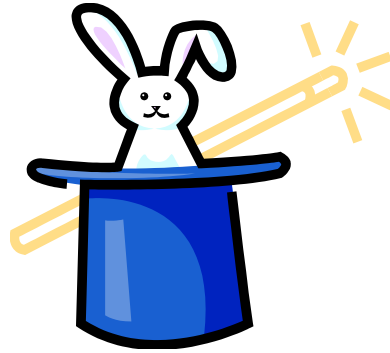
- in the worst case, doubles the size of the program
- can be restricted to only inline functions that directly or indirectly call `error()` function

Easy to implement at compiler level

- create “failing” and “passing” versions of each function
- reduce “passing” functions to returning paths
- in `main()`, introduce new basic block `bb.F` for every failing function `F()`, and call `failing.F` in `bb.F`
- inline all failing calls
- replace every call to `F` to non-deterministic jump to `bb.F` or call to passing `F`

Increases context-sensitivity of context-insensitive analyses

- context of failing paths is explicit in `main` (because of inlining)
- enables / improves many traditional analyses



SOLVING CONSTRAINED HORN CLAUSES

A Magician's Guide to Solving Undecidable Problems

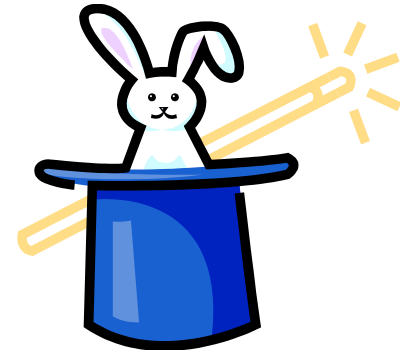
Develop a procedure **P** for a decidable problem

Show that **P** is a decision procedure for the problem

- e.g., model checking of finite-state systems

Choose one of

- Always terminate with some answer (over-approximation)
- Always make useful progress (under-approximation)



Extend procedure **P** to procedure **Q** that “solves” the undecidable problem

- Ensure that **Q** is still a decision procedure whenever **P** is
- Ensure that **Q** either always terminates or makes progress

Procedures for Solving CHC(T)

Predicate abstraction by lifting Model Checking to HORN

- QARMC, Eldarica, ...

Maximal Inductive Subset from a finite Candidate space (Houdini)

- TACAS'18: hoice, FreqHorn

Machine Learning

- PLDI'18: sample, ML to guess predicates, DT to guess combinations

Abstract Interpretation (Poly, intervals, boxes, arrays...)

- Approximate least model by an abstract domain (SeaHorn, ...)

Interpolation-based Model Checking

- Duality, QARMC, ...

SMT-based Unbounded Model Checking (IC3/PDR)

- Spacer, Implicit Predicate Abstraction

Linear CHC Satisfiability

Satisfiability of a set of linear CHCs is reducible to satisfiability of THREE clauses of the form

$$\begin{array}{c} Init(X) \rightarrow P(X) \\ P(X) \wedge Tr(X, X') \rightarrow P(X') \\ P(X) \rightarrow \neg Bad(X) \end{array}$$

where, $X' = \{x' \mid x \in X\}$, P a fresh predicate, and $Init$, Bad , and Tr are constraints

Proof:

add extra arguments to distinguish between predicates

$$\frac{Q(y) \wedge \phi \rightarrow W(y, z)}{P(\text{id}='Q', y) \wedge \phi \rightarrow P(\text{id}='W', y, z)}$$

IC3, PDR, and Friends (1)

IC3: A SAT-based Hardware Model Checker

- Incremental Construction of Inductive Clauses for Indubitable Correctness
- A. Bradley: SAT-Based Model Checking without Unrolling. VMCAI 2011

PDR: Explained and extended the implementation

- Property Directed Reachability
- N. Eén, A. Mishchenko, R. K. Brayton: Efficient implementation of property directed reachability. FMCAD 2011

PDR with Predicate Abstraction (easy extension of IC3/PDR to SMT)

- A. Cimatti, A. Griggio, S. Mover, St. Tonetta: IC3 Modulo Theories via Implicit Predicate Abstraction. TACAS 2014
- J. Birgmeier, A. Bradley, G. Weissenbacher: Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). CAV 2014

IC3, PDR, and Friends (2)

GPDR: Non-Linear CHC with Arithmetic constraints

- Generalized Property Directed Reachability
- K. Hoder and N. Bjørner: Generalized Property Directed Reachability. SAT 2012

SPACER: Non-Linear CHC with Arithmetic

- fixes an incompleteness issue in GPDR and extends it with under-approximate summaries
- A. Komuravelli, A. Gurfinkel, S. Chaki: SMT-Based Model Checking for Recursive Programs. CAV 2014

PolyPDR: Convex models for Linear CHC

- simulating Numeric Abstract Interpretation with PDR
- N. Bjørner and A. Gurfinkel: Property Directed Polyhedral Abstraction. VMCAI 2015

ArrayPDR: CHC with constraints over Arithmetic + Arrays

- Required to model heap manipulating programs
- A. Komuravelli, N. Bjørner, A. Gurfinkel, K. L. McMillan: Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays. FMCAD 2015

IC3, PDR, and Friends (3)

Quip: Forward Reachable States + Conjectures

- Use both forward and backward reachability information
- A. Gurfinkel and A. Ivrii: Pushing to the Top. FMCAD 2015

Avy: Interpolation with IC3

- Use SAT-solver for blocking, IC3 for pushing
- Y. Vizel, A. Gurfinkel: Interpolating Property Directed Reachability. CAV 2014

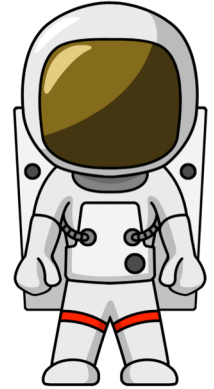
uPDR: Constraints in EPR fragment of FOL

- Universally quantified inductive invariants (or their absence)
- A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, S. Shoham: Property-Directed Inference of Universal Invariants or Proving Their Absence. CAV 2015

Quic3: Universally quantified invariants for LIA + Arrays

- Extending Spacer with quantified reasoning
- A. Gurfinkel, S. Shoham, Y. Vizel: Quantifiers on Demand. ATVA 2018

Spacer: Solving SMT-constrained CHC



Spacer: a solver for SMT-constrained Horn Clauses

- now the default (and only) CHC solver in Z3
 - <https://github.com/Z3Prover/z3>
 - dev branch at <https://github.com/agurfinkel/z3>

Supported SMT-Theories

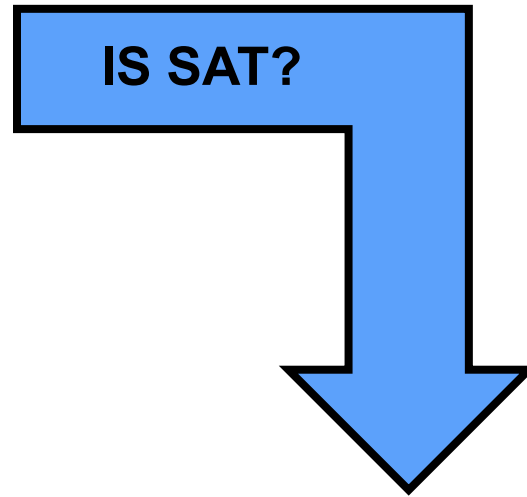
- Linear Real and Integer Arithmetic
- Quantifier-free theory of arrays
- Universally quantified theory of arrays + arithmetic
- Best-effort support for many other SMT-theories
 - data-structures, bit-vectors, non-linear arithmetic

Support for Non-Linear CHC

- for procedure summaries in inter-procedural verification conditions
- for compositional reasoning: abstraction, assume-guarantee, thread modular, etc.

Program Verification with HORN(LIA)

```
z = x; i = 0;  
assume (y > 0);  
while (i < y) {  
    z = z + 1;  
    i = i + 1;  
}  
assert(z == x + y);
```



$z = x \ \& \ i = 0 \ \& \ y > 0$	$\rightarrow$	$\text{Inv}(x, y, z, i)$
$\text{Inv}(x, y, z, i) \ \& \ i < y \ \& \ z1=z+1 \ \& \ i1=i+1$	$\rightarrow$	$\text{Inv}(x, y, z1, i1)$
$\text{Inv}(x, y, z, i) \ \& \ i \geq y \ \& \ z \neq x+y$	$\rightarrow$	false

In SMT-LIB

```
(set-logic HORN)

;; Inv(x, y, z, i)
(declare-fun Inv ( Int Int Int Int) Bool)

(assert
  (forall ( (A Int) (B Int) (C Int) (D Int))
    (=> (and (> B 0) (= C A) (= D 0))
      (Inv A B C D)))
)
(assert
  (forall ( (A Int) (B Int) (C Int) (D Int) (C1 Int) (D1 Int) )
    (=>
      (and (Inv A B C D) (< D B) (= C1 (+ C 1)) (= D1 (+ D
1))))
    (Inv A B C1 D1)
  )
)
(assert
  (forall ( (A Int) (B Int) (C Int) (D Int))
    (=> (and (Inv A B C D) (>= D B) (not (= C (+ A B)))))
      false
    )
  )
)

(check-sat)
(get-model)
```

\$ z3 add-by-one.smt2

```
sat
(model
  (define-fun Inv ((x!0 Int) (x!1 Int) (x!2 Int) (x!3 Int)) Bool
    (and (<= (+ x!2 (* (- 1) x!0) (* (- 1) x!3)) 0)
      (<= (+ x!2 (* (- 1) x!0) (* (- 1) x!1)) 0)
      (<= (+ x!0 x!3 (* (- 1) x!2)) 0)))
  )
```

$\text{Inv}(x, y, z, i)$

$z = x + i$

$z \leq x + y$

IC3/PDR: Solving Linear (Propositional) CHC

Unreachable and Reachable

- terminate the algorithm when a solution is found

Unfold

- increase search bound by 1

Candidate

- choose a bad state in the last frame

Decide

- extend a cex (backward) consistent with the current frame
- choose an assignment s s.t. $(s \wedge F_i \wedge Tr \wedge cex')$ is SAT

Conflict

- construct a lemma to explain why cex cannot be extended
- Find a clause L s.t. $L \Rightarrow \neg cex$, $Init \Rightarrow L$, and $L \wedge F_i \wedge Tr \Rightarrow L'$

Induction

- propagate a lemma as far into the future as possible

From Propositional PDR to Solving CHC

Theories with infinitely many models

- infinitely many satisfying assignments
- can't simply enumerate (when computing predecessor)
- can't block one assignment at a time (when blocking)

Non-Linear Horn Clauses

- multiple predecessors (when computing predecessors)

The problem is undecidable in general, but we want an algorithm that makes progress

- doesn't get stuck in a decidable sub-problem
- guaranteed to find a counterexample (if it exists)

IC3/PDR: Solving Linear (Propositional) CHC

Unreachable and Reachable

- terminate the algorithm when a solution is found

Unfold

- increase search bound by 1

Candidate

- choose a bad state in the last frame

Decide

- extend a cex (backward) consistent with the current frame
- choose an assignment s s.t. $(s \wedge R_i \wedge Tr \wedge cex')$ is SAT

Conflict

- construct a lemma to explain why cex cannot be extended
- Find a clause L s.t. $L \Rightarrow \neg cex$, $Init \Rightarrow L$, and $L \wedge R_i \wedge Tr \Rightarrow L'$

Induction

- propagate a lemma as far into the future as possible

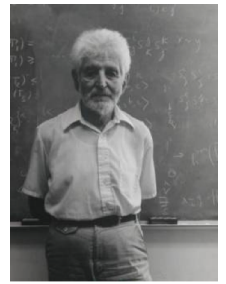
**Theory
dependent**

$$\begin{aligned} ((F_i \wedge Tr) \vee Init') &\Rightarrow \varphi' \\ \varphi' &\Rightarrow \neg c' \end{aligned}$$

Looking for ϕ'

ARITHMETIC CONFLICT

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

$$A \Rightarrow I \quad I \Rightarrow \neg B \quad \Sigma(I) \in \Sigma(A) \cap \Sigma(B)$$

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

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

Examples of Craig Interpolation for Theories

Boolean logic

$$A = (\neg b \wedge (\neg a \vee b \vee c) \wedge a)$$

$$B = (\neg a \vee \neg c)$$

$$ITP(A, B) = a \wedge c$$

Equality with Uninterpreted Functions (EUF)

$$A = (f(a) = b \wedge p(f(a)))$$

$$B = (b = c \wedge \neg p(c))$$

$$ITP(A, B) = p(b)$$

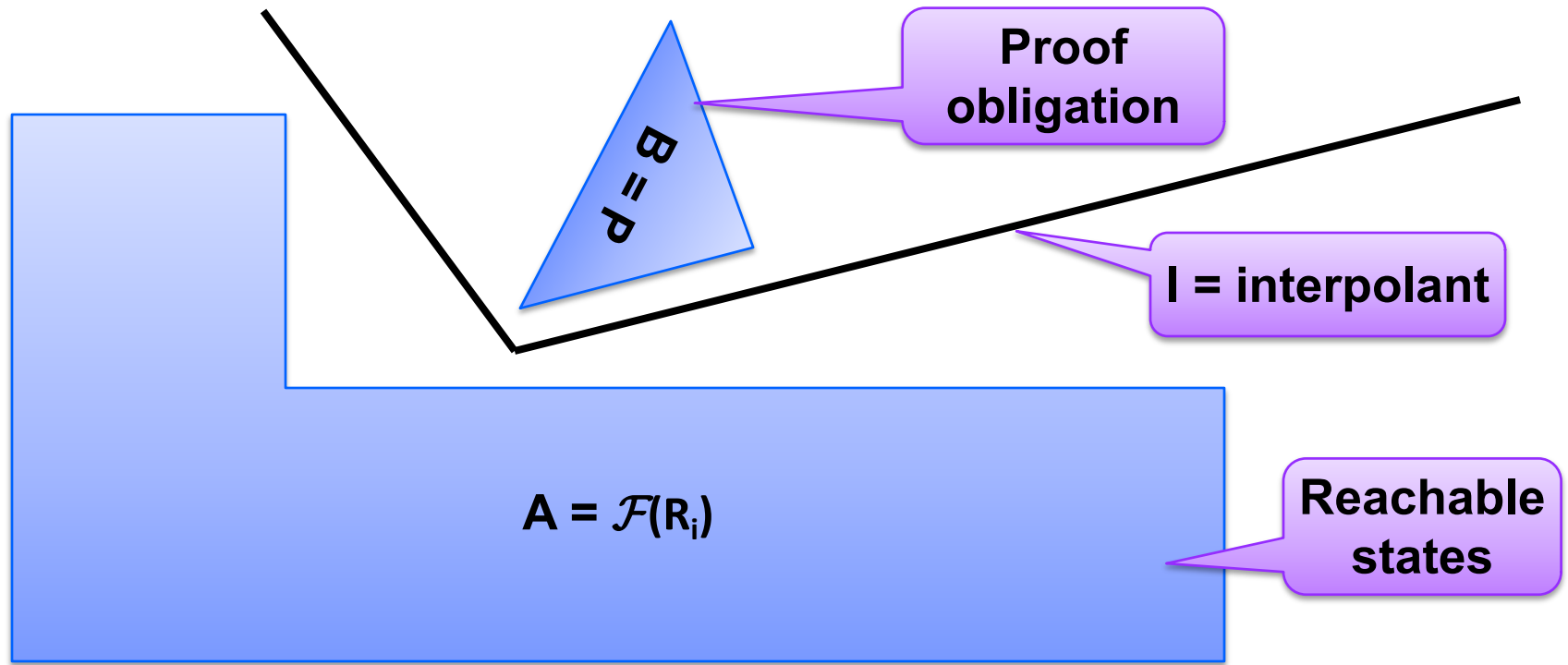
Linear Real Arithmetic (LRA)

$$A = (z + x + y > 10 \wedge z < 5)$$

$$B = (x < -5 \wedge y < -3)$$

$$ITP(A, B) = x + y > 5$$

Craig Interpolation for Linear Arithmetic



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

- $I \in \text{ITP}(A, B)$ then $\neg I \in \text{ITP}(B, A)$
- 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

Arithmetic Conflict

Notation: $\mathcal{F}(A) = (A(X) \wedge Tr) \vee Init(X')$.

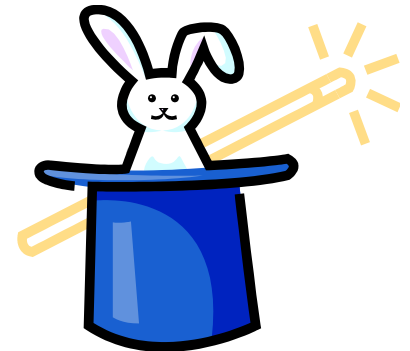
Conflict For $0 \leq i < N$, given a counterexample $\langle P, i+1 \rangle \in Q$ s.t.
 $\mathcal{F}(F_i) \wedge P'$ is unsatisfiable, add $P^\uparrow = \text{ITP}(\mathcal{F}(F_i), P')$ to F_j for $j \leq i+1$.

Counterexample is blocked using Craig Interpolation

- summarizes the reason why the counterexample cannot be extended

Generalization is not inductive

- weaker than IC3/PDR
- inductive generalization for arithmetic is still an open problem



Computing Interpolants for IC3/PDR

Much simpler than general interpolation problem for $A \wedge B$

- B is always a conjunction of literals
- A is dynamically split into DNF by the SMT solver
- DPLL(T) proofs do not introduce new literals

Interpolation algorithm is reduced to analyzing all theory lemmas in a DPLL(T) proof produced by the solver

- every theory-lemma that mixes B-pure literals with other literals is interpolated to produce a single literal in the final solution
- interpolation is restricted to clauses of the form $(\wedge B_i \Rightarrow \vee A_j)$

Interpolating (UNSAT) Cores

- improve interpolation algorithms and definitions to the specific case of PDR
- classical interpolation focuses on eliminating non-shared literals
- in PDR, the focus is on finding good generalizations

Farkas Lemma

Let $M = t_1 \geq b_1 \wedge \dots \wedge t_n \geq b_n$, where t_i are linear terms and b_i are constants

M is *unsatisfiable* iff $0 \geq 1$ is derivable from M by resolution

M is *unsatisfiable* iff $M \vdash 0 \geq 1$

- e.g., $x + y > 10, -x > 5, -y > 3 \vdash (x+y-x-y) > (10 + 5 + 3) \vdash 0 > 18$

M is unsatisfiable iff there exist *Farkas* coefficients $g_1, \dots, g_n$ such that

- $g_i \geq 0$
- $g_1 \times t_1 + \dots + g_n \times t_n = 0$
- $g_1 \times b_1 + \dots + g_n \times b_n \geq 1$

Frakas Lemma Example

Interpolants

$$z + x + y > 10 \quad \times 1$$

$$-z > -5 \quad \times 1$$

$$\left. \begin{array}{l} z + x + y > 10 \\ -z > -5 \end{array} \right\} x + y > 5$$

$$-x > 5 \quad \times 1$$

$$-y > 3 \quad \times 1$$

$$\left. \begin{array}{l} -x > 5 \\ -y > 3 \end{array} \right\} x + y < -8$$

$$0 > 13$$

Interpolation for Linear Real Arithmetic

Let $M = A \wedge B$ be UNSAT, where

- $A = t_1 \geq b_1 \wedge \dots \wedge t_i \geq b_i$, and
- $B = t_{i+1} \geq b_i \wedge \dots \wedge t_n \geq b_n$

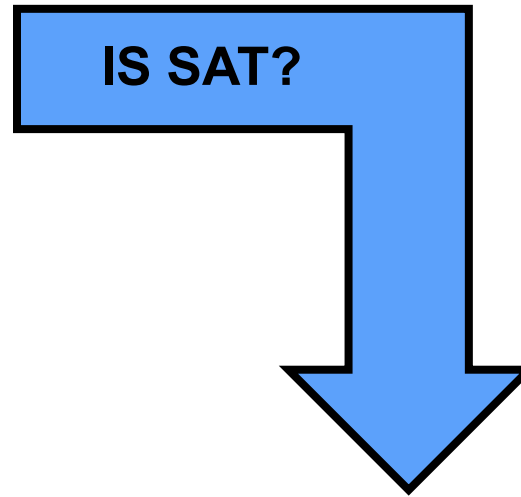
Let $g_1, \dots, g_n$ be the Farkas coefficients witnessing UNSAT

Then

- $g_1 \times (t_1 \geq b_1) + \dots + g_i \times (t_i \geq b_i)$ is an interpolant between A and B
- $g_{i+1} \times (t_{i+1} \geq b_i) + \dots + g_n \times (t_n \geq b_n)$ is an interpolant between B and A
- $g_1 \times t_1 + \dots + g_i \times t_i = - (g_{i+1} \times t_{i+1} + \dots + g_n \times t_n)$
- $\neg(g_{i+1} \times (t_{i+1} \geq b_i) + \dots + g_n \times (t_n \geq b_n))$ is an interpolant between A and B

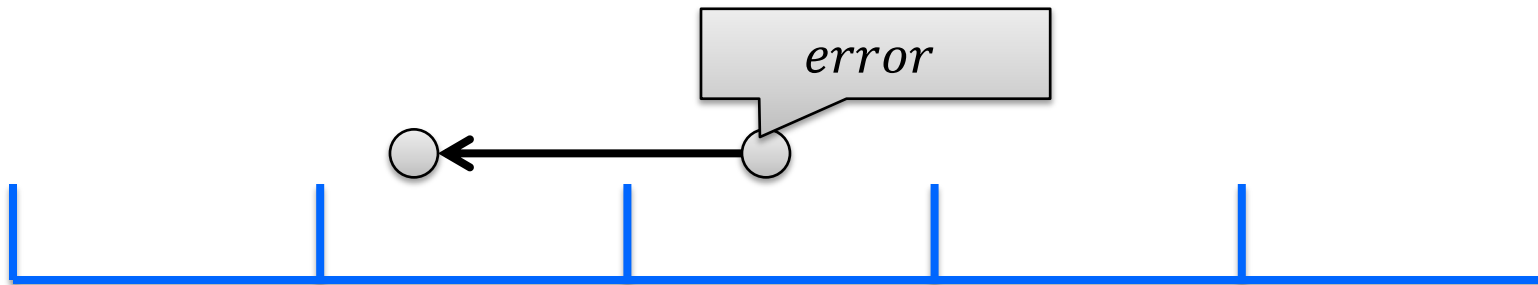
Program Verification with HORN(LIA)

```
z = x; i = 0;  
assume (y > 0);  
while (i < y) {  
    z = z + 1;  
    i = i + 1;  
}  
assert(z == x + y);
```



$z = x \ \& \ i = 0 \ \& \ y > 0$	$\rightarrow$	$\text{Inv}(x, y, z, i)$
$\text{Inv}(x, y, z, i) \ \& \ i < y \ \& \ z1=z+1 \ \& \ i1=i+1$	$\rightarrow$	$\text{Inv}(x, y, z1, i1)$
$\text{Inv}(x, y, z, i) \ \& \ i \geq y \ \& \ z \neq x+y$	$\rightarrow$	false

Lemma Generation Example



Transition Relation

$$x = x_0 \wedge z = z_0 + 1 \wedge i = i_0 + 1 \wedge y > i_0$$

Pob

$$i \geq y \wedge x + y > z$$

Farkas explanation for unsat

$$\begin{array}{c}
 \frac{x_0 + y_0 \leq z_0, \quad x \leq x_0, \quad z_0 < z, \quad i \leq i_0 + 1}{x + i \leq z} \qquad \frac{i \geq y, \quad x + y > z}{x + i > z} \\
 \hline
 \text{false}
 \end{array}$$

Learn lemma:

$$x + i \leq z$$

Interpolation Problem in Spacer

Given an arbitrary LRA formula A and a conjunction of literals s such that $A \wedge s$ are UNSAT, compute an interpolant I such that

- $s \Rightarrow I$ $I \wedge A \Rightarrow \text{FALSE}$ I is over symbols common to s and A

Use an SMT solver to decide that $s \wedge A$ are UNSAT

- SMT solver uses LRA theory lemmas (called Farkas Theory Lemmas) of the form:
 $\neg ((s_1 \wedge \dots \wedge s_k) \wedge (a_1 \wedge \dots \wedge a_m))$
where s_i are literals from s and a_i are literals from A
- For each such lemma L_j , $((s_1 \wedge \dots \wedge s_k) \wedge (a_1 \wedge \dots \wedge a_m))$ is UNSAT
- Let t_j be an interpolant corresponding to L_j

Then, an interpolant between s and A is a clause of the form

$(\neg t_1 \vee \dots \vee \neg t_k)$ with one literal per each theory lemma

- in practice, interpolation is optimized by examining and restructuring SMT resolution proof, dealing with Boolean reasoning, and global optimization

Computing Interpolants in Spacer

Much simpler than general interpolation problem for $A \wedge B$

- B is always a conjunction of literals
- A is dynamically split into DNF by the SMT solver
- DPLL(T) proofs do not introduce new literals

Interpolation algorithm is reduced to analyzing all theory lemmas in a DPLL(T) proof produced by the solver

- every theory-lemma that mixes B-pure literals with other literals is interpolated to produce a single literal in the final solution
- interpolation is restricted to clauses of the form $(\wedge B_i \Rightarrow \vee A_j)$

Interpolating (UNSAT) Cores

- improve interpolation algorithms and definitions to the specific case of PDR
- classical interpolation focuses on eliminating non-shared literals
- in PDR, the focus is on finding good generalizations

$$s \subseteq pre(c)$$
$$\equiv s \Rightarrow \exists X' . Tr \wedge c'$$

Computing a predecessor s of a counterexample c

ARITHMETIC DECIDE

Model Based Projection

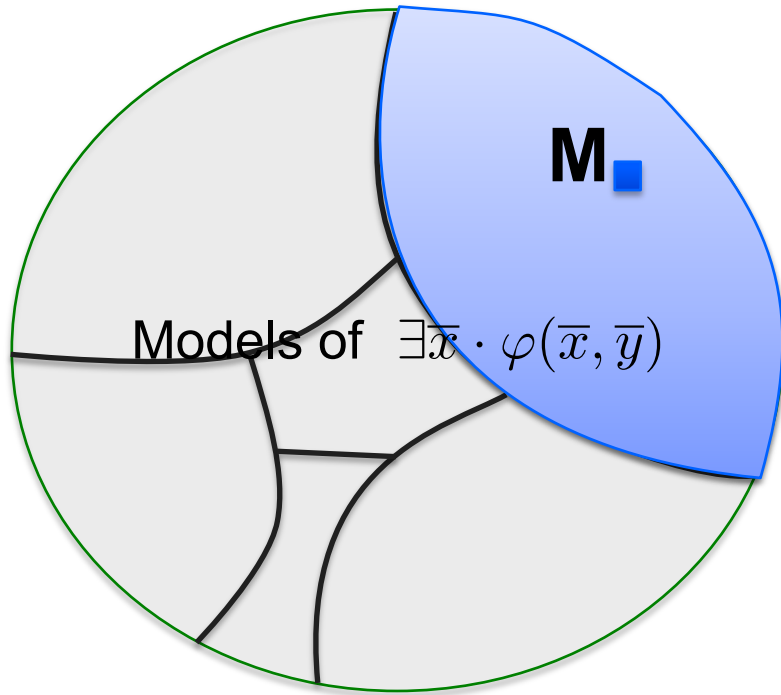
Definition: Let ϕ be a formula, U a set of variables, and M a model of ϕ . Then $\psi = \text{MBP}(U, M, \phi)$ is a Model Based Projection of U , M and ϕ iff

1. ψ is a monomial
2. $\text{Vars}(\psi) \subseteq \text{Vars}(\phi) \setminus U$
3. $M \models \psi$
4. $\psi \Rightarrow \exists U . \phi$

Model Based Projection under-approximates existential quantifier elimination relative to a given model (i.e., satisfying assignment)

Model Based Projection

Expensive to find a quantifier-free $\psi(\bar{y}) \equiv \exists \bar{x} \cdot \varphi(\bar{x}, \bar{y})$



1. Find model M of $\varphi(x,y)$

2. Compute a partition containing M

Quantifier Elimination

A **quantifier elimination** is a procedure that takes a formula of the form $\exists x \psi(x)$ and returns an equivalent formula φ without existential quantifier and without the variable x

- $\text{QELIM}(\exists x \psi(x)) = \varphi$ and $\exists x \psi(x) \Leftrightarrow \varphi$

Quantifier elimination in propositional logic

- $\text{QELIM}(\exists x \psi(x)) = \psi(\text{TRUE}) \vee \psi(\text{FALSE})$

Many theories support quantifier elimination (e.g., linear arithmetic)

- but not all
- No quantifier elimination for EUF, e.g., $(\exists x f(x) \neq g(x))$ cannot be expressed without the existential quantifier

Quantifier elimination is usually expensive

- e.g., propositional qelim is exponential in the number of variables quantified

Loos-Weispfenning Quantifier Elimination for LRA

ϕ is LRA formula in Negation Normal Form

E is set of $x=t$ atoms, U set of $x < t$ atoms, and L set of $s < x$ atoms

There are no other occurrences of x in $\phi[x]$

$$\exists x. \varphi[x] \equiv \varphi[\infty] \vee \bigvee_{x=t \in E} \varphi[t] \vee \bigvee_{x < t \in U} \varphi[t - \epsilon]$$

where

$$(x < t')[t - \epsilon] \equiv t \leq t' \quad (s < x)[t - \epsilon] \equiv s < t \quad (x = e)[t - \epsilon] \equiv \text{false}$$

The case of lower bounds is dual

- using $-\infty$ and $t+\epsilon$

Fourier–Motzkin Quantifier Elimination for LRA

$$\begin{aligned} & \exists x \cdot \bigwedge_i s_i < x \wedge \bigwedge_j x < t_j \\ &= \bigwedge_i \bigwedge_j \text{resolve}(s_i < x, x < t_j, x) \\ &= \bigwedge_i \bigwedge_j s_i < t_j \end{aligned}$$

Quadratic increase in the formula size per each eliminated variable

Quantifier Elimination with Assumptions

$$\begin{aligned} & \left(\bigwedge_{j \neq 0} t_0 \leq t_j \right) \wedge \exists x \cdot \bigwedge_i s_i < x \wedge \bigwedge_j x < t_j \\ = & \left(\bigwedge_{j \neq 0} t_0 \leq t_j \right) \wedge \bigwedge_i \text{resolve}(s_i < x, x < t_0, x) \\ = & \left(\bigwedge_{j \neq 0} t_0 \leq t_j \right) \wedge \bigwedge_i s_i < t_0 \end{aligned}$$

Quantifier elimination is simplified by a choice of a minimal upper bound

- For each choice of minimal upper bound, no increase in term size
- Dually, can use largest lower bound

How to choose the assumptions?!

- MBP == use the order chosen by the model

MBP for Linear Rational Arithmetic

Compute a **single** disjunct from LW-QE that includes the model

- Use the Model to uniquely pick a substitution term for x

$$Mbp_x(M, x = s \wedge L) = L[x \leftarrow s]$$

$$Mbp_x(M, x \neq s \wedge L) = Mbp_x(M, s < x \wedge L) \text{ if } M(x) > M(s)$$

$$Mbp_x(M, x \neq s \wedge L) = Mbp_x(M, -s < -x \wedge L) \text{ if } M(x) < M(s)$$

$$Mbp_x(M, \bigwedge_i s_i < x \wedge \bigwedge_j x < t_j) = \bigwedge_i s_i < t_0 \wedge \bigwedge_j t_0 \leq t_j \text{ where } M(t_0) \leq M(t_i), \forall i$$

MBP techniques have been developed for

- Linear Rational Arithmetic, Linear Integer Arithmetic
- Theories of Arrays, and Recursive Data Types

Arithmetic Decide

Notation: $\mathcal{F}(A) = (A(X) \wedge Tr(X, X') \vee Init(X'))$.

Decide If $\langle P, i+1 \rangle \in Q$ and there is a model $m(X, X')$ s.t. $m \models \mathcal{F}(F_i) \wedge P'$,
add $\langle P_{\downarrow}, i \rangle$ to Q , where $P_{\downarrow} = MBP(X', m, \mathcal{F}(F_i) \wedge P')$.

Compute a predecessor using Model Based Projection

To ensure progress, Decide must be finite

- finitely many possible predecessors when all other arguments are fixed

Alternatively

- Completeness can follow from an interaction of **Decide** and **Conflict**
 - but requires more rules to propagate implicants backward (as in PDR) and forward (as in Spacer and Quip)

PolyPDR: Solving CHC(LRA)

Unreachable and Reachable

- terminate the algorithm when a solution is found

Unfold

- increase search bound by 1

Candidate

- choose a bad state in the last frame

Decide

- extend a cex (backward) consistent with the current frame
- find a model $\mathbf{M}$ of $\mathbf{s}$ s.t. $(F_i \wedge \text{Tr} \wedge \text{cex}')$, and let $\mathbf{s} = \text{MBP}(X', F_i \wedge \text{Tr} \wedge \text{cex}')$

Conflict

- construct a lemma to explain why cex cannot be extended
- Find an interpolant L s.t. $L \Rightarrow \neg \text{cex}$, $\text{Init} \Rightarrow L$, and $F_i \wedge \text{Tr} \Rightarrow L'$

Induction

- propagate a lemma as far into the future as possible

Non-Linear CHC Satisfiability

Satisfiability of a set of arbitrary (i.e., linear or non-linear) CHCs is reducible to satisfiability of THREE (3) clauses of the form

$$\mathit{Init}(X) \rightarrow P(X)$$

$$P(X) \wedge P(X^o) \wedge \mathit{Tr}(X, X^o, X') \rightarrow P(X')$$

$$P(X) \rightarrow \neg \mathit{Bad}(X)$$

where, $X' = \{x' \mid x \in X\}$, $X^o = \{x^o \mid x \in X\}$, P a fresh predicate, and Init , Bad , and Tr are constraints

Generalized GPDR

Input: A safety problem $\langle \text{Init}(X), \text{Tr}(X, X^o, X'), \text{Bad}(X) \rangle$.

Output: *Unreachable* or *Reachable*

Data: A cex queue Q , where a cex $\langle c_0, \dots, c_k \rangle \in Q$ is a tuple, each $c_j = \langle m, i \rangle$, m is a cube over state variables, and $i \in \mathbb{N}$. A level N .
A trace $F_0, F_1, \dots$.

Notation: $\mathcal{F}(A, B) = \text{Init}(X') \vee (A(X) \wedge B(X^o) \wedge \text{Tr})$, and $\mathcal{F}(A) = \mathcal{F}(A, A)$

Initially: $Q = \emptyset$, $N = 0$, $F_0 = \text{Init}$, $\forall i > 0 \cdot F_i = \emptyset$

Require: $\text{Init} \rightarrow \neg \text{Bad}$

repeat

Unreachable If there is an $i < N$ s.t. $F_i \subseteq F_{i+1}$ **return** *Unreachable*.

Reachable if exists $t \in Q$ s.t. for all $\langle c, i \rangle \in t$, $i = 0$, **return** *Reachable*.

Unfold If $F_N \rightarrow \neg \text{Bad}$, then set $N \leftarrow N + 1$ and $Q \leftarrow \emptyset$.

Candidate If for some m , $m \rightarrow F_N \wedge \text{Bad}$, then add $\langle \langle m, N \rangle \rangle$ to Q .

Decide If there is a $t \in Q$, with $c = \langle m, i + 1 \rangle \in t$, $m_1 \rightarrow m$, $l_0 \wedge m_0^o \wedge m'_1$ is satisfiable, and $l_0 \wedge m_0^o \wedge m'_1 \rightarrow F_i \wedge F_i^o \wedge \text{Tr} \wedge m'$ then add $\hat{t}$ to Q , where $\hat{t} = t$ with c replaced by two tuples $\langle l_0, i \rangle$, and $\langle m_0, i \rangle$.

Conflict If there is a $t \in Q$ with $c = \langle m, i + 1 \rangle \in t$, s.t. $\mathcal{F}(F_i) \wedge m'$ is unsatisfiable. Then, add $\varphi = \text{ITP}(\mathcal{F}(F_i), m')$ to F_j , for all $0 \leq j \leq i + 1$.

Leaf If there is $t \in Q$ with $c = \langle m, i \rangle \in t$, $0 < i < N$ and $\mathcal{F}(F_{i-1}) \wedge m'$ is unsatisfiable, then add $\hat{t}$ to Q , where $\hat{t}$ is t with c replaced by $\langle m, i + 1 \rangle$.

Induction For $0 \leq i < N$ and a clause $(\varphi \vee \psi) \in F_i$, if $\varphi \notin F_{i+1}$, $\mathcal{F}(\phi \wedge F_i) \rightarrow \phi'$, then add φ to F_j , for all $j \leq i + 1$.

until ∞ ;

counterexample
is a tree

two
predecessors

theory-aware
Conflict

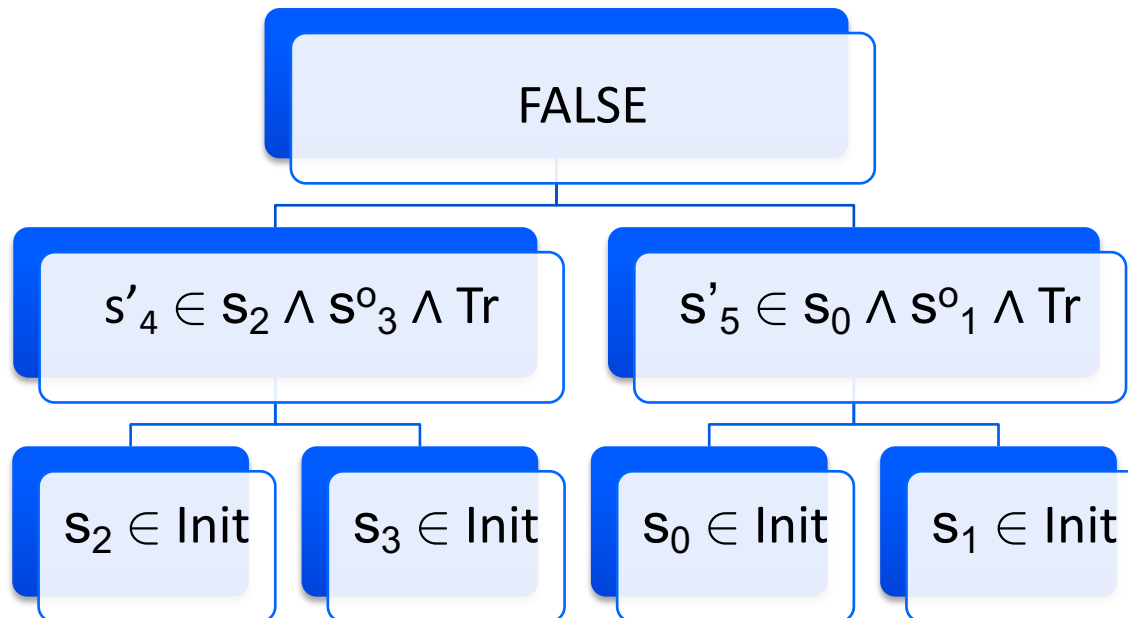
Counterexamples to non-linear CHC

A set S of CHC is unsatisfiable iff S can derive FALSE

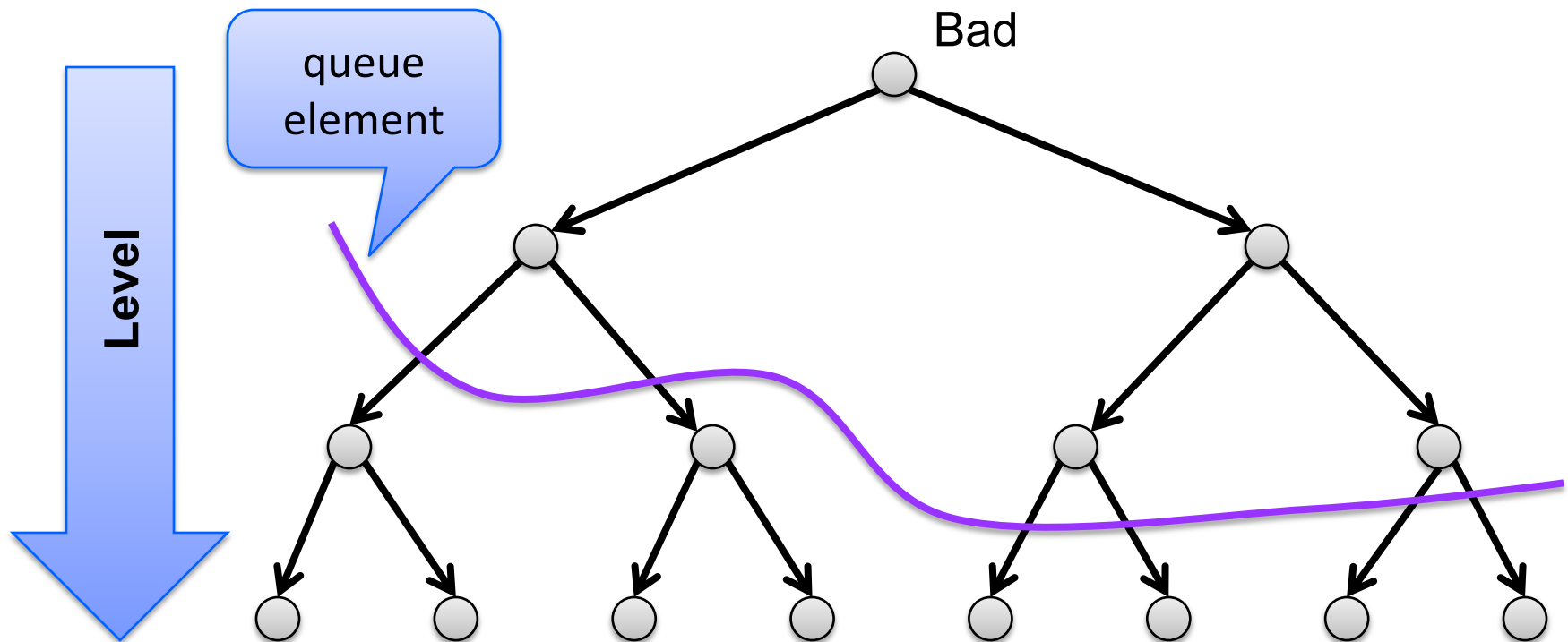
- we call such a derivation a counterexample

For linear CHC, the counterexample is a path

For non-linear CHC, the counterexample is a tree



GPDR Search Space



In Decide, one POB in the frontier is chosen and its two children are expanded

GPDR: Splitting predecessors

Consider a clause

$$P(x) \wedge P(y) \wedge x > y \wedge z = x + y \implies P(z)$$

How to compute a predecessor for a proof obligation $z > 0$

Predecessor over the constraint is:

$$\begin{aligned} & \exists z \cdot x > y \wedge z = x + y \wedge z > 0 \\ = & x > y \wedge x + y > 0 \end{aligned}$$

Need to create two separate proof obligation

- one for $P(x)$ and one for $P(y)$
- gpdr solution: split by substituting values from the model (incomplete)

GPDR: Deciding predecessors

Decide If there is a $t \in Q$, with $c = \langle m, i + 1 \rangle \in t$, $m_1 \rightarrow m$, $l_0 \wedge m_0^o \wedge m_1'$ is satisfiable, and $l_0 \wedge m_0^o \wedge m_1' \rightarrow F_i \wedge F_i^o \wedge Tr \wedge m'$ then add $\hat{t}$ to Q , where $\hat{t} = t$ with c replaced by two tuples $\langle l_0, i \rangle$, and $\langle m_0, i \rangle$.

Compute two predecessors at each application of **GPDR/Decide**

Can explore both predecessors in parallel

- e.g., BFS or DFS exploration order

Number of predecessors is unbounded

- incomplete even for finite problem (i.e., non-recursive CHC)

No caching/summarization of previous decisions

- worst-case exponential for Boolean Push-Down Systems

Spacer

Same queue as
in IC3/PDR

Cache Reachable
states

Three variants of
Decide

Same **Conflict** as
in APDR/GPDR

Input: A safety problem $\langle \text{Init}(X), \text{Tr}(X, X^o, X'), \text{Bad}(X) \rangle$.

Output: *Unreachable* or *Reachable*

Data: A cex queue Q , where a cex $c \in Q$ is a pair $\langle m, i \rangle$, m is a cube over state variables, and $i \in \mathbb{N}$. A level N . A set of reachable states REACH . A trace $F_0, F_1, \dots$

Notation: $\mathcal{F}(A, B) = \text{Init}(X') \vee (A(X) \wedge B(X^o) \wedge \text{Tr})$, and $\mathcal{F}(A) = \mathcal{F}(A, A)$

Initially: $Q = \emptyset, N = 0, F_0 = \text{Init}, \forall i > 0 \cdot F_i = \emptyset, \text{REACH} = \text{Init}$

Require: $\text{Init} \rightarrow \neg \text{Bad}$

repeat

Unreachable If there is an $i < N$ s.t. $F_i \subseteq F_{i+1}$ **return** *Unreachable*.

Reachable If $\text{REACH} \wedge \text{Bad}$ is satisfiable, **return** *Reachable*.

Unfold If $F_N \rightarrow \neg \text{Bad}$, then set $N \leftarrow N + 1$ and $Q \leftarrow \emptyset$.

Candidate If for some $m, m \rightarrow F_N \wedge \text{Bad}$, then add $\langle m, N \rangle$ to Q .

Successor If there is $\langle m, i + 1 \rangle \in Q$ and a model $M \models \psi$, where $\psi = \mathcal{F}(\vee \text{REACH}) \wedge m'$. Then, add s to REACH , where $s' \in \text{MBP}(\{X, X^o\}, \psi)$.

DecideMust If there is $\langle m, i + 1 \rangle \in Q$, and a model $M \models \psi$, where $\psi = \mathcal{F}(F_i, \vee \text{REACH}) \wedge m'$. Then, add s to Q , where $s \in \text{MBP}(\{X^o, X'\}, \psi)$.

DecideMay If there is $\langle m, i + 1 \rangle \in Q$ and a model $M \models \psi$, where $\psi = \mathcal{F}(F_i) \wedge m'$. Then, add s to Q , where $s^o \in \text{MBP}(\{X, X'\}, \psi)$.

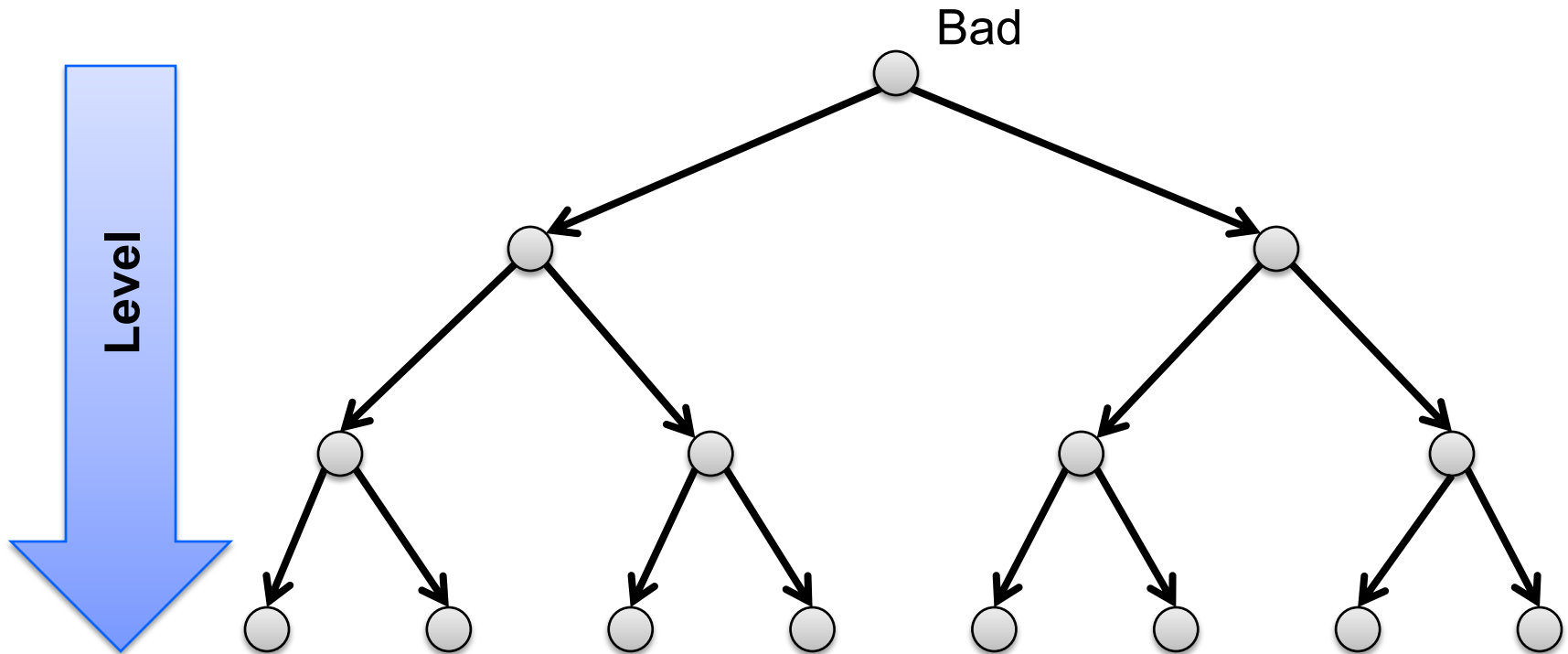
Conflict If there is an $\langle m, i + 1 \rangle \in Q$, s.t. $\mathcal{F}(F_i) \wedge m'$ is unsatisfiable. Then, add $\varphi = \text{ITP}(\mathcal{F}(F_i), m')$ to F_j , for all $0 \leq j \leq i + 1$.

Leaf If $\langle m, i \rangle \in Q, 0 < i < N$ and $\mathcal{F}(F_{i-1}) \wedge m'$ is unsatisfiable, then add $\langle m, i + 1 \rangle$ to Q .

Induction For $0 \leq i < N$ and a clause $(\varphi \vee \psi) \in F_i$, if $\varphi \notin F_{i+1}$, $\mathcal{F}(\phi \wedge F_i) \rightarrow \phi'$, then add φ to F_j , for all $j \leq i + 1$.

until ∞ ;

SPACER Search Space



In Decide, unfold the derivation tree in a fixed depth-first order

- use MBP to decide on counterexamples

Successor: Learn new facts (reachable states) on the way up

- use MBP to propagate facts bottom up

Successor Rule: Computing Reachable States

Successor If there is $\langle m, i + 1 \rangle \in Q$ and a model $M \models \psi$, where $\psi = \mathcal{F}(\text{REACH}) \wedge m'$. Then, add s to REACH, where $s' \in \text{MBP}(\{X, X^o\}, \psi)$.

Computing new reachable states by under-approximating forward image using MBP

- since MBP is finite, guarantee to exhaust all reachable states

Second use of MBP

- orthogonal to the use of MBP in Decide
- can allow REACH to contain auxiliary variables, but this might explode

For Boolean CHC, the number of reachable states is bounded

- complexity is polynomial in the number of states
- same as reachability in Push Down Systems

Decide Rule: Must and May refinement

DecideMust If there is $\langle m, i + 1 \rangle \in Q$, and a model $M \models \psi$, where $\psi = \mathcal{F}(F_i, \text{REACH}) \wedge m'$. Then, add s to Q , where $s \in \text{MBP}(\{X^o, X'\}, \psi)$.

DecideMay If there is $\langle m, i + 1 \rangle \in Q$ and a model $M \models \psi$, where $\psi = \mathcal{F}(F_i) \wedge m'$. Then, add s to Q , where $s^o \in \text{MBP}(\{X, X'\}, \psi)$.

DecideMust

- use computed summary (REACH) to skip over a call site

DecideMay

- use over-approximation of a calling context to guess an approximation of the call-site
- the call-site either refutes the approximation (**Conflict**) or refines it with a witness (**Successor**)