Algorithmic Logic-based Verification

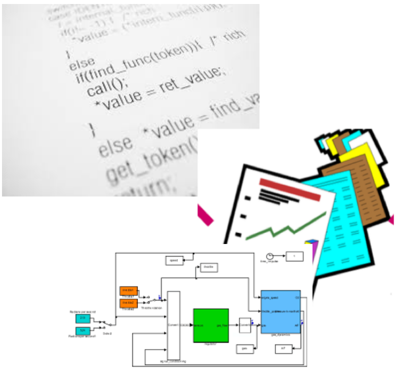
Arie Gurfinkel
Electrical and Computer Engineering
University of Waterloo

Marktoberdorf Summer School 2018

UNIVERSITY OF WATERLOO

Automated (Software) Verification

Program and/or model



Automated Reasoning

Correct

Incorrect

Alan M. Turing. 1936: “Undecidable”

Alan M. Turing. ”Checking a large routine” 1949

How can one check a routine in the sense of making sure that it is right?

programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Automated Verification

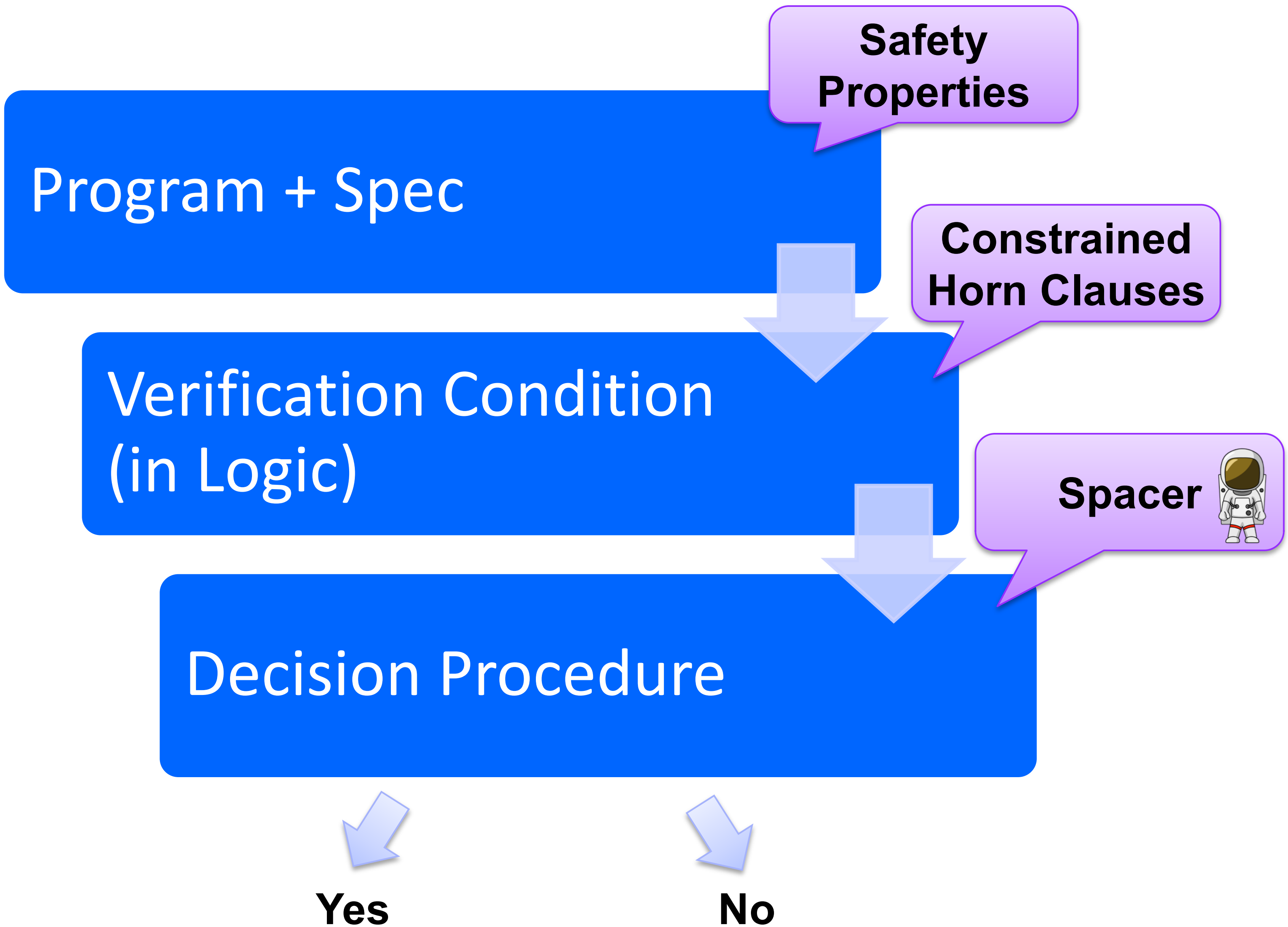
Deductive Verification

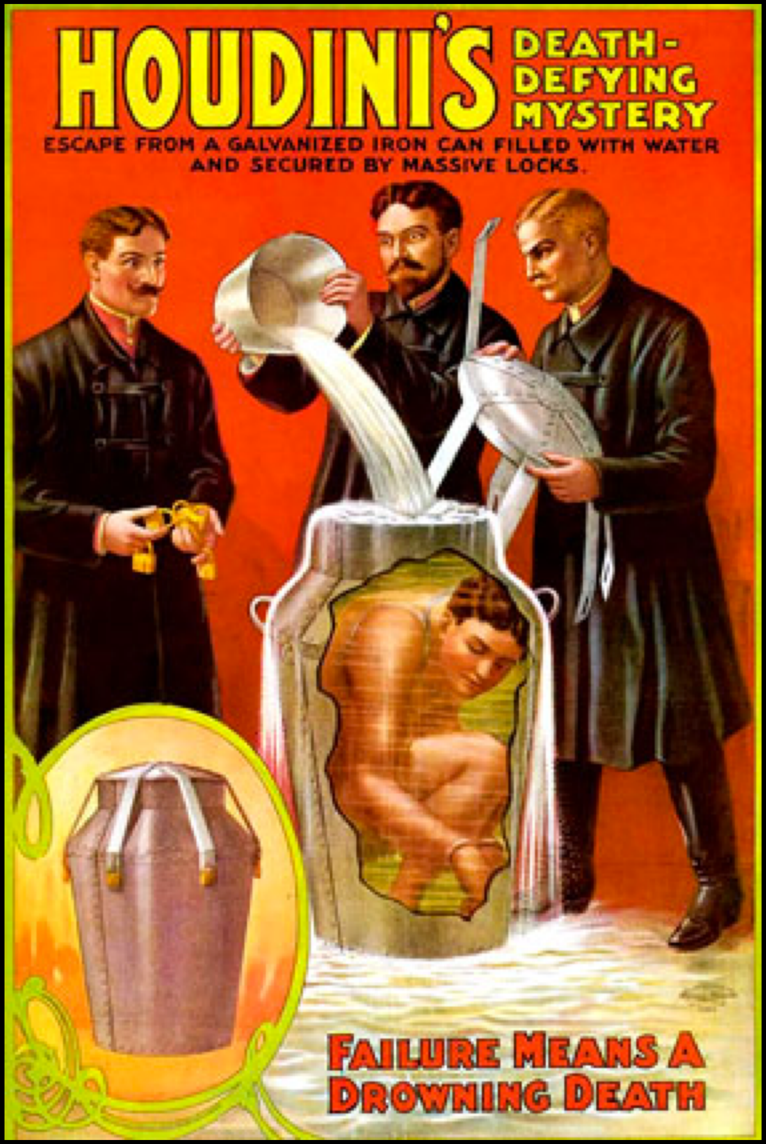
- A user provides a program and a verification certificate
  - e.g., inductive invariant, pre- and post-conditions, function summaries, etc.
- A tool automatically checks validity of the certificate
  - this is not easy! (might even be undecidable)
- Verification is manual but machine certified

Algorithmic Verification

- A user provides a program and a desired specification
  - e.g., program never writes outside of allocated memory
- A tool automatically checks validity of the specification
  - and generates a verification certificate if the program is correct
  - and generates a counterexample if the program is not correct
- Verification is completely automatic – "push-button"

Algorithmic Logic-Based Verification



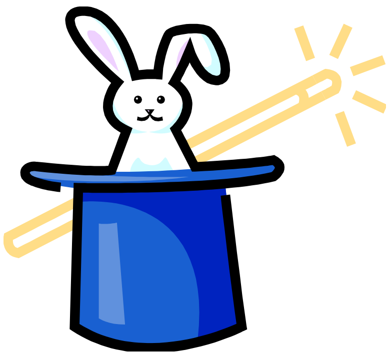

HOUDINI'S
DEATH-DEFYING MYSTERY
ESCAPE FROM A GALVANIZED IRON CAN FILLED WITH WATER AND SECURED BY MASSIVE LOCKS.
FAILURE MEANS A DROWNING DEATH

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

Outline

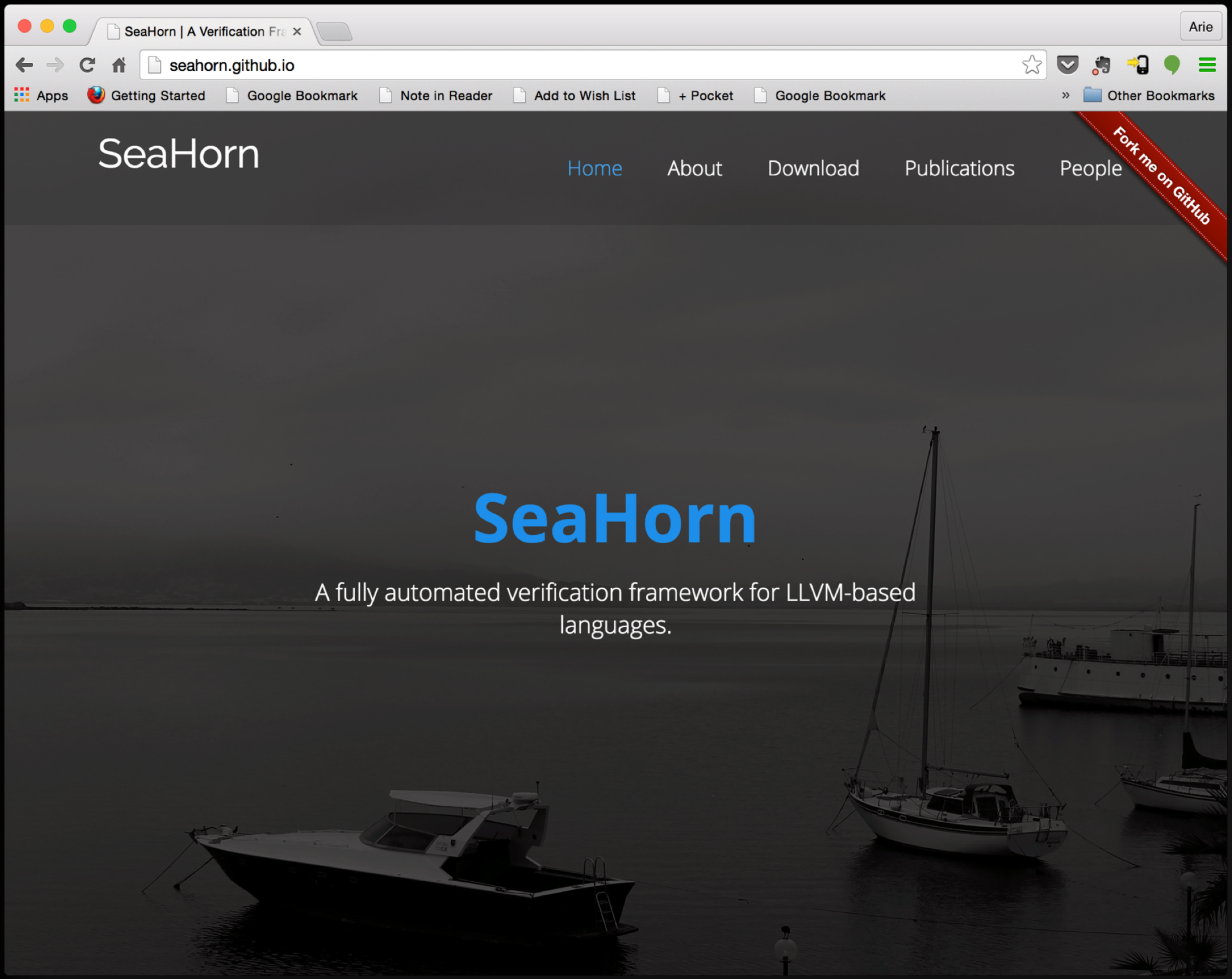
Lecture 1: Overview of SeaHorn and Algorithmic Logic-Based Verification

Lecture 2: Generating verification conditions for automated analysis

Lecture 3: IC3: Incremental Construction of Inductive Clauses for Indubitable Correctness

Lecture 4: Solving Constrained Horn Clauses over Linear Real Arithmetic

Extra slides: What about Machine Learning?



http://seahorn.github.io

SeaHorn Usage

Example: in test.c, check that x is always greater than or equal to y

test.c

```
extern int nd();
extern void __VERIFIER_error() __attribute__((noreturn));
void assert (int cond) { if (!cond) __VERIFIER_error (); }
int main(){
  int x,y;
  x=1; y=0;
  while (nd ())
  {
    x=x+y;
    y++;
  }
  assert (x>=y);
 return 0;
}
```

SeaHorn command:

```
)-> sea pf test.c
```

SeaHorn result:

```
        SEAHORN
----------------------------
PROPERTY (line 12) | TRUE
----------------------------
TIME(ms)           |  0.06
```

SeaHorn at a glance

Publicly Available (http://seahorn.github.io)
state-of-the-art Software Model Checker

Industrial-strength front-end based on Clang and LLVM

Abstract Interpretation engine: Crab

SMT-based verification engine: Spacer

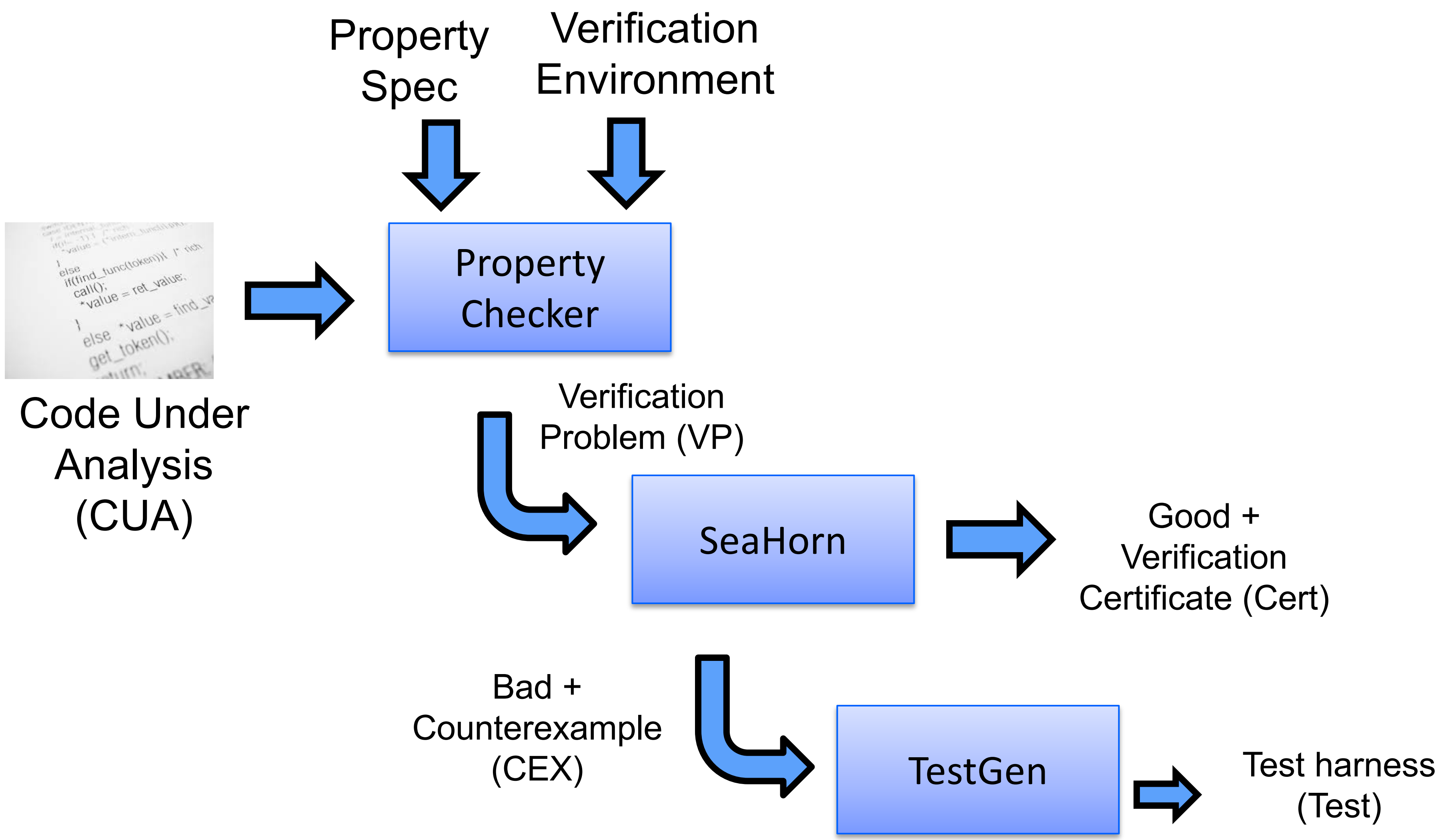
Bit-precise Bounded Model Checker and Symbolic Execution

Executable Counter-Examples

A framework for research and application of logic-based verification

UNIVERSITY OF WATERLOO

SeaHorn Workflow



SeaHorn workflow components

Code Under Analysis (CUA)

- code being analyzed. Device driver, component, library, etc.

Verification Environment

- stubs for the environment with which CUA interacts
- e.g., libc, memcpy, malloc, OS system calls, user input, socket, file, …

Property Checker

- static instrumentation of a program with a monitor that indicates when an error has happened
- similar to dynamic sanitizers, but can use verifier-specific API to perform symbolic actions
- property spec is specific to a property checker

Verification Problem

- a prepared instance of program with embedded assertions, potentially simplified by abstracting away irrelevant parts of execution

Test Gen

- generates a test harness that includes all stubs and stimuli to guide CUA to a property failure discovered by the verifier

Developing a Static Property Checker

A static property checker is similar to a dynamic checker

- e.g., clang sanitizer (address, thread, memory, etc.)

A significant development effort for each new property

- new specialized static analyses to rule out trivial cases
- different instrumentations have affect on performance

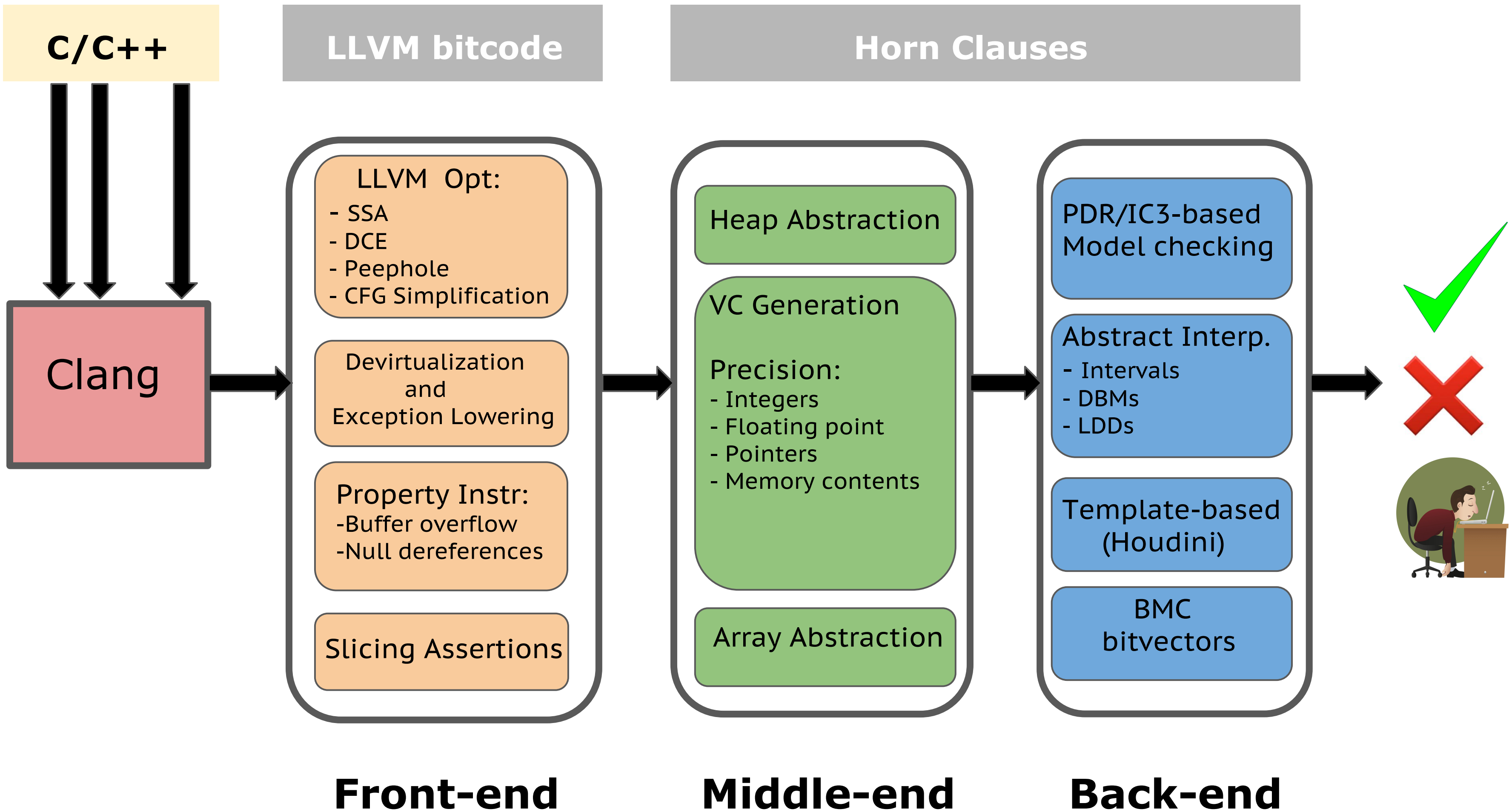
Developed by a domain expert

- understanding of verification techniques is useful (but not required)
- 3-6 month effort for a new property
  - but many things can be reused between similar properties
  - e.g., memory safety, null-dereference, taint checking, use-after-free, etc.

SeaHorn property checkers:

- memory safety (out of bound uses, null pointer)
  - ongoing work to improve scalability and usability
- taint analysis (being developed by Princeton, see CAV 2018)

Architecture of Seahorn



DEMO

UNIVERSITY OF WATERLOO

SeaHorn Memory Model

Block-based memory model

- each allocation (malloc/alloca/etc) creates a new object
- a pointer is a pair (id,off), called cell, where id is an object identifier and off is a positive numeric offset
- similar to the C memory model

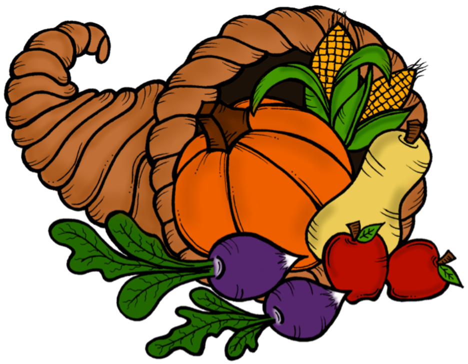
Abstract Memory Model

- the number of allocation regions is finite
- allocation site is used as an object identifier
- custom pointer-analysis is used to approximate abstract points to graph

Pointer Analysis: Sea-DSA (SAS 2017)

- unification-based (like LLVM-DSA)
- context-, field-, and array-sensitive

Crab Abstract Interpretation Library



Crab – Cornucopia of Abstract Domains

- Numerical domains (intervals, zones, boxes)
- 3rd party domains (apron, elina)
- arrays, uninterpreted functions, null, pointer

Language independent core with plugins for LLVM bitcode

- fixed-point engine
- widening / narrowing strategies
- crab-llvm : integrates LLVM optimizations and analysis of LLVM bitcode

Support for inter-procedural analysis

- pre-, post-conditions, function summaries

Extensible, publicly available on GitHub, open C++ API

UNIVERSITY OF WATERLOO

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
  - extend IC3/PDR algorithms from Hardware Model Checking

Symbolic Reachability Problem

P = (X, Init, Tr, Bad)

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

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

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

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

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

UNIVERSITY OF WATERLOO

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, …, p_n, h are n-ary predicates
- $p_i[X]$ is an application of a predicate to first-order terms

UNIVERSITY OF WATERLOO

CHC Satisfiability

A $\mathcal{T}$-**model** of a set of a 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

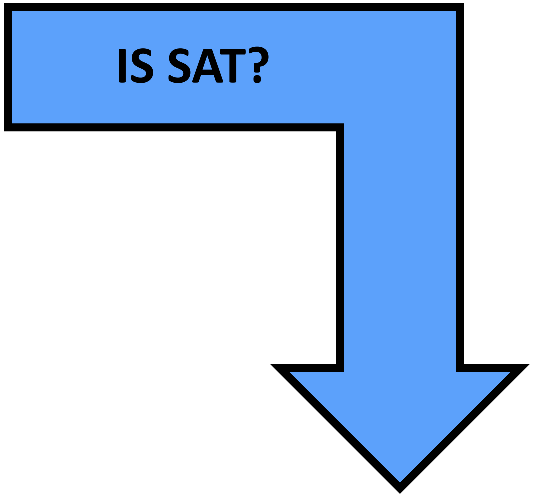
A $\boldsymbol{\mathcal{T}}$**-solution** of a set of CHCs Π is a substitution σ from predicates p_i to $\mathcal{T}$-formulas such that $\Pi\sigma$ is $\mathcal{T}$-valid

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

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                          →  Inv(x, y, z, i)
Inv(x, y, z, i) & i < y & z1=z+1 & i1=i+1      →  Inv(x, y, z1, i1)
Inv(x, y, z, i) & i >= y & z != x+y            →  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)))
)
```

```
Inv(x, y, z, i)

    z  = x + i

    z <= x + y
```

Horn Clauses for Program Verification

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

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

```
5. incorrect :- Z=W+1, W≥0, W+1<
                read(A,W,U), read(A,Z
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]:

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

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

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

Bjørner, Gurfinkel, McMillan, and Rybalchenko:

Horn Clause Solvers for Program Verification

UNIVERSITY OF WATERLOO

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' ,

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

multi-threaded program P is safe

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

$$\left\{ R(\mathsf{g}, \mathsf{p}_{\sigma(1)}, \mathsf{l}_{\sigma(1)}, \dots, \mathsf{p}_{\sigma(k)}, \mathsf{l}_{\sigma(k)}) \leftarrow dist(\mathsf{p}_1, \dots, \mathsf{p}_k) \wedge R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \dots, \mathsf{p}_k, \mathsf{l}_k) \right\}_{\sigma \in S_k} \tag{6}$$

$$R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \dots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_1, \dots, \mathsf{p}_k) \wedge Init(\mathsf{g}, \mathsf{l}_1) \wedge \dots \wedge Init(\mathsf{g}, \mathsf{l}_k) \tag{7}$$

$$R(\mathsf{g}', \mathsf{p}_1, \mathsf{l}'_1, \dots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_1, \dots, \mathsf{p}_k) \wedge ((\mathsf{g}, \mathsf{l}_1) \xrightarrow{\mathsf{p}_1} (\mathsf{g}', \mathsf{l}'_1)) \wedge R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \dots, \mathsf{p}_k, \mathsf{l}_k) \tag{8}$$

$$R(\mathsf{g}', \mathsf{p}_1, \mathsf{l}_1, \dots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_0, \mathsf{p}_1, \dots, \mathsf{p}_k) \wedge ((\mathsf{g}, \mathsf{l}_0) \xrightarrow{\mathsf{p}_0} (\mathsf{g}', \mathsf{l}'_0)) \wedge RConj(0, \dots, k) \tag{9}$$

$$false \leftarrow dist(\mathsf{p}_1, \dots, \mathsf{p}_r) \wedge \Big(\bigwedge_{j=1,\dots,m} (\mathsf{p}_j = p_j \wedge (\mathsf{g}, \mathsf{l}_j) \in E_j) \Big) \wedge RConj(1, \dots, r) \tag{10}$$

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

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

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

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

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

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

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

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

(initial) $\mathsf{init}(g, x_1) \wedge \dots \wedge \mathsf{init}(g, x_n) \rightarrow Inv(g, \ell_{\mathsf{init}}, x_1, \dots, \ell_{\mathsf{init}}, x_k)$

(inductive) $Inv(g, \ell_1, x_1, \dots, \ell_i, x_i, \dots, \ell_k, x_k) \wedge s(g, x_i, g', x'_i) \rightarrow Inv(g', \ell_1, x_1, \dots, \ell'_i, x'_i, \dots, \ell_k,$

(non-interference) $Inv(g, \ell_1, x_1, \dots, \ell_k, x_k) \wedge$
$Inv(g, \ell^\dagger, x^\dagger, \ell_2, x_2, \dots, \ell_k, x_k) \wedge$
$\vdots$
$Inv(g, \ell_1, x_1, \dots, \ell_{k-1}, x_{k-1}, \ell^\dagger, x^\dagger) \wedge s(g, x^\dagger, g', \cdot) \rightarrow Inv(g', \ell_1, x_1, \dots, \ell_k, x_k)$

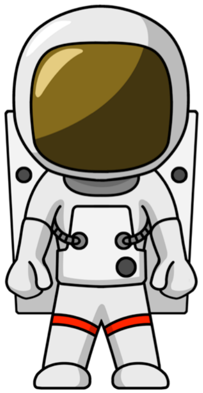
(safe) $Inv(g, \ell_1, x_1, \dots, \ell_k, x_k) \wedge \mathsf{err}(g, \ell_1, x_1, \dots, \ell_m, x_m) \rightarrow false$

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 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

UNIVERSITY OF WATERLOO

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

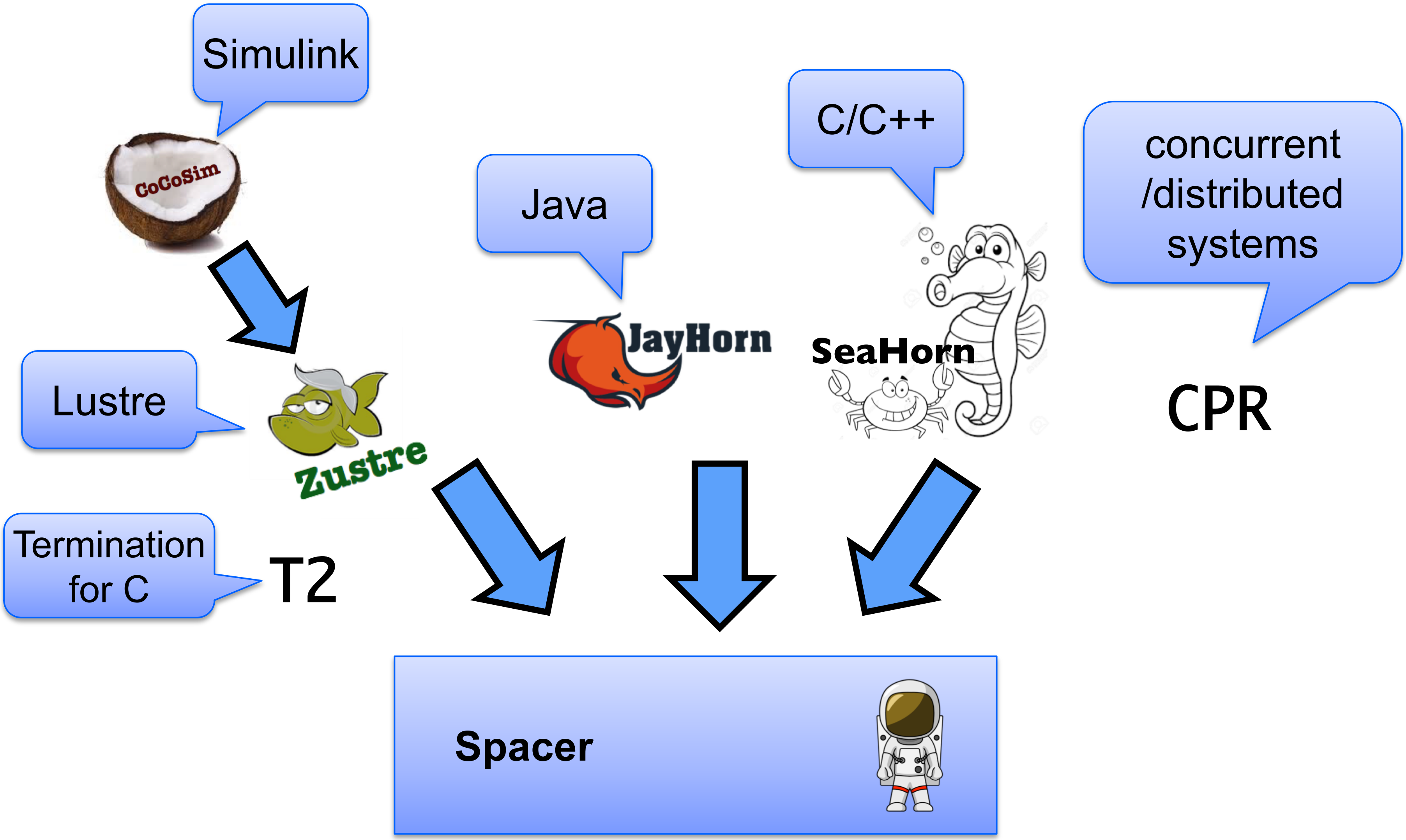
Supported SMT-Theories

- 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.

Logic-based Algorithmic Verification



UNIVERSITY OF WATERLOO

VERIFICATION CONDITIONS FOR PROGRAMS

UNIVERSITY OF WATERLOO

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

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 ⇒ **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

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

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)

UNIVERSITY OF WATERLOO

Example of a WLP Horn Encoding

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

ToHorn

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

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

$$\texttt{C3: } \mathit{false} \leftarrow I(x,y,x_o,y_o),\ y<=0,\ x \neq x_o+y_o$$

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

Control Flow Graph

A CFG is a graph of basic blocks

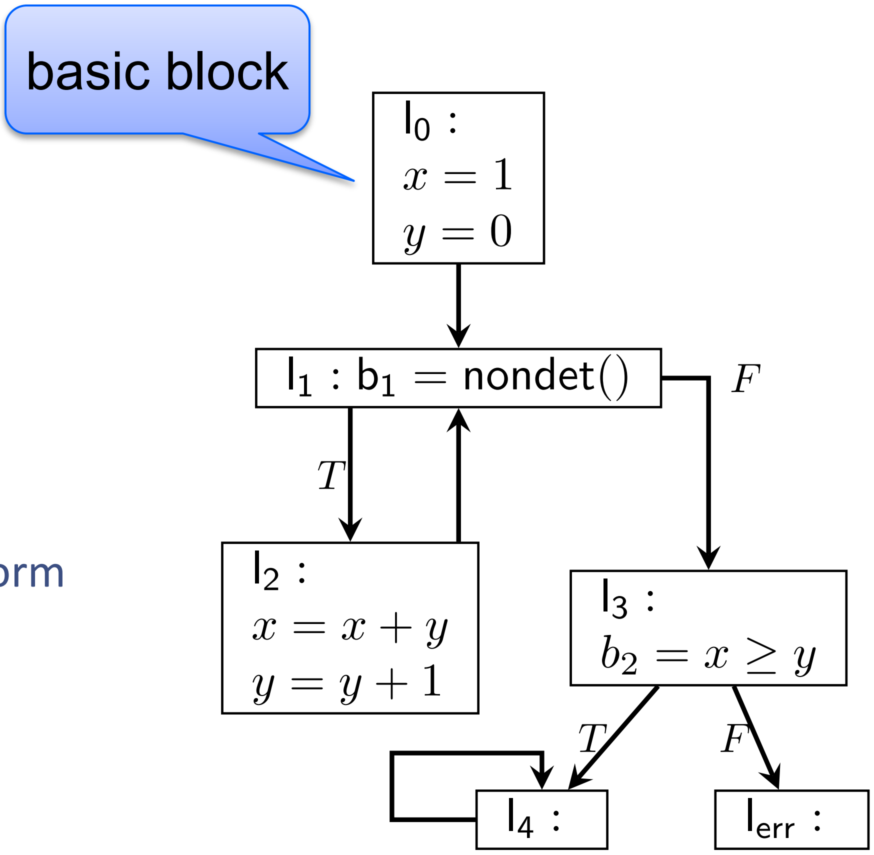
- edges represent different control flow

A CFG corresponds to a program syntax

- where statements are restricted to the form

$$\texttt{L}_\texttt{i}\texttt{:S ; goto L}_\texttt{j}$$

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



Dual WLP

Dual weakest liberal pre-condition

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

s ∈ **dual-wlp** (P, Post) IFF there exists an execution of P that starts in s and ends in Post

dual-wlp (P, Post) is the weakest condition ensuring that an execution of P can reach a state in Post

Examples of dual-wlp

dual-wlp(**assume**(E), Q) = ¬wlp(**assume**(E), ¬ Q) = ¬(E ⇒ ¬ Q) = E ∧ Q

dual-wlp(x := x+y; y := y+1, x=x’ ∧ y=y’) = y+1=y’ ∧ x+y=x’

wlp(x := x + y, ¬(y+1=y ∧ x=x'))

= let x = x+y in ¬ (y+1=y' ∧ x=x')

= ¬ (y+1=y' ∧ x+y=x')

wlp(y:=y+1, ¬(x=x' ∧ y=y'))

= let y = y+1 in ¬(y=y' ∧ x=x')

= ¬ (y+1=y ∧ x=x')

Horn Clauses by Dual WLP

Assumptions

- each procedure is represent by a control flow graph
  - i.e., statements of the form `l`$_i$`:S ; goto l`$_j$, where S is loop-free
- program is unsafe iff the last statement of `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)
  - $p_{en}(x_0, x)$ for entry location of procedure p()
  - $p_{ex}(x_0, r)$ 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:

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

$l_j(x_0,w') \leftarrow l_i(x_0,w) \land \neg wlp\ (S, \neg(w=w'))$

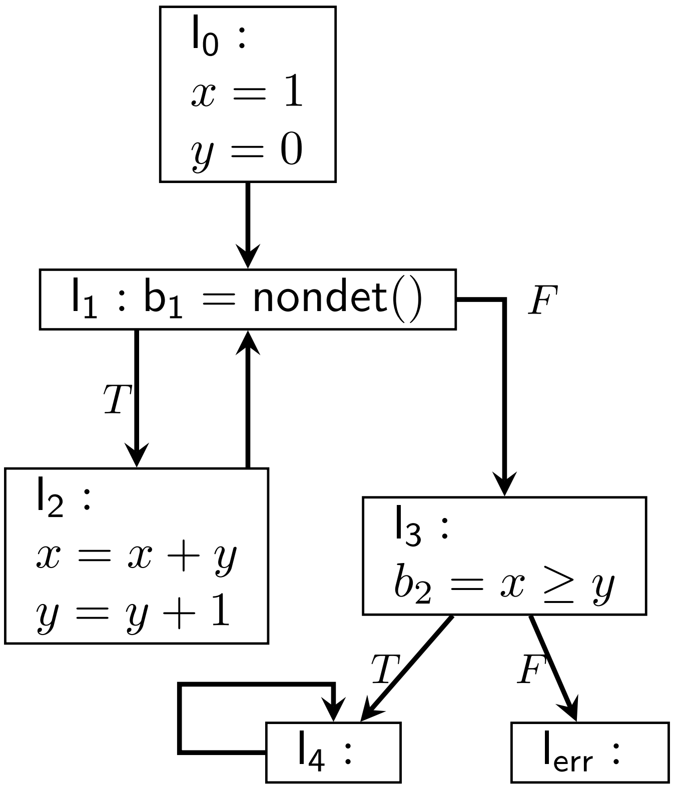
- for each statement $\mathtt{l_i\text{: } S\text{; goto } l_j}$

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

$false \leftarrow Main_{ex}(x, ret)$

Example Horn Encoding

$\text{int } x = 1;$
$\text{int } y = 0;$
$\text{while } (*) \ \{$
$\quad x = x + y;$
$\quad y = y + 1;$
$\}$
$\text{assert}(x \geq y);$



$\langle 1 \rangle$ p_0.
$\langle 2 \rangle$ $\mathsf{p}_1(x, y) \leftarrow \mathsf{p}_0, x = 1, y = 0.$
$\langle 3 \rangle$ $\mathsf{p}_2(x, y) \leftarrow \mathsf{p}_1(x, y)$.
$\langle 4 \rangle$ $\mathsf{p}_3(x, y) \leftarrow \mathsf{p}_1(x, y)$.
$\langle 5 \rangle$ $\mathsf{p}_1(x', y') \leftarrow \mathsf{p}_2(x, y), x' = x + y, y' = y + 1.$
$\langle 6 \rangle$ $\mathsf{p}_4 \leftarrow (x \geq y), \mathsf{p}_3(x, y).$
$\langle 7 \rangle$ $\mathsf{p}_{err} \leftarrow (x < y), \mathsf{p}_3(x, y).$
$\langle 8 \rangle$ $\mathsf{p}_4 \leftarrow \mathsf{p}_4.$
$\langle 9 \rangle$ $\bot \leftarrow \mathsf{p}_{err}.$

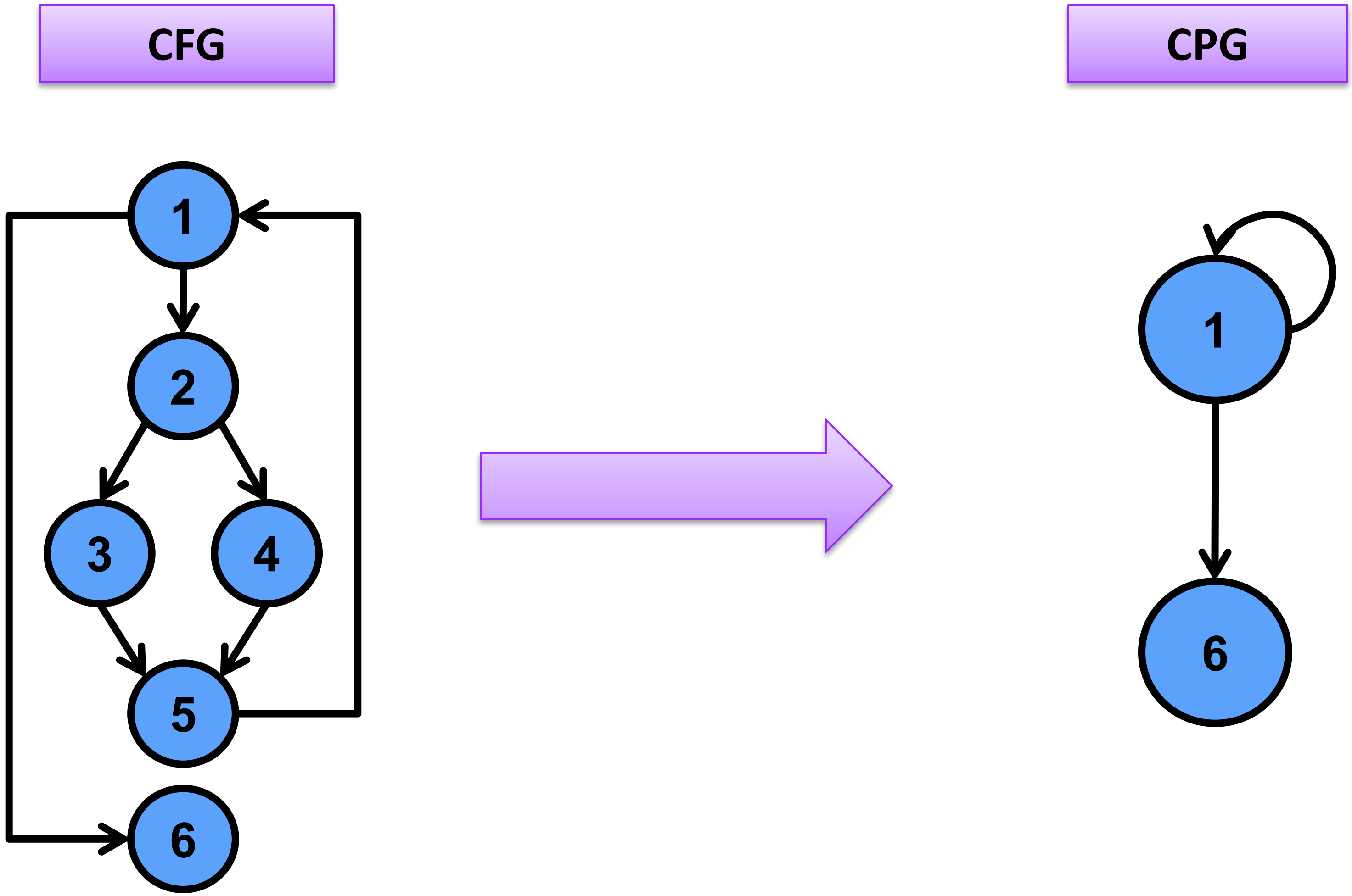
From CFG to Cut Point Graph

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



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 = 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

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:
```

UNIVERSITY OF WATERLOO

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)

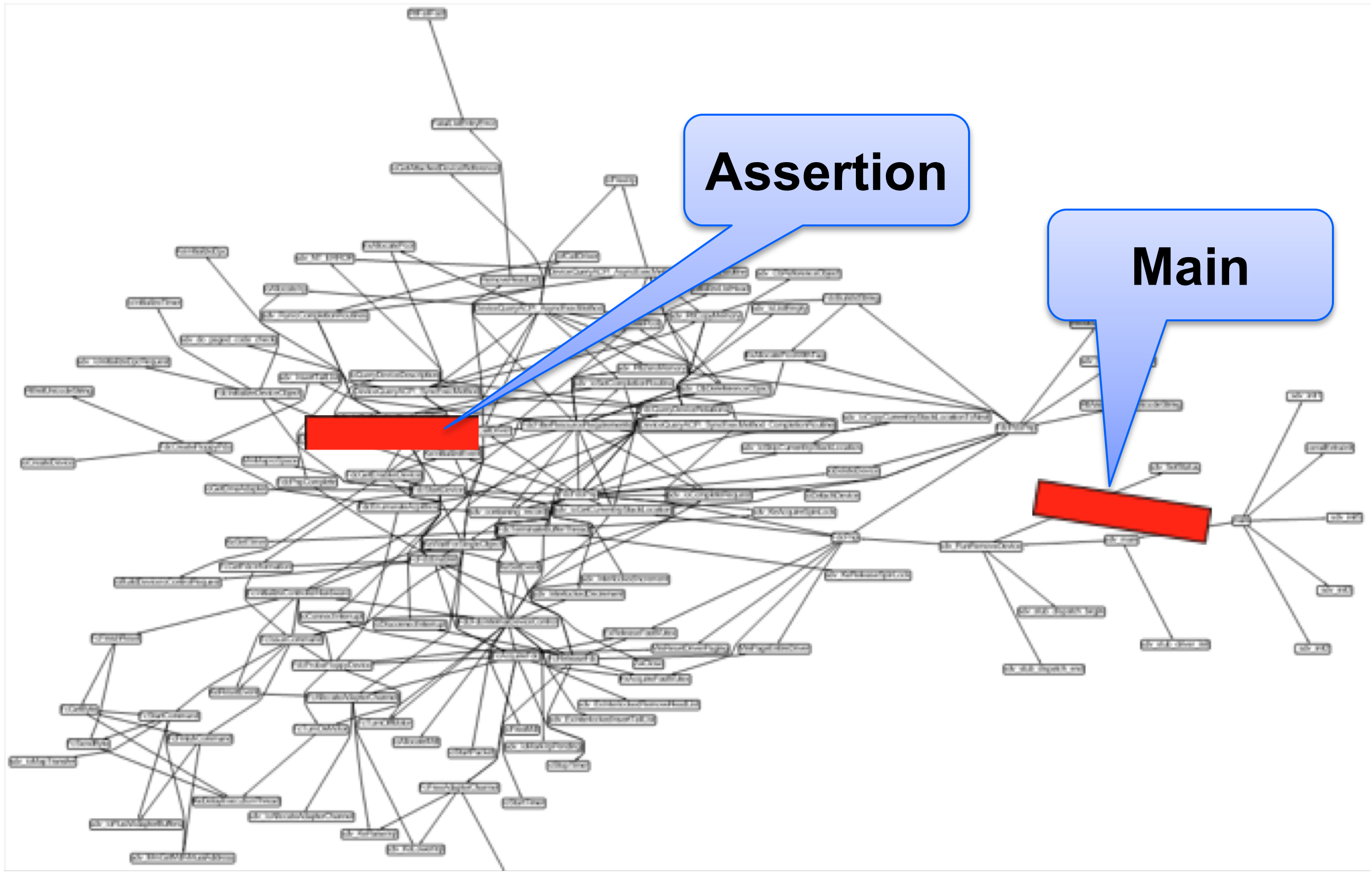
For each edge (s, t) in CPG use dual-wlp to construct the constraint for an execution to flow from s to t

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

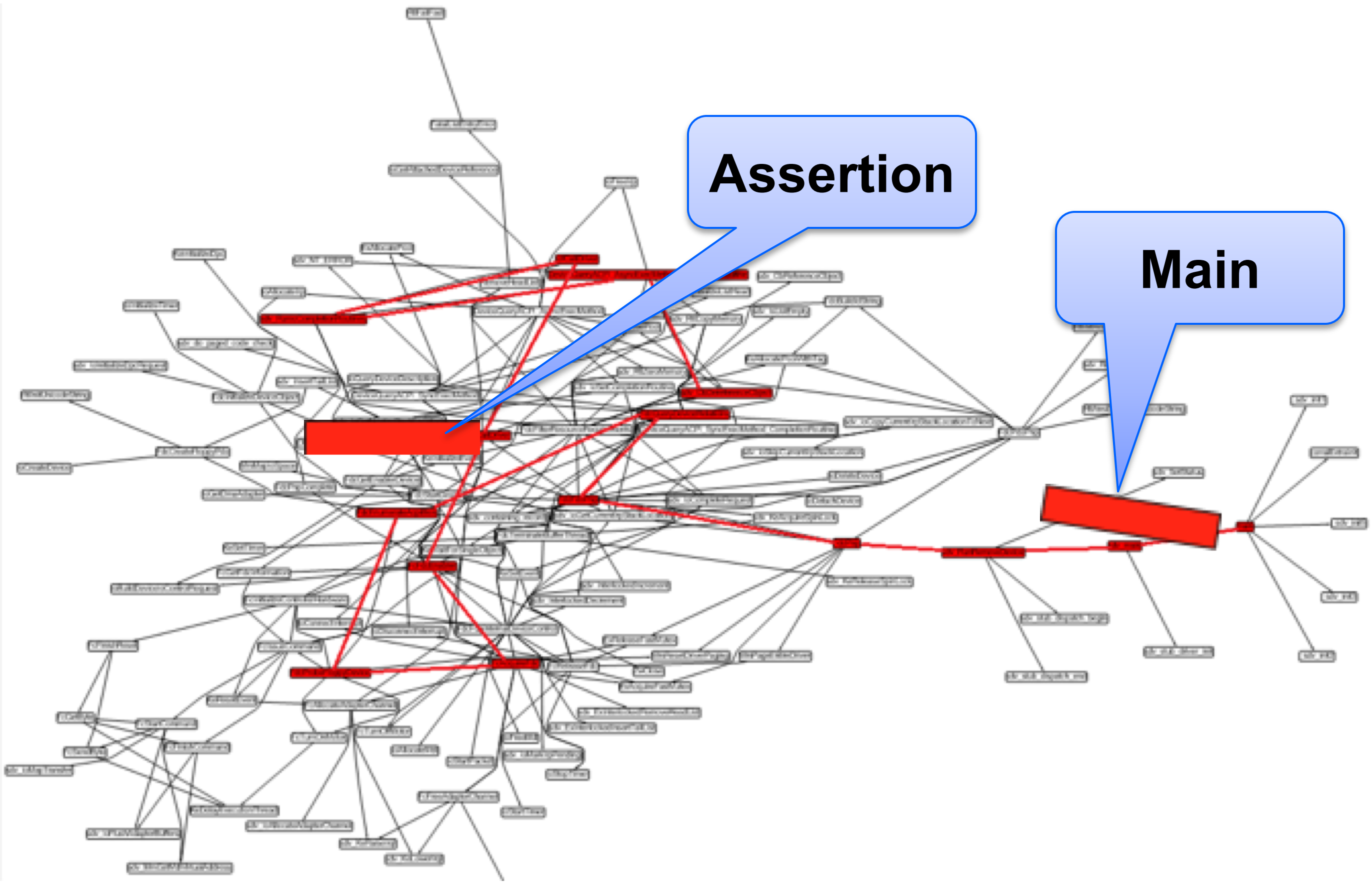
Mixed Semantics

PROGRAM TRANSFORMATION

Deeply nested assertions



Deeply nested assertions



Counter-examples are long

Hard to determine (from main) what is relevant

Mixed Semantics

[GWC'08,LQ'14]

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

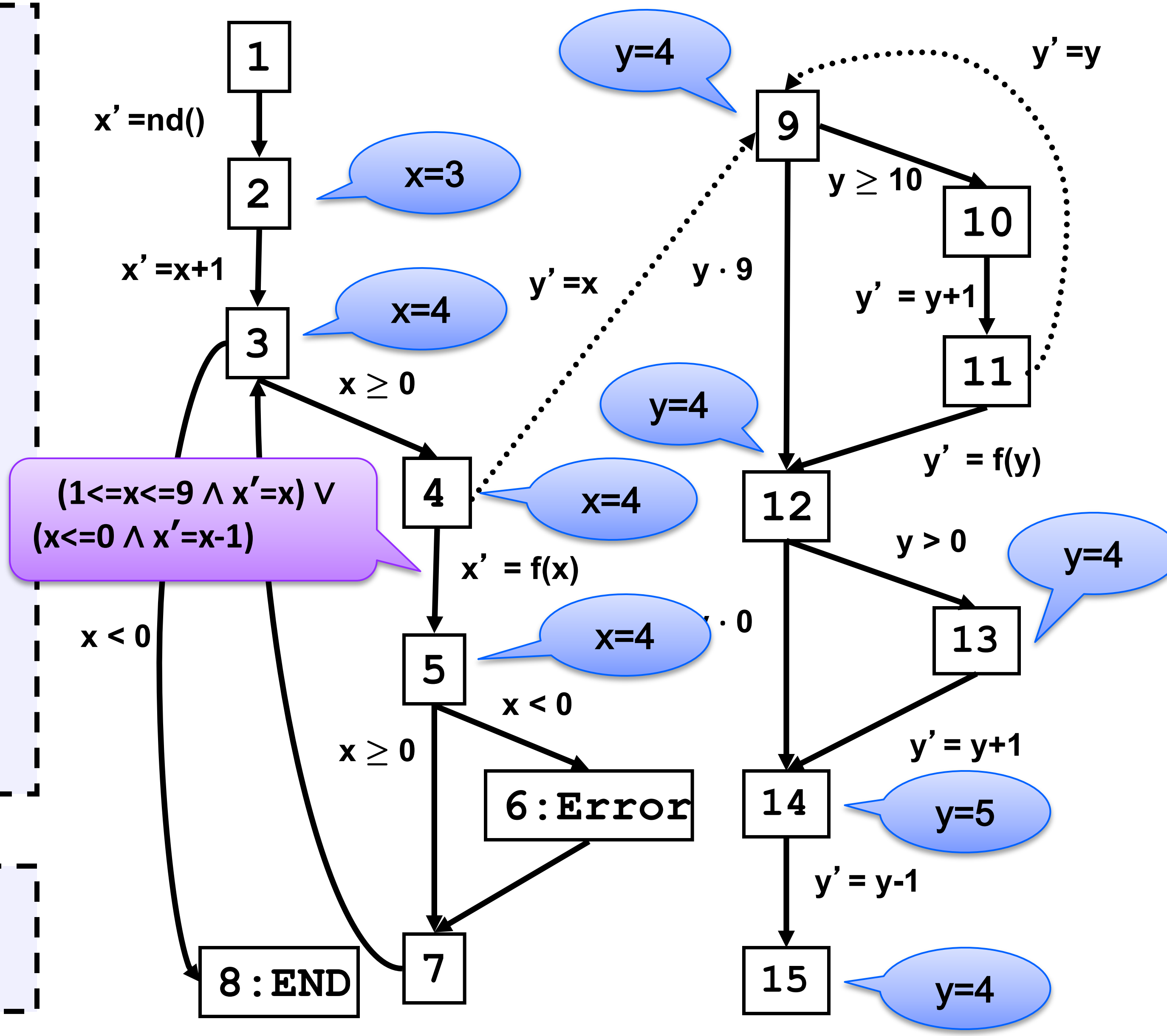
Theorem: Let K be the operational semantics, K^m the stack-free semantics, and L a program location. Then,

$K \models EF\,(pc=L) \Leftrightarrow K^m \models EF\,(pc=L)$ and $K \models EG\,(pc \neq L) \Leftrightarrow K^m \models EG\,(pc \neq L)$

```
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:
```

Summary of f(y)

$(1<=y<=9 \wedge y'=y) \vee$
$(y<=0 \wedge y'=y-1)$



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 errror() 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

Incremental Construction of Inductive Clauses for Indubitable Correctness

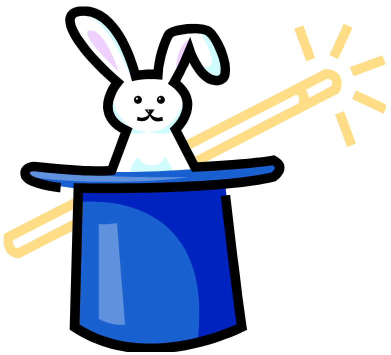
IC3

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

UNIVERSITY OF WATERLOO

Symbolic Reachability Problem

P = (X, Init, Tr, Bad)

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

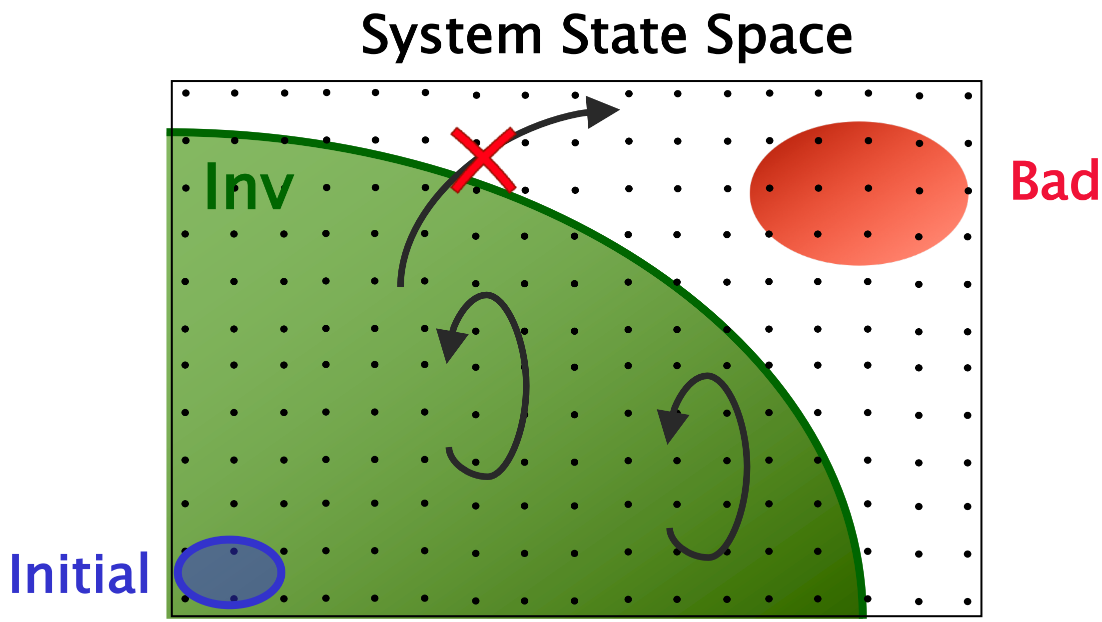
$$Init(X_0) \wedge \left(\bigwedge_{i=0}^{N-1} Tr(X_i, X_{i+1}) \right) \wedge Bad(X_N) \;\not\Rightarrow\; \bot$$

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

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

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

Inductive Invariants

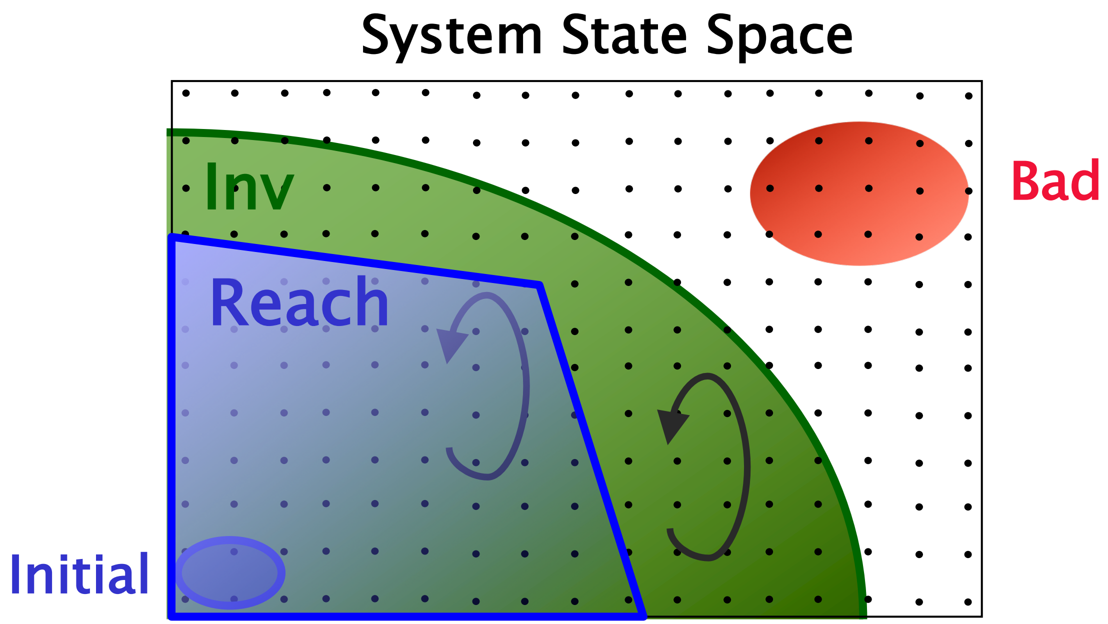


System S is safe iff there exists an inductive invariant Inv:

- **Initiation:** Initial $\subseteq$ Inv
- **Safety:** Inv $\cap$ Bad $= \emptyset$
- **Consecution:** TR(Inv) $\subseteq$ Inv

i.e., if $s \in$ Inv and $s \rightsquigarrow t$ then $t \in$ Inv

Inductive Invariants



System S is safe iff there exists an inductive invariant Inv:

- **Initiation:** Initial ⊆ Inv
- **Safety:** Inv ∩ Bad = ∅
- **Consecution:** TR(Inv) ⊆ Inv

i.e., if s ∈ Inv and s⤳t then t ∈ Inv

System S is safe if Reach ∩ Bad = ∅

UNIVERSITY OF WATERLOO

IC3

(Bradley, VMCAI 2011)

IC3 = **I**ncremental **C**onstruction of **I**nductive **C**lauses for **I**ndubitable **C**orrectness

The Goal: Find an Inductive Invariant stronger than P

- Recall: F is an inductive invariant stronger than P if
  - INIT => F
  - $F \wedge T$ => F'
  - F => P

by learning relatively inductive facts (incrementally)

In a property directed manner

- Also called “Property Directed Reachability” (PDR)

IC3 Basics

Iteratively compute Over-Approximated Reachability Sequence (OARS) $<F_0,F_1,\ldots,F_{k+1}>$ s.t.

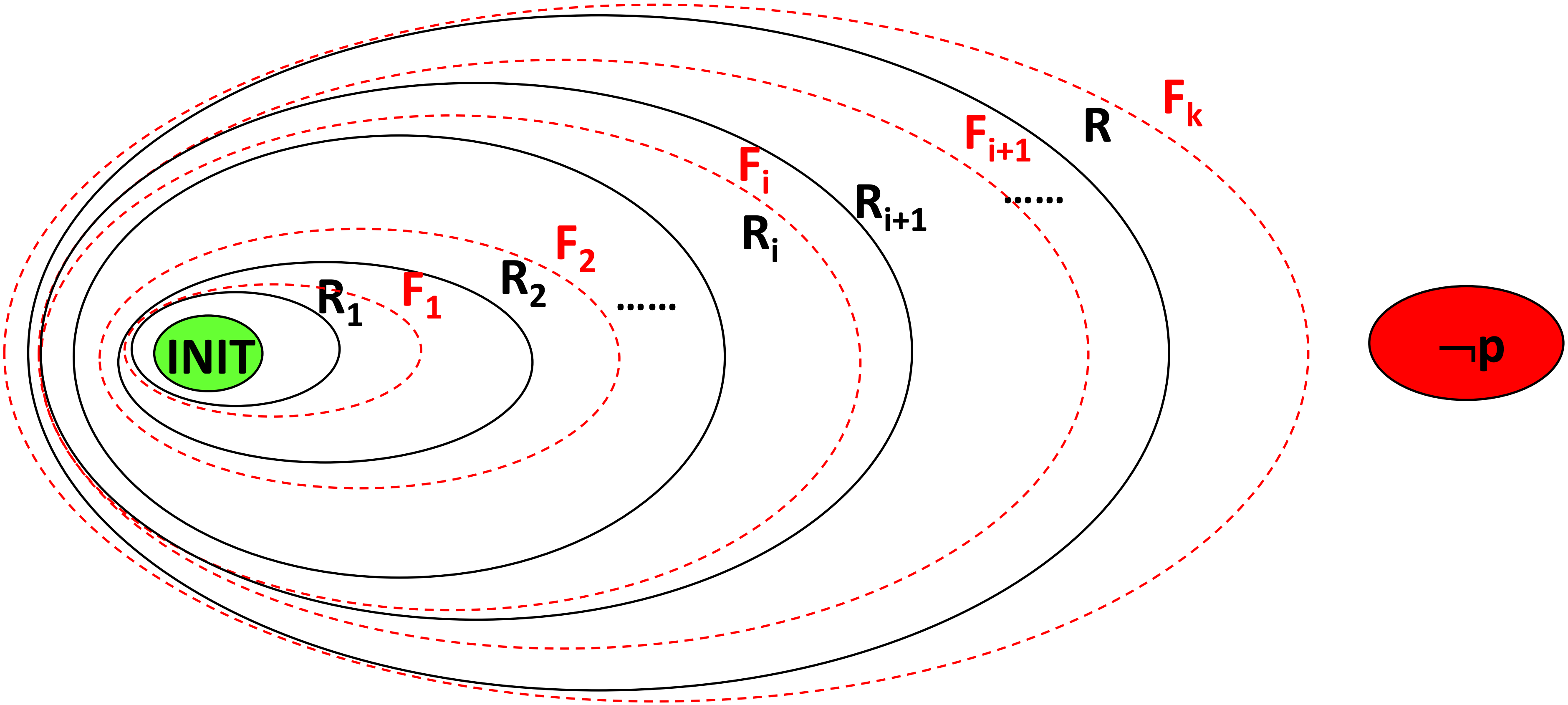
- F_0 = INIT
- $F_i \Rightarrow F_{i+1}$ monotone: $F_i \subseteq F_{i+1}$
- $F_i \wedge T \Rightarrow F'_{i+1}$ inductive: simulates one forward step
- $F_i \Rightarrow P$ safe: p is an invariant up to k+1

F_i - CNF formula given as a set of clauses

F_i over-approximates R_i

- If $F_{i+1} \Rightarrow F_i$ then fixpoint: F_i is an inductive invariant

OARS (aka Inductive Trace)



$$F_{i+1}(V') \Leftarrow F_i(V) \wedge T(V,V')$$

If $F_{k+1} \equiv F_k$ then F_k is an inductive invariant

IC3 Basics (cont.)

c is inductive relative to F if

- $INIT \Rightarrow c$
- $F \wedge c \wedge T \Rightarrow c'$

Notation:

- cube s: conjunction of literals
  - $\neg v_1 \wedge v_2 \wedge \neg v_3$ - Represents a state
- s is a cube => **¬s is a clause** (DeMorgan)

IC3 - Initialization

OARS:

- $F_0 = INIT$
- $F_i \Rightarrow F_{i+1}$
- $F_i \wedge T \Rightarrow F'_{i+1}$
- $F_i \Rightarrow P$

Check satisfiability of the two formulas:

- $INIT \wedge \neg P$
- $INIT \wedge T \wedge \neg P'$

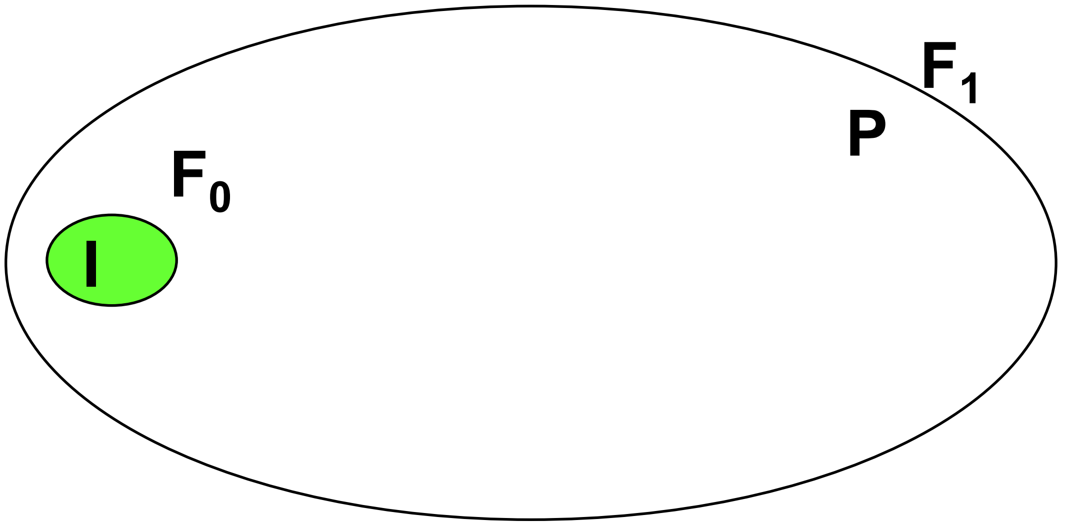
If at least one is satisfiable: cex found

If both are unsatisfiable then:

- $INIT \Rightarrow P$
- $INIT \wedge T \Rightarrow P'$

Therefore

- $F_0 = INIT, F_1 = P$
  - $\langle F_0, F_1 \rangle$ is an OARS



IC3 - Iteration

OARS:

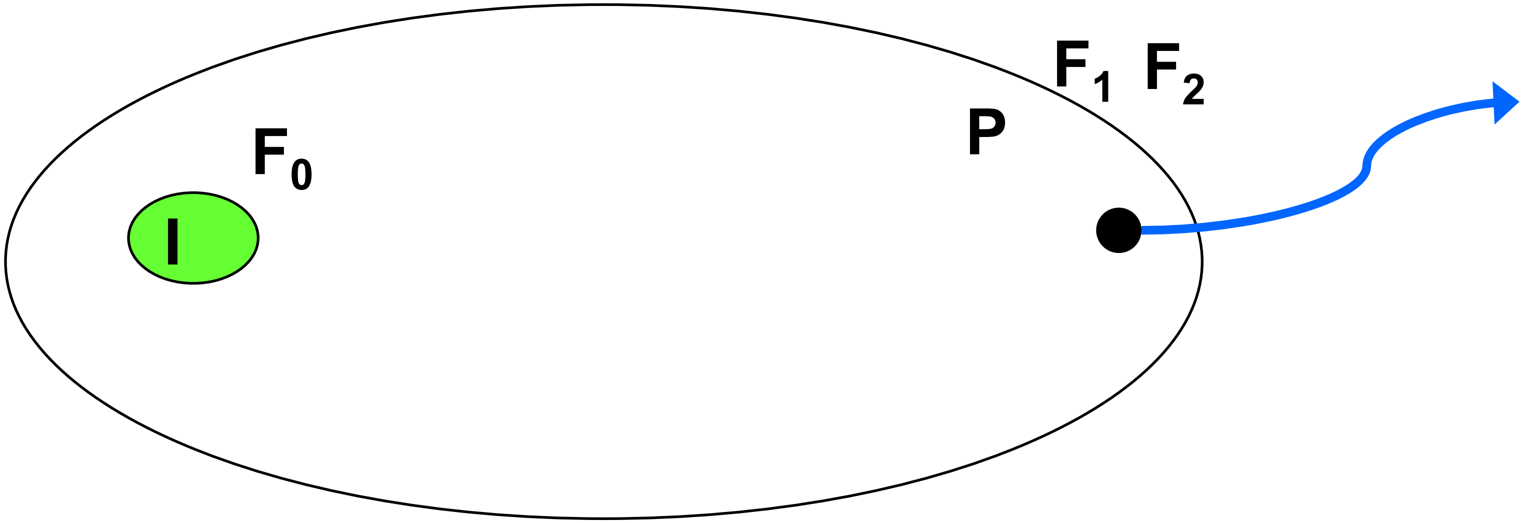
- $F_0 = INIT$
- $F_i \Rightarrow F_{i+1}$
- $F_i \wedge T \Rightarrow F'_{i+1}$
- $F_i \Rightarrow P$

Our OARS contains F_0 and F_1

Initialize F_2 to P

- If P is an inductive invariant – done! ☺
- Otherwise: $F_1 \wedge T \not\Rightarrow F'_2$

  => F_1 should be strengthened



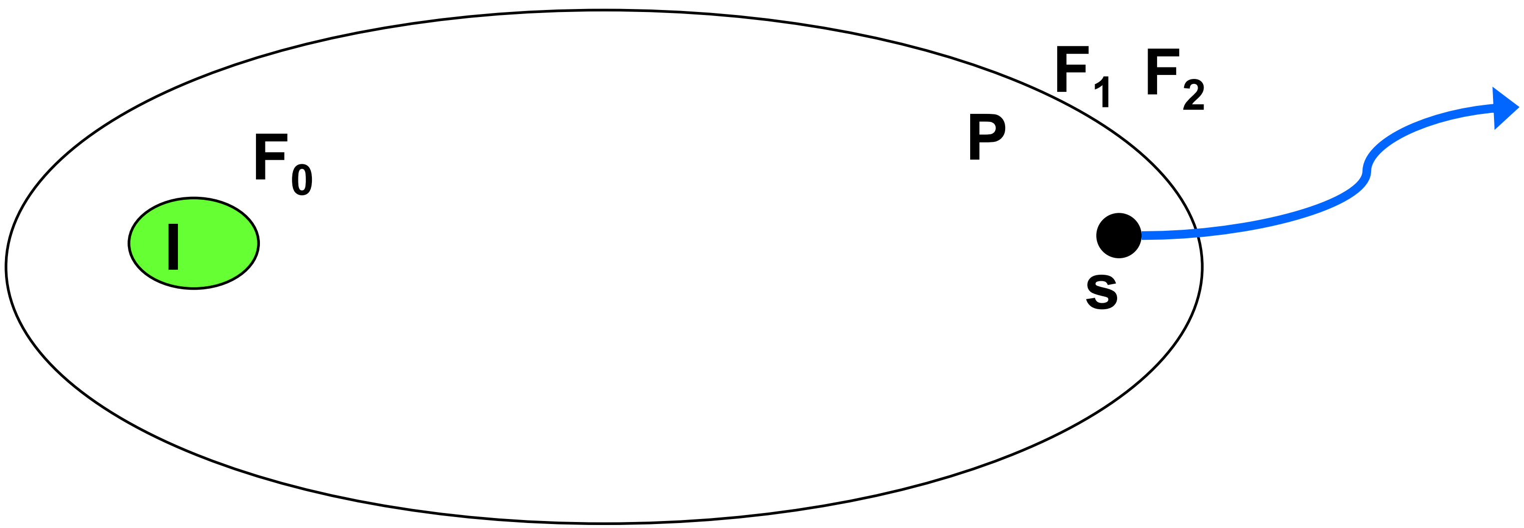
IC3 - Iteration

OARS:

- F_0 = INIT
- $F_i \Rightarrow F_{i+1}$
- $F_i \wedge T \Rightarrow F'_{i+1}$
- $F_i \Rightarrow P$

If P is not an inductive invariant

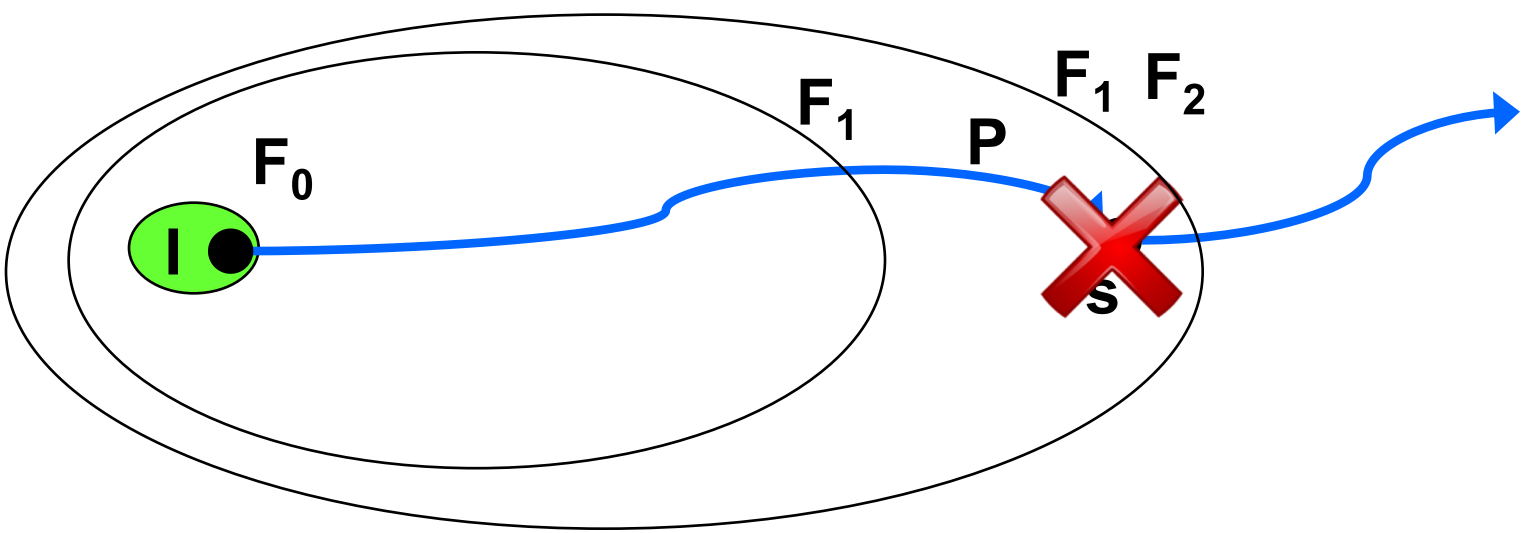
- $F_1 \wedge T \wedge \neg P'$ is satisfiable
  - $(F \wedge T \wedge \neg P')$ sat IFF $(F \wedge T \Rightarrow P')$ not valid
- From the satisfying assignment get a state s that can reach a bad state



IC3 - Iteration

Is s reachable in one transition from the previous set?

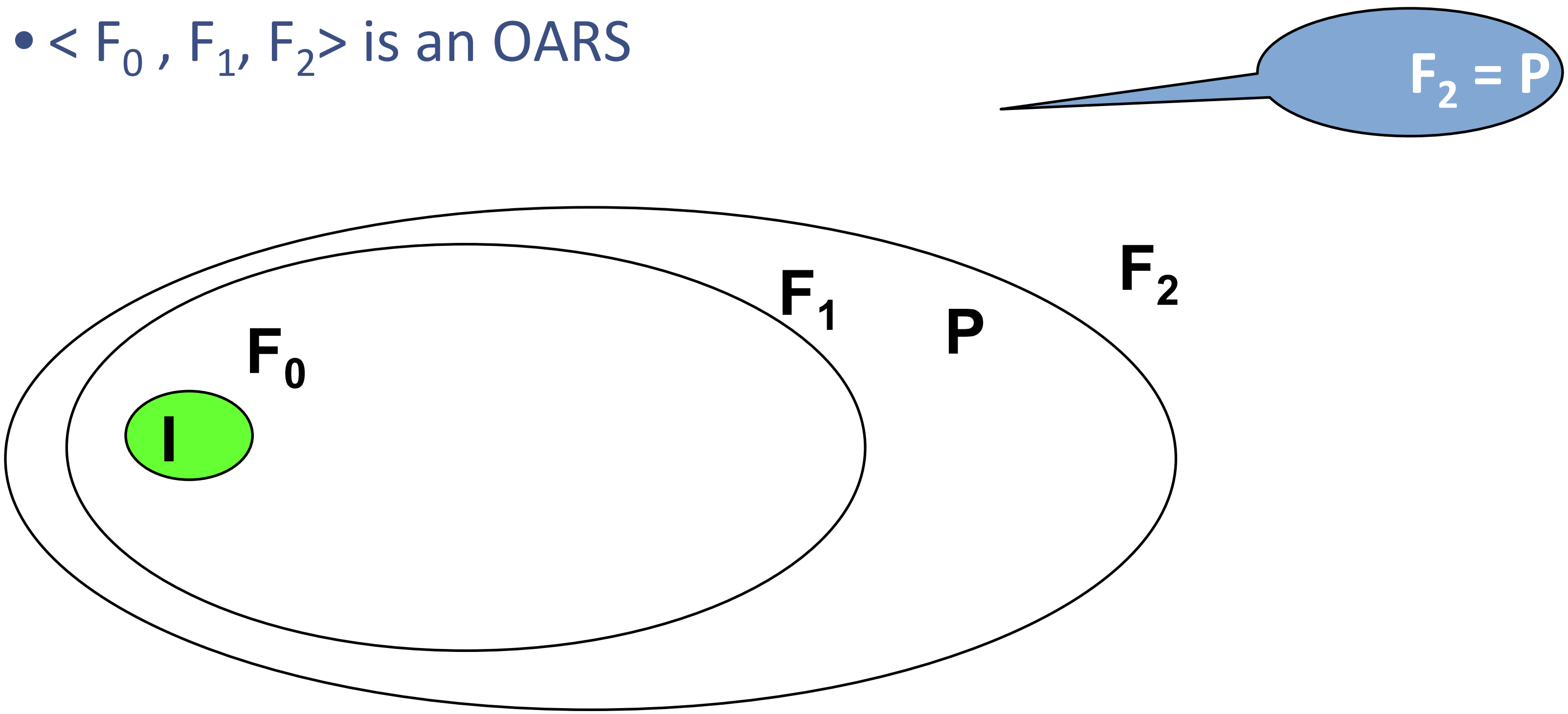
- backward search: Check $F_0 \wedge T \wedge s'$
- If satisfiable, s is reachable from F_0 : CEX
- Otherwise, block s, i.e. remove it from F_1
  - $F_1 = F_1 \wedge \neg s$



IC3 - Iteration

Iterate this process until $F_1 \wedge T \wedge \neg P'$ becomes unsatisfiable

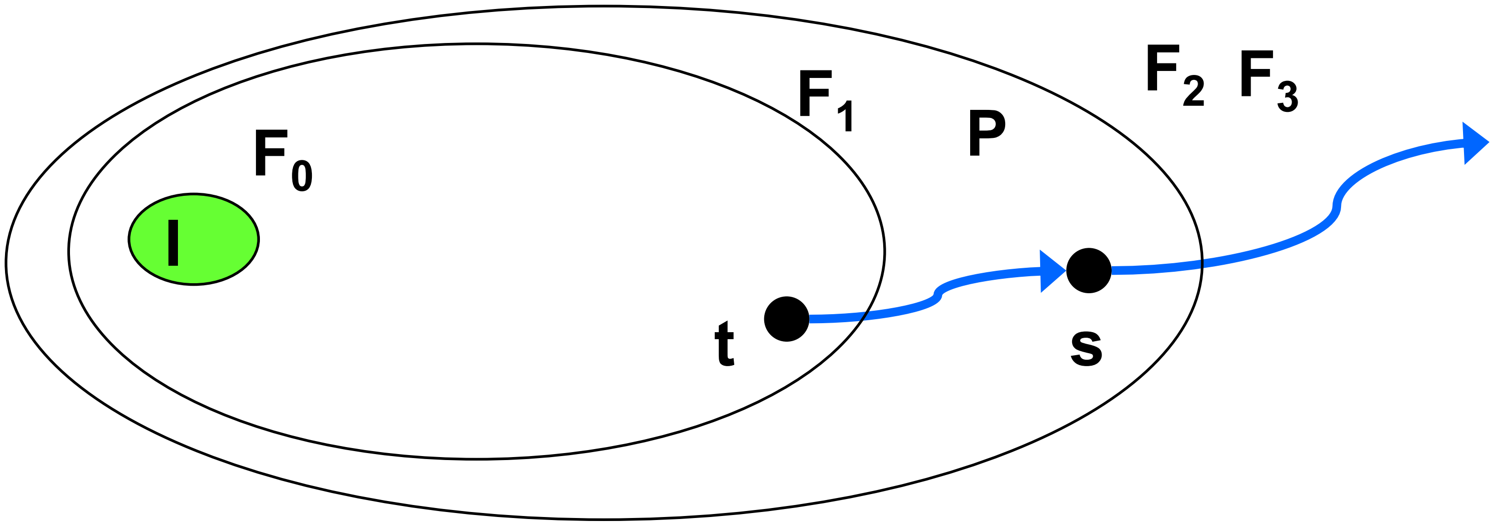
- $F_1 \wedge T \Rightarrow P'$ holds
- $< F_0 , F_1, F_2 >$ is an OARS



IC3 - Iteration

New iteration, initialize F_3 to P, check $F_2 \wedge T \wedge \neg P'$

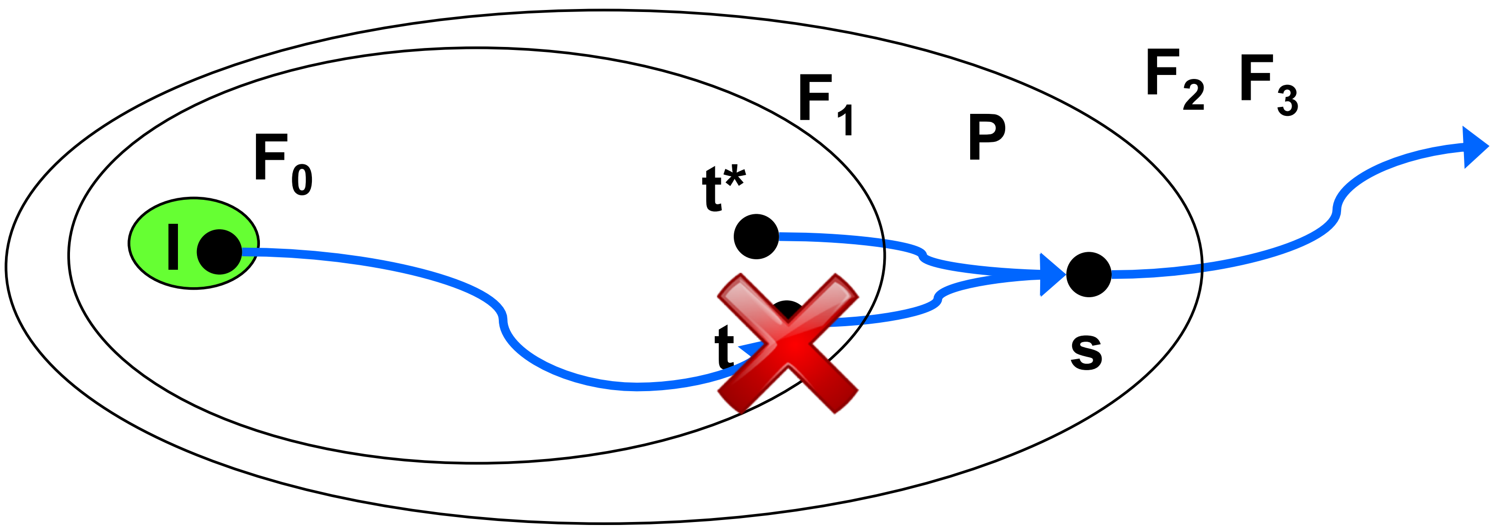
- If satisfiable, get s that can reach ¬P
- Now check if s can be reached from F_1 by $F_1 \wedge T \wedge s'$
- **If it can be reached, get t and try to block it**



IC3 - Iteration

To block t, check $F_0 \wedge T \wedge t'$

- If satisfiable, a CEX
- If not, t is blocked, get a "new" t* by $F1 \wedge T \wedge s'$ and try to block t*

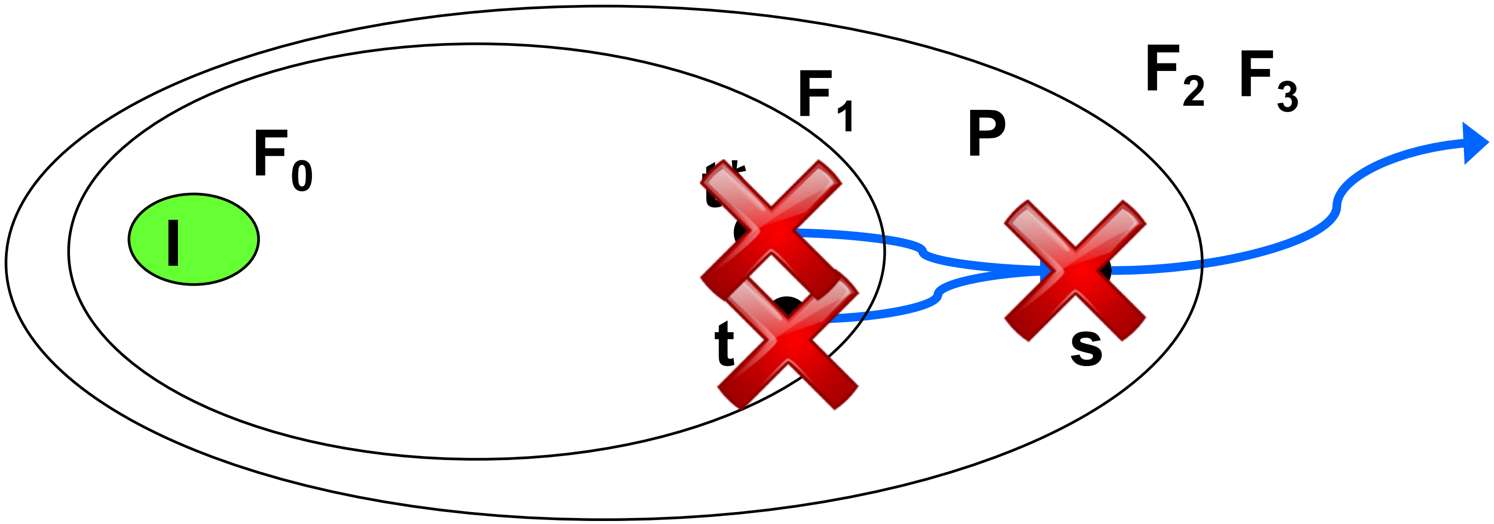


IC3 - Iteration

When $F_1 \wedge T \wedge s'$ becomes unsatisfiable

- s is blocked, get a "new" s* by $F_2 \wedge T \wedge \neg P'$ and try to block s*

……You get the picture ☺

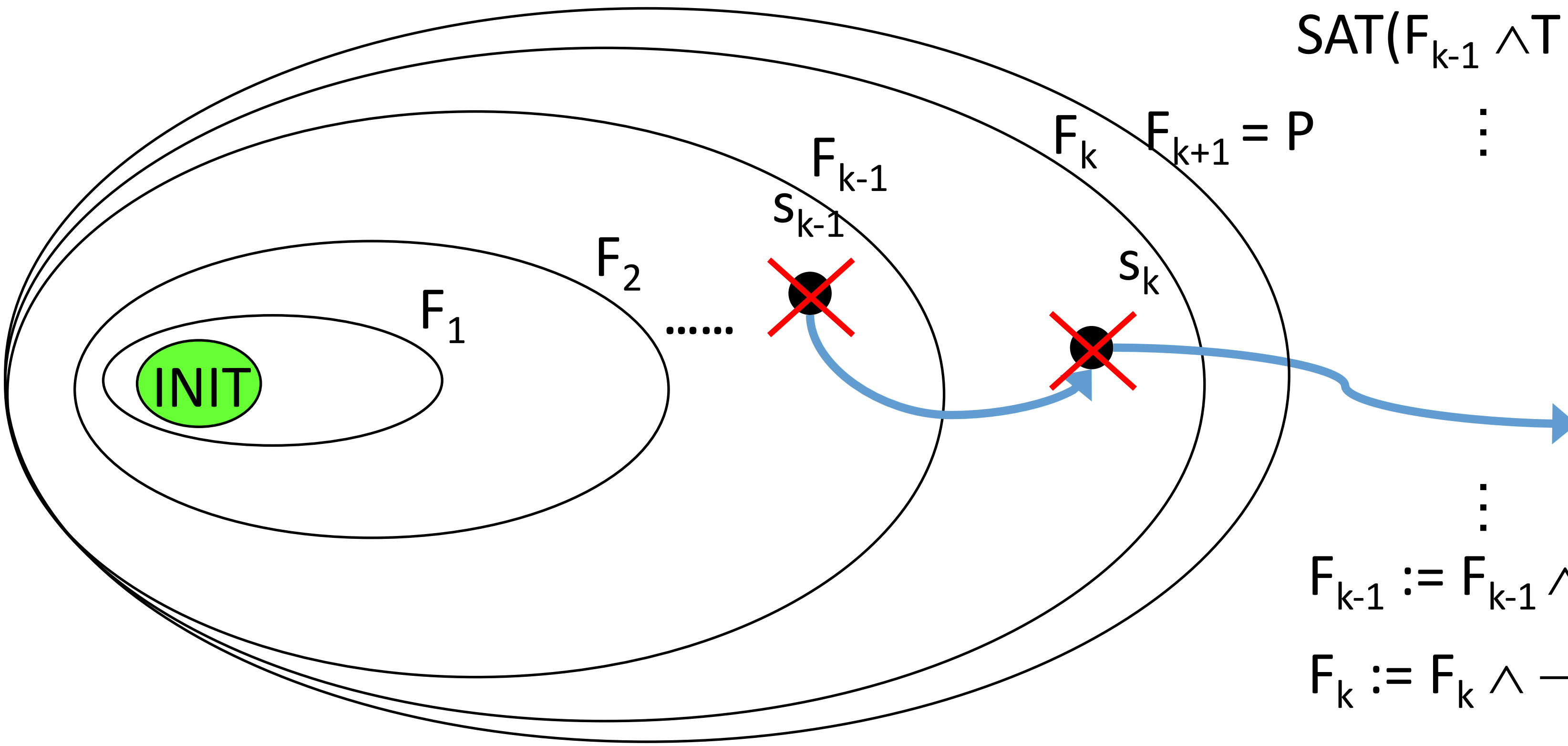


General Iteration

$\mathrm{SAT}(F_k \wedge T \wedge \neg P')$?

$\mathrm{SAT}(F_{k-1} \wedge T \wedge s_k')$?

⋮



⋮

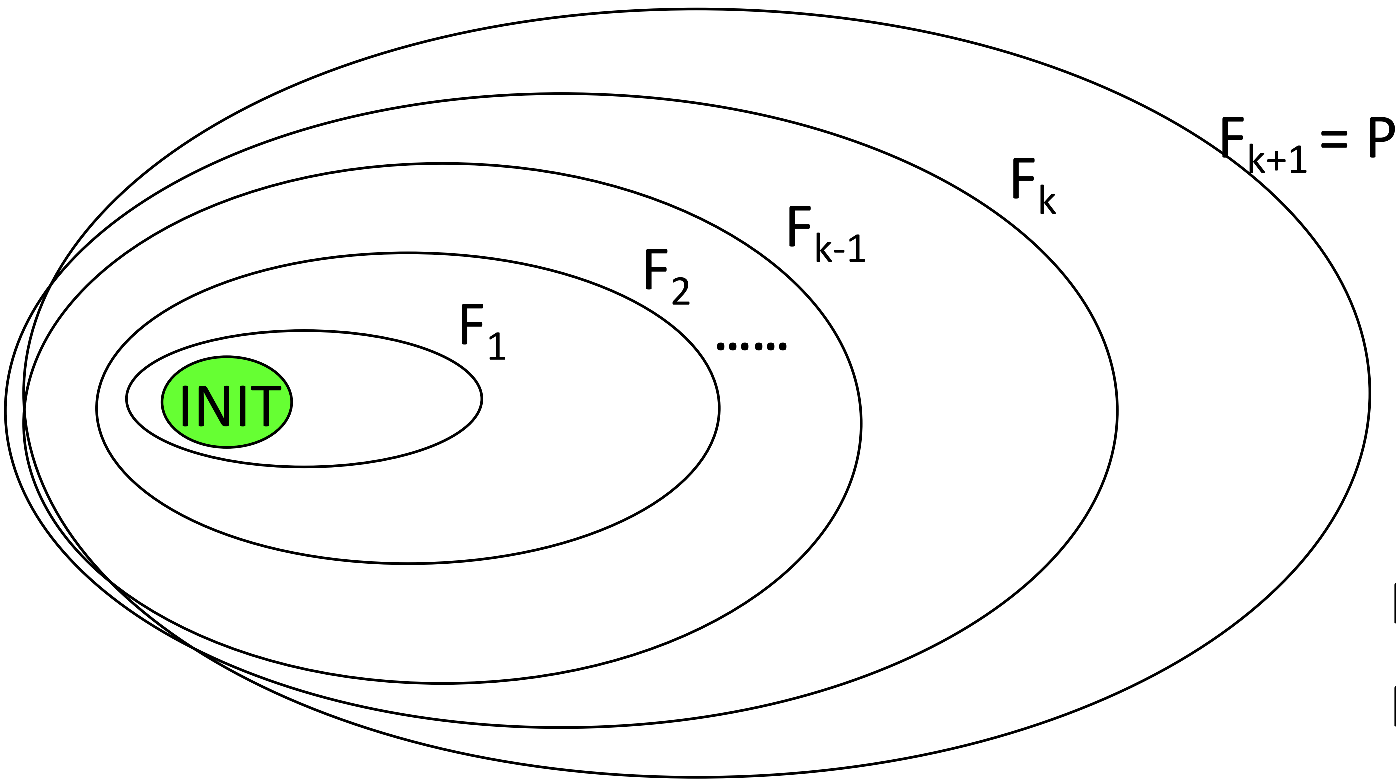
$F_{k-1} := F_{k-1} \wedge \neg s_{k-1}$

$F_k := F_k \wedge \neg s_k$

If s_k is reachable (in k steps): counterexample

If s_k is unreachable: strengthen F_k to exclude s_k

General Iteration



$\vdots$

$F_{k-1} := F_{k-1} \wedge \neg s_{k-1}$

$F_k := F_k \wedge \neg s_k$

Until $F_k \wedge T \wedge \neg P'$ is unsatisfiable, i.e. $F_k \wedge T \Rightarrow P'$

➔ We have an OARS again. Check fixpoint and increase k

IC3 - Iteration

Given an OARS $<F_0,F_1,\ldots,F_k>$, set $F_{k+1} = P$

Apply a backward search

1. Find predecessor s_k in F_k that can reach a bad state
   - $F_k \wedge T \not=> P'$ ($F_k \wedge T \wedge \neg P'$ is sat)
2. If none exists, move to next iteration (check fixpoint first)
3. If exists, try to find a predecessor s_{k-1} to s_k in F_{k-1}
   - $F_{k-1} \wedge T \not=> \neg s_k'$ ($F_{k-1} \wedge T \wedge s_k'$ is sat)
4. If none exists, remove s_k from F_k and go back to 3
   - $F_k := F_k \wedge \neg s_k$
5. Otherwise: Recur on (s_{k-1},F_{k-1})
   - We call $(s_{k-1},k-1)$ a “proof obligation” / “counterexample to induction”

If we reach INIT, a CEX exists

UNIVERSITY OF WATERLOO

That Simple?

Looks simple

- But this “simple” does NOT work

Simple = State Enumeration

- Too many states…

Does IC3 enumerate states?

- No – removing more than one state at a time
- But, yes (when IC3 doesn’t perform well)

Generalization of a blocked state

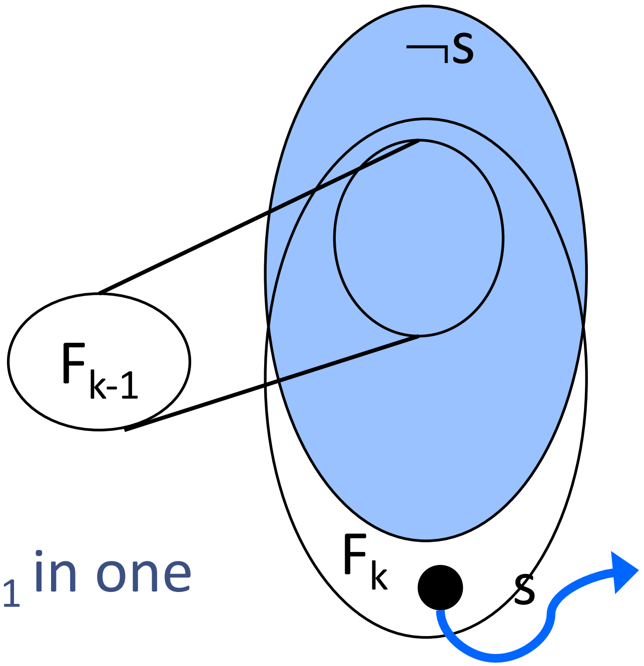
s in F_k can reach a bad state in one transition (or more)

But $F_{k-1} \wedge T \Rightarrow \neg s'$ holds

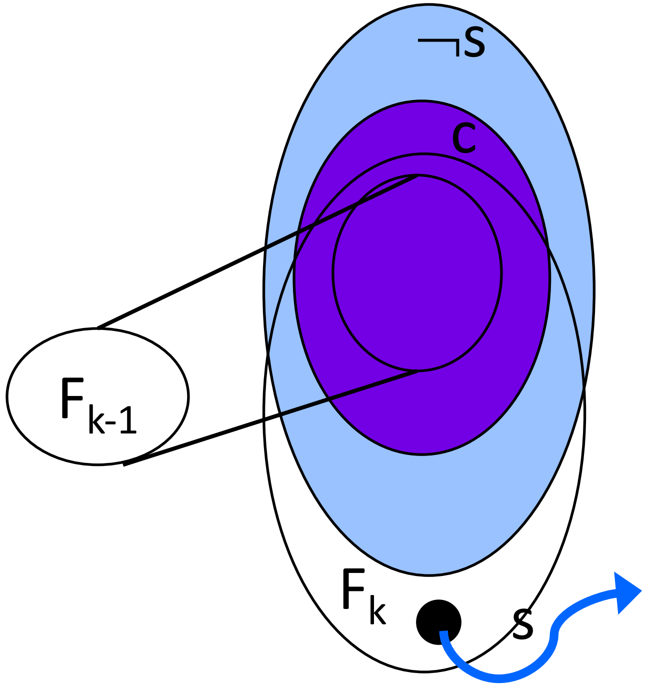
- Therefore, s is not reachable in k transitions
- $F_k := F_k \wedge \neg s$

We want to generalize this fact

- s is a single state
- Goal: learn a stronger fact
  - Find a set of states, unreachable from F_{k-1} in one step



Generalization



We know $F_{k-1} \wedge T \Rightarrow \neg s'$

And, $\neg s$ is a clause

Generalization:

Find a sub-clause $c \subseteq \neg s$ s.t.

$F_{k-1} \wedge$ T => c' and INIT => c

- Sub clause means less literals
- Less literals implies less satisfying assignments
  - $(a \vee b)$ vs. $(a \vee b \vee c)$
- $c \Rightarrow \neg s$ i.e. c is a stronger fact

$F_k := F_k \wedge c$

- More states are removed from $F_{k,}$ making it stronger/more precise (closer to R_k)

Generalization

How do we find a sub-clause $c \subseteq \neg s$ s.t. $\mathbf{F_{k-1} \wedge T \Rightarrow c'}$?

Trial and Error

- Try to remove literals from $\neg s$ while $\mathbf{F_{k-1}} \wedge \mathbf{T} \wedge \neg c'$ and $\mathbf{INIT} \wedge \neg c'$ remain unsatisfiable

Use the UnSAT Core

- $\mathbf{(INIT' \vee (F_{k-1} \wedge T))} \wedge s'$ is unsatisfiable
- Conflict clauses can also be used

$F_{k-1} \wedge T \wedge s'$ is UNSAT

Desired:

$c \Longrightarrow \neg s$

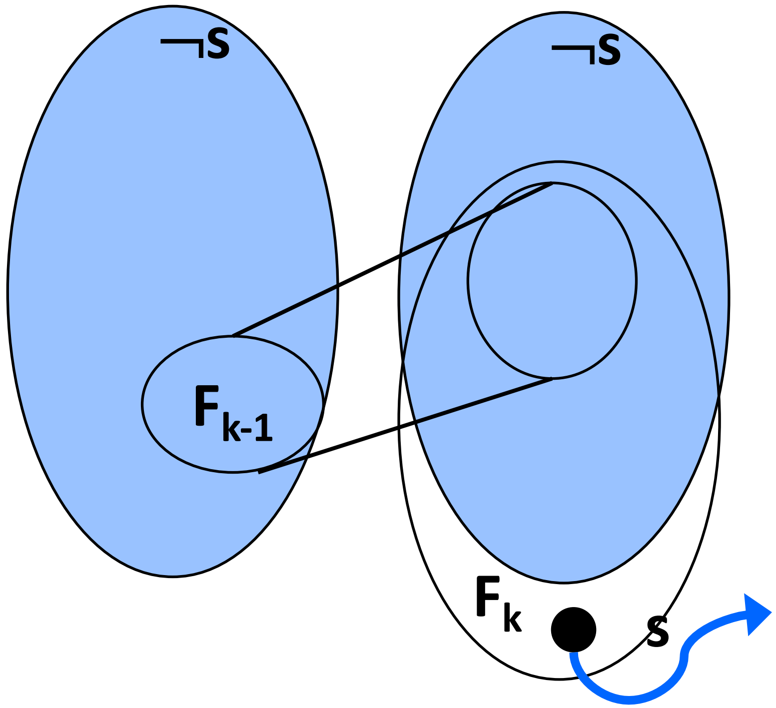
$F_{k-1} \wedge T \wedge \neg c'$ is UNSAT

Looks familiar?

Observation 1

Assume a state s in F_k can reach a bad state in a number of transitions

- Important Fact: **s is not in F_{k-1}** (!!)
  - If s was in F_{k-1} we would have found it in an earlier iteration



- Therefore: $F_{k-1} \Rightarrow \neg s$

Observation 1

Assume a state s in F_k can reach a bad state in a number of transitions

Therefore: $F_{k-1} \Rightarrow \neg s$

Assume $F_{k-1} \wedge T \Rightarrow \neg s'$ holds

- It's blocking time…

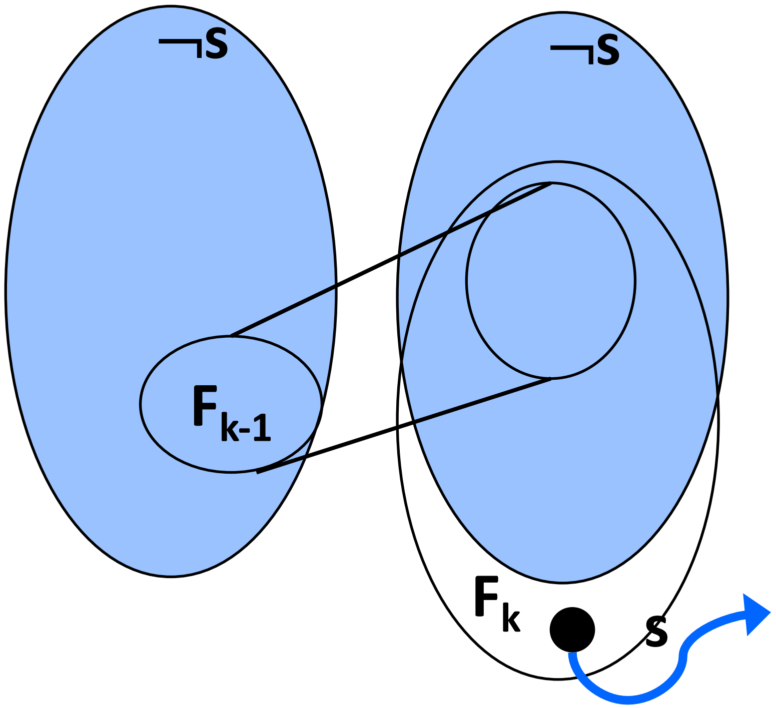
So, this is equivalent to

$$F_{k-1} \wedge \neg s \wedge T \Rightarrow \neg s'$$

Further INIT => ¬s

- Otherwise, CEX!
  (INIT ≠> ¬s IFF s is in INIT)

- This looks familiar!
  - ¬s is inductive relative to F_{k-1}



Inductive Generalization

We now know that $\neg s$ is inductive relative to F_{k-1}

And, $\neg s$ is a clause

Inductive Generalization:

Find sub-clause $c \subseteq \neg s$ s.t.

$\mathbf{F_{k-1} \wedge c \wedge T \Rightarrow c'}$ (and INIT => c)

- Stronger inductive fact

$F_k := F_k \wedge c$

- It may be the case that $\mathbf{F_{k-1} \wedge T \Rightarrow F_k}$ no longer holds
  - Why?

UNIVERSITY OF WATERLOO

Inductive Generalization

$F_{k-1} \wedge c \wedge T \Rightarrow c'$ and $INIT \Rightarrow c$ hold

$F_k := F_k \wedge c$

c is also inductive relative to $F_{k-1}, F_{k-2},\ldots,F_0$

- Add c to all of these sets
- For every $i \leq k$: $F_i^* = F_i \wedge c$

$\mathbf{F_i^* \wedge T \Rightarrow F_{i+1}}^*$ holds for every $i < k$

Observation 2

Assume state s in F_i can reach a bad state in a number of transitions

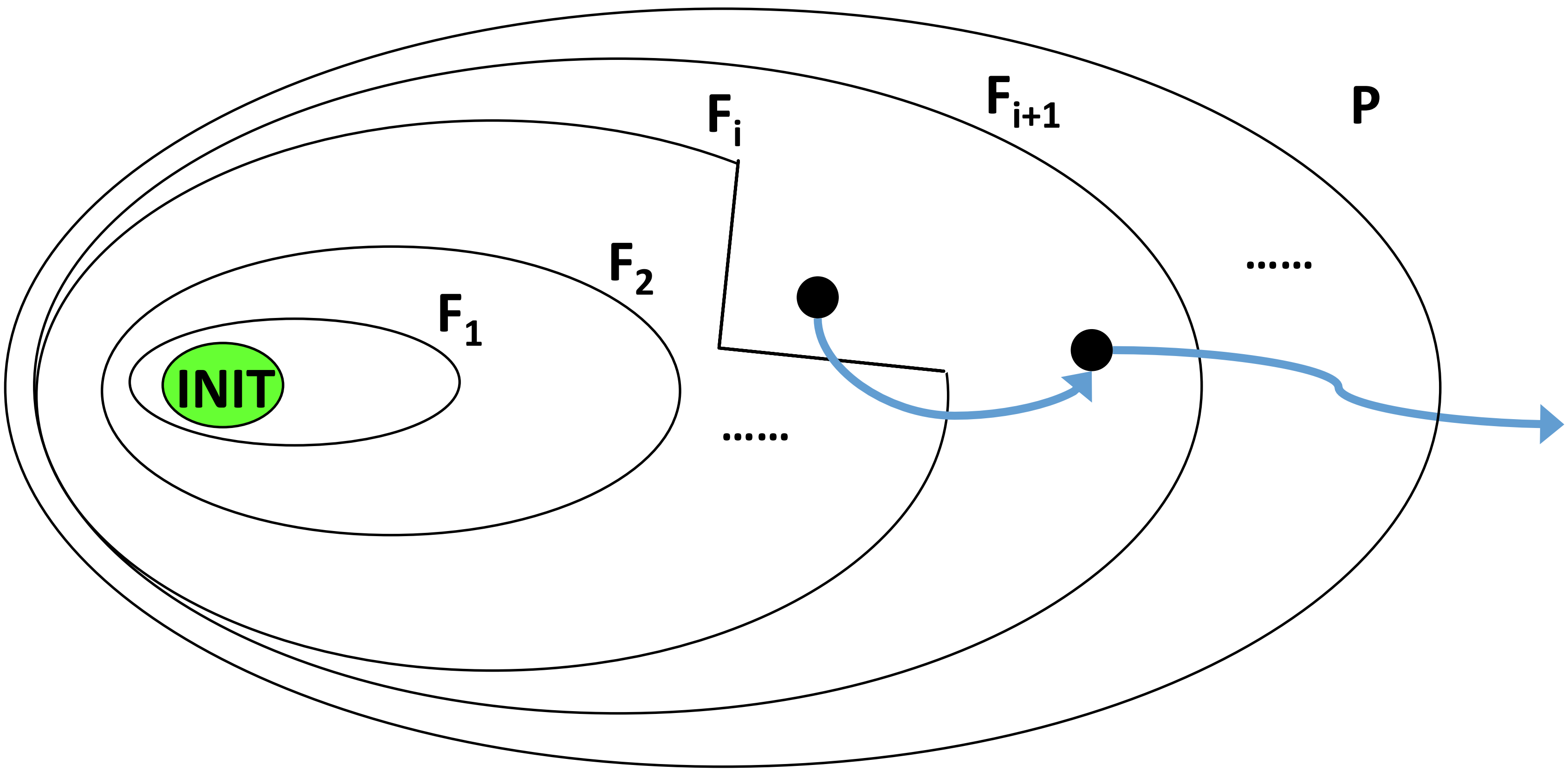
s is also in F_j for $j > i$ ($F_i \Rightarrow F_j$)

- a longer CEX may exist
- s may not be reachable in i steps, but it may be reachable in j steps

If s is blocked in F_i, it must be blocked in F_j for $j > i$

- Otherwise, a CEX exists

Push Forward



Push Forward

Suppose s is removed from F_i

- by conjoining a sub-clause c
- $F_i := F_i \wedge c$

c is a clause learnt at level i

try to push c forward for j > i

- If $F_j \wedge c \wedge T \Rightarrow c'$ holds
  - c is inductive in level j
  - $F_{j+1} := F_{j+1} \wedge c$
- Else: s was not blocked at level j > i
  - Add a proof obligation (s,j)
  - If s is reachable from INIT in j steps, CEX!

Generalizing Predecessor

Suppose s_{k-1} is a predecessor obtained by $F_{k-1} \wedge T \wedge s_k'$

- New proof obligation

Try to generalize s_{k-1} to a set of states (cube m) such that

$$m \Longrightarrow \exists V' . F_{k-1} \wedge T \wedge s_k'$$

- Drop a literal from s_{k-1} and use ternary simulation to check whether $F_{k-1} \wedge T \wedge s_k'$ evaluates to true under current assignment

Recursive Blocking Stage in IC3

```
// Find a counterexample, or strengthen the inductive trace
// s.t. F_N ⇒ ¬s holds
IC3_recBlockCube(s, N)
    Add(Q, (s, N))
    while ¬Empty(Q) do
        (s, k) ← Pop(Q)
        if (k = 0) return "Counterexample"
        if (F_k ⇒ ¬s) continue
        if (F_k-1 ∧ Tr ∧ s') is SAT
            t ← generalized predecessor of s
            Add(Q, (t, k-1))
            Add(Q, (s, k))
        else
            ¬t ← generalize ¬s by inductive generalization (to
                                                          level m≥k)
            add ¬t to F_m
            if (m<N) Add(Q, (s, m+1))
```

Pushing stage in IC3

```
// Push each clause to the highest possible frame up to N
IC3_Push()
    for k = 1 .. N-1 do
        for c ∈ F_k \ F_{k+1} do
            if (F_k ∧ Tr ⇒ c')
                add c to F_{k+1}
        if (F_k = F_{k+1})
            return "Proof" // F_k is a safe inductive invariant
```

IC3 – Key Ingredients

Backward Search

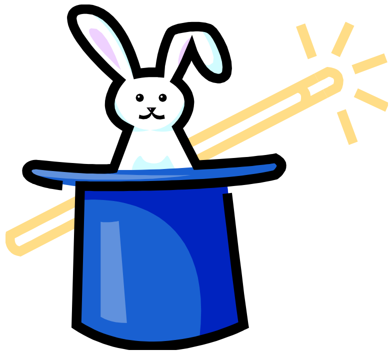
- Find a state s that can reach a bad state in a number of steps
- [lifting: generalize s to a set of states]
- s may not be reachable (over-approximations)

Block a State

- Do it efficiently, block more than s
  - Generalization / Inductive generalization

Push Forward

- An inductive fact at frame i, may also be inductive at higher frames
- If not, a longer CEX may be found



SOLVING CONSTRAINED HORN CLAUSES

Constrained Horn Clauses (CHC)

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, …, p_n, h are n-ary predicates
- $p_i[X]$ is an application of a predicate to first-order terms

CHC Notation and Terminology

Rule $\overset{\text{head}}{h[X]} \leftarrow \overbrace{p_1[X_1], \ldots, p_n[X_n], \overset{\text{constraint}}{\phi}}^{\text{body}}.$

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

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

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

Non-Linear CHC $h[X] \leftarrow p_1[X_1], \ldots, p_n[X_n], \phi.$

for $n > 1$

CHC Satisfiability

A $\mathcal{T}$-**model** of a set of a 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 $\boldsymbol{\mathcal{T}}$**-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

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

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

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

Proof:

add extra arguments to distinguish between predicates

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

UNIVERSITY OF WATERLOO

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 Airthmetic + 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

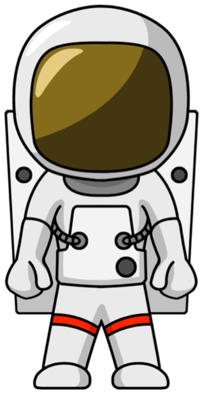
uPDR: Constraints in EPR fragment of FOL

- 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);
```

IS SAT?

```
z = x & i = 0 & y > 0                      →  Inv(x, y, z, i)
Inv(x, y, z, i) & i < y & z1=z+1 & i1=i+1  →  Inv(x, y, z1, i1)
Inv(x, y, z, i) & i >= y & z != x+y        →  false
```

In SMT-LIB

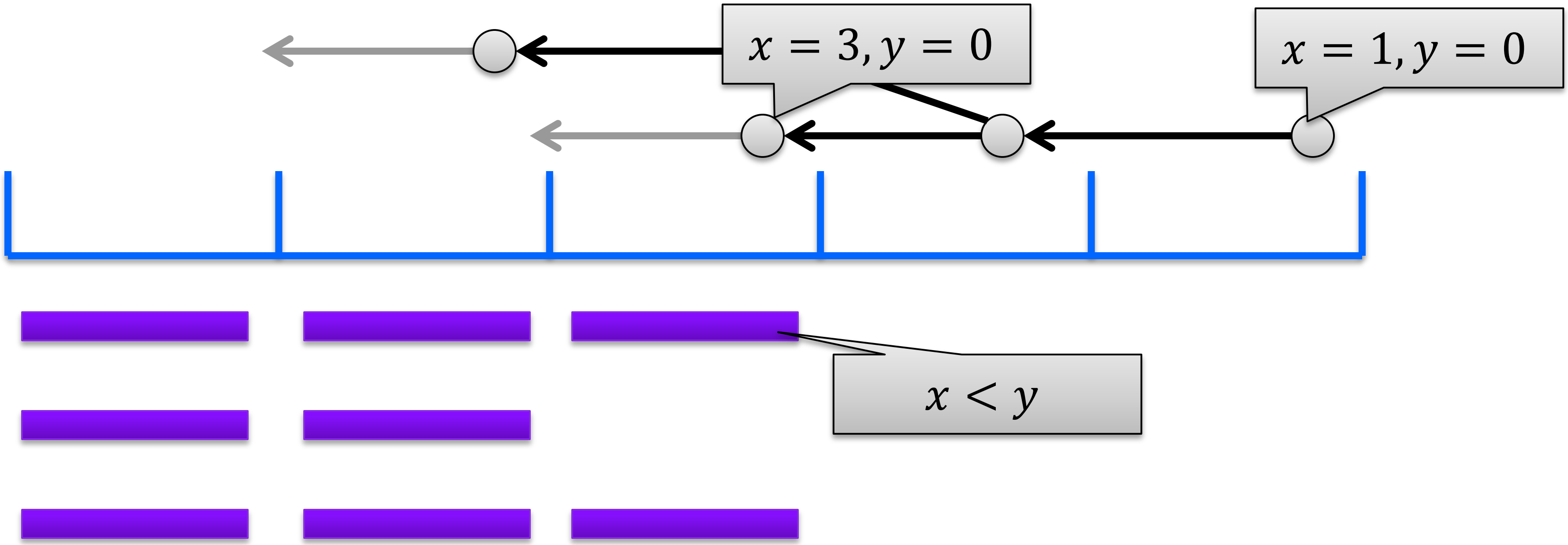
```
(set-logic HORN)

;; 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)))
)
```

```
Inv(x, y, z, i)
    z  = x + i
    z <= x + y
```

IC3/PDR In Pictures: MkSafe

MkSafe



Predecessor

$$\text{find } M \text{ s.t. } M \models F_i \wedge Tr \wedge m'$$

$$\text{find } m \text{ s.t. } (M \models m) \wedge (m \implies \exists V' \cdot Tr \wedge m')$$

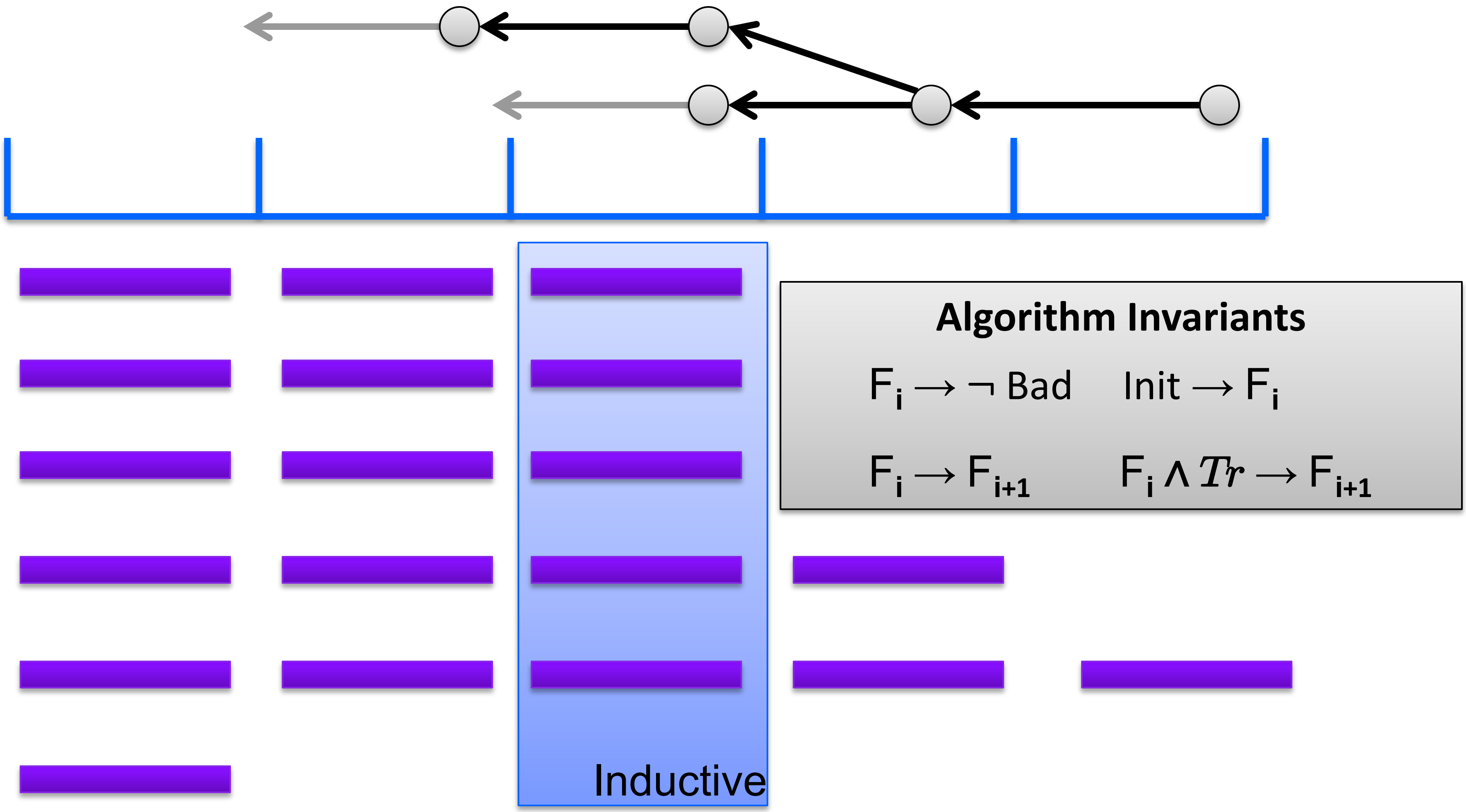
NewLemma

$$\text{find } \ell \text{ s.t. } (F_i \wedge Tr \implies \ell') \wedge (\ell \implies \neg m)$$

UNIVERSITY OF WATERLOO

IC3/PDR in Pictures: Push

Push



Algorithm Invariants

$F_i \rightarrow \neg \text{Bad}$ $\quad$ $\text{Init} \rightarrow F_i$

$F_i \rightarrow F_{i+1}$ $\quad$ $F_i \wedge \mathit{Tr} \rightarrow F_{i+1}$

$$\text{SMT-query: } \vdash \ell \wedge F_i \wedge Tr \implies \ell'$$

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
- (optionally) strengthen by dropping literals

UNIVERSITY OF WATERLOO

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

Theory dependent

Decide

- extend a cex (backward) consistent with the current frame
- choose an assignment ***s*** s.t. $(s \land R_i \land Tr \land cex')$ 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 $L \land R_i \land Tr \Rightarrow L'$

Induction

- propagate a lemma as far into the future as possible
- (optionally) strengthen by dropping literals

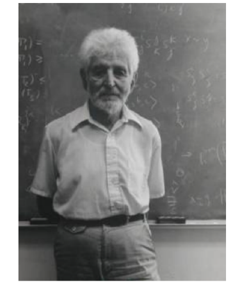
$$((F_i \wedge Tr) \vee Init') \Rightarrow \varphi'$$

$$\varphi' \Rightarrow \neg c'$$

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

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

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

UNIVERSITY OF WATERLOO

Examples of Craig Interpolation for Theories

Boolean logic

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

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

Equality with Uniterpreted Functions (EUF)

$$A = (f(a) = b \wedge p(f(a))) \qquad B = (b = c \wedge \neg p(c))$$

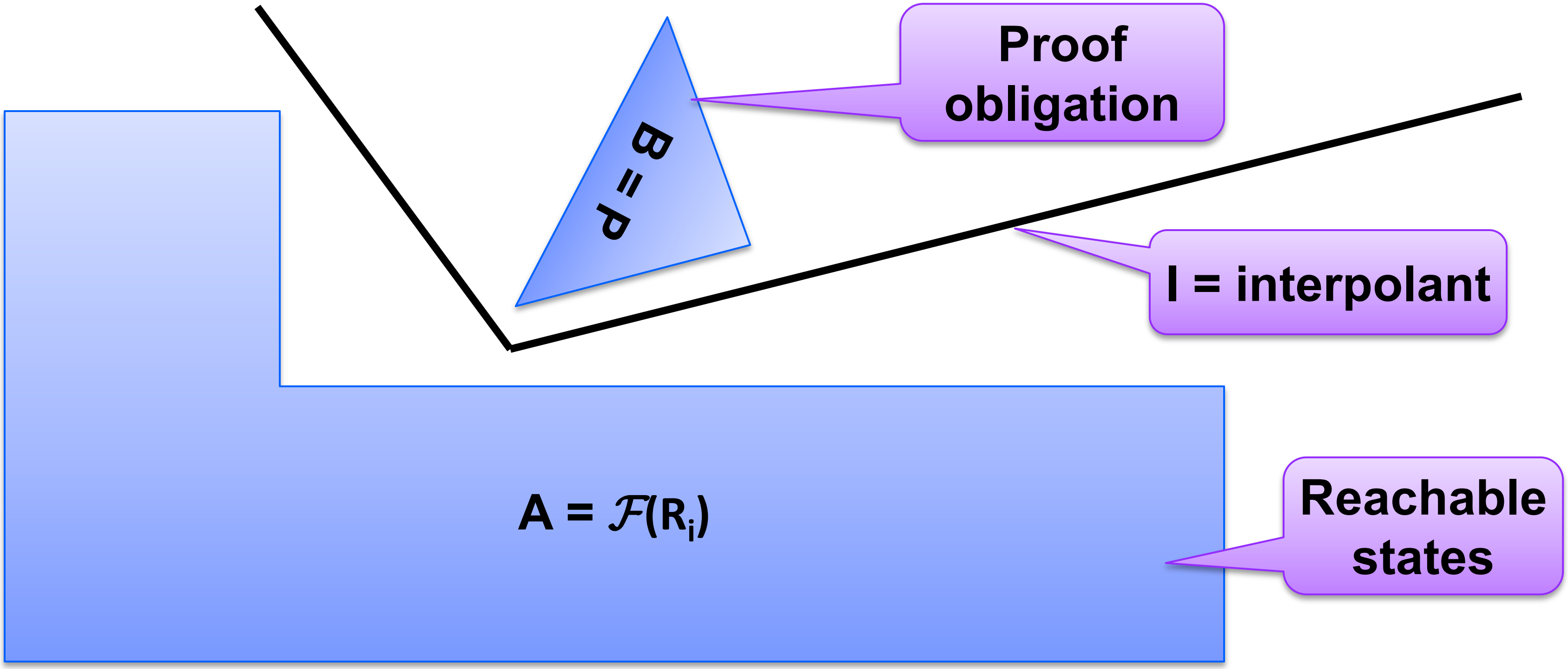
$$ITP(A, B) = p(b)$$

Linear Real Arithmetic (LRA)

$$A = (z + x + y > 10 \wedge z < 5) \qquad B = (x < -5 \wedge y < -3)$$

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

Craig Interpolation for Linear Arithmetic



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

- $I \in ITP(A, B)$ 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

Arithmetic Conflict

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

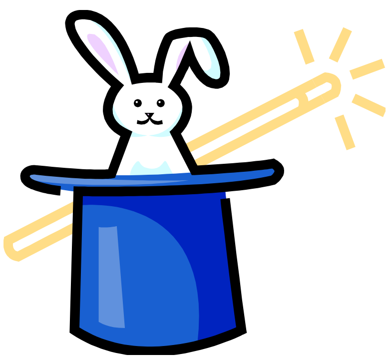
Conflict For $0 \leq i < N$, given a counterexample $\langle P, i+1 \rangle \in Q$ s.t. $\mathcal{F}(F_i) \wedge P'$ is unsatisfiable, add $P^{\uparrow} = \textsc{Itp}(\mathcal{F}(F_i), P')$ to F_j for $j \leq i+1$.

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 $(\bigwedge B_i \Rightarrow \bigvee A_j)$

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 $M = t_1 \geq b_1 \wedge \ldots \wedge t_n \geq b_n$, 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 $g_1, \ldots, g_n$ such that

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

Frakas Lemma Example

Interpolants

$$\left.\begin{aligned} z + x + y &> 10 & \times 1 \\ -z &> -5 & \times 1 \end{aligned}\right\} \quad x + y > 5$$

$$\left.\begin{aligned} -x &> 5 & \times 1 \\ -y &> 3 & \times 1 \end{aligned}\right\} \quad x + y < -8$$

$$\overline{}$$

$$0 > 13$$

Interpolation for Linear Real Arithmetic

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

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

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

Then

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

- $g_1 \times t_1 + \ldots + g_i \times t_i = -(g_{i+1} \times t_{i+1} + \ldots + g_n \times t_n)$
- $\neg(g_{i+1} \times (t_{i+1} \geq b_i) + \ldots + g_n \times (t_n \geq b_n))$ 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);
```

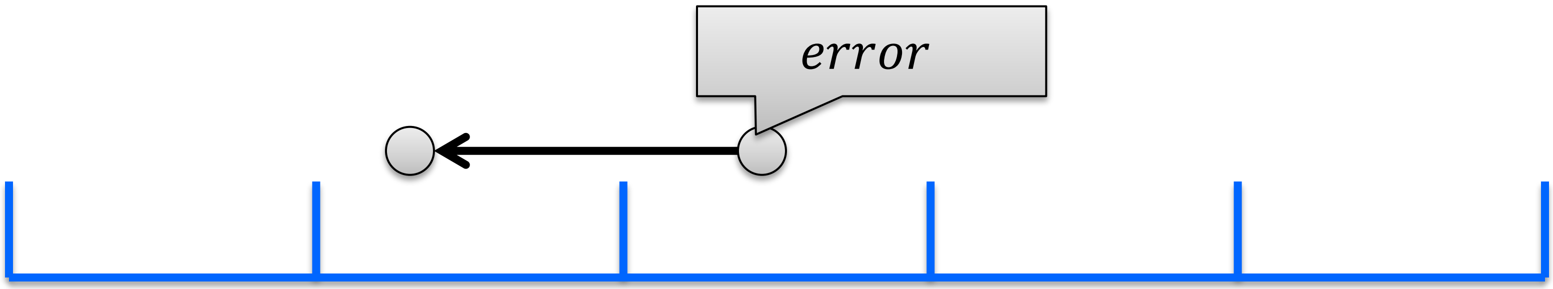
IS SAT?

```
z = x & i = 0 & y > 0                             →  Inv(x, y, z, i)
Inv(x, y, z, i) & i < y & z1=z+1 & i1=i+1         →  Inv(x, y, z1, i1)
Inv(x, y, z, i) & i >= y & z != x+y               →  false
```

UNIVERSITY OF WATERLOO

Lemma Generation Example

MkSafe



Transition Relation

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

Pob

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

Farkas explanation for unsat

$$\dfrac{\dfrac{x_0 + y_0 <= z_0,\ \ x <= x_0,\ z_0 < z,\ i <= i_0 + 1}{x + i <= z} \qquad \dfrac{i >= y,\ x+y > z}{x + i > z}}{\text{false}}$$

Learn lemma: $x + i <= z$

$$s \subseteq pre(c)$$

$$\equiv \quad s \Rightarrow \exists X' \,.\, Tr \wedge c'$$

Computing a predecessor ***s*** of a counterexample ***c***

ARITHMETIC DECIDE

Model Based Projection

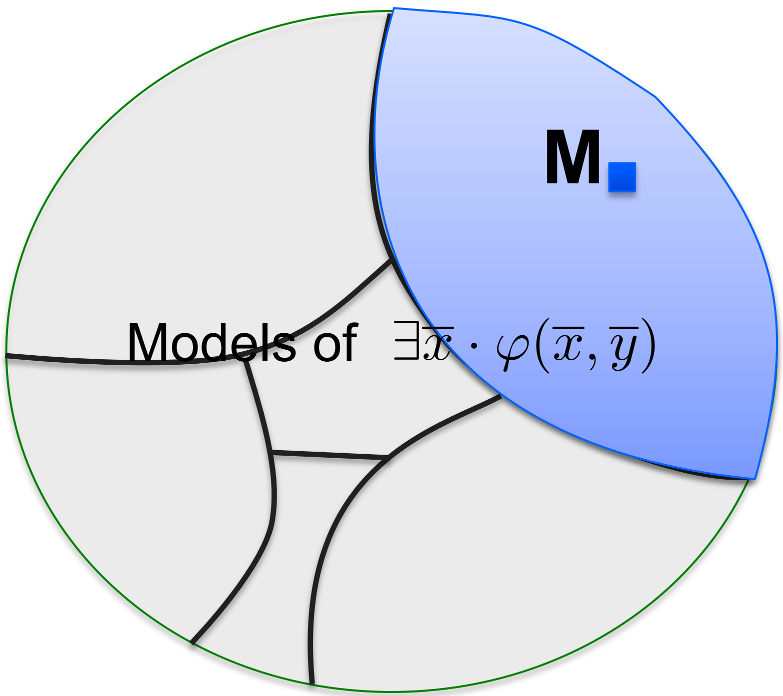
Definition: Let ϕ be a formula, U a set of variables, and M a model of ϕ. Then ψ = MBP (U, M, ϕ) is a Model Based Projection of U, M and ϕ iff

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

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 $\psi(\overline{y}) \equiv \exists \overline{x} \cdot \varphi(\overline{x}, \overline{y})$



1. Find model M of φ (x,y)

2. Compute a partition containing M

Loos-Weispfenning Quantifier Elimination

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 ϕ[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

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

The case of lower bounds is dual

- using –∞ and t+ϵ

Fourier–Motzkin Quantifier Elimination

$$\exists x \cdot \bigwedge_i s_i < x \wedge \bigwedge_j x < t_j$$

$$= \quad \bigwedge_i \bigwedge_j resolve(s_i < x, x < t_j, x)$$

$$= \quad \bigwedge_i \bigwedge_j s_i < t_j$$

Quadratic increase in the formula size per each eliminated variable

Quantifier Elimination with Assumptions

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

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 chose an the assumptions?!

- MBP == use the order chosen by the model

UNIVERSITY OF WATERLOO

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

UNIVERSITY OF WATERLOO

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 $P_\downarrow = \text{MBP}(X', m, \mathcal{F}(F_i) \wedge P')$.

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 **M** of ***s*** s.t. $(F_i \wedge Tr \wedge cex')$, and let **s** = $MBP(X', F_i \wedge Tr \wedge cex')$

Conflict

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

Induction

- propagate a lemma as far into the future as possible
- (optionally) strengthen by dropping literals

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

$$Init(X) \rightarrow P(X)$$

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

$$P(X) \rightarrow \neg Bad(X)$$

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

UNIVERSITY OF WATERLOO

Generalized GPDR

Input: A safety problem $\langle Init(X), Tr(X, X^o, X'), Bad(X)\rangle$.
Output: *Unreachable* or *Reachable*
Data: A cex queue Q, where a cex $\langle c_0, \ldots, c_k\rangle \in Q$ 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 $F_0, F_1, \ldots$
Notation: $\mathcal{F}(A, B) = Init(X') \vee (A(X) \wedge B(X^o) \wedge Tr)$, and $\mathcal{F}(A) = \mathcal{F}(A, A)$
Initially: $Q = \emptyset$, $N = 0$, $F_0 = Init$, $\forall i > 0 \cdot F_i = \emptyset$
Require: $Init \rightarrow \neg Bad$
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 $F_N \rightarrow \neg Bad$, then set $N \leftarrow N + 1$ and $Q \leftarrow \emptyset$.
- **Candidate** If for some m, $m \rightarrow F_N \wedge Bad$, 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 $l_0 \wedge m_0^o \wedge m_1' \rightarrow F_i \wedge F_i^o \wedge Tr \wedge m'$ 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 $\varphi = \textsc{Itp}(\mathcal{F}(F_i), m')$ 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**

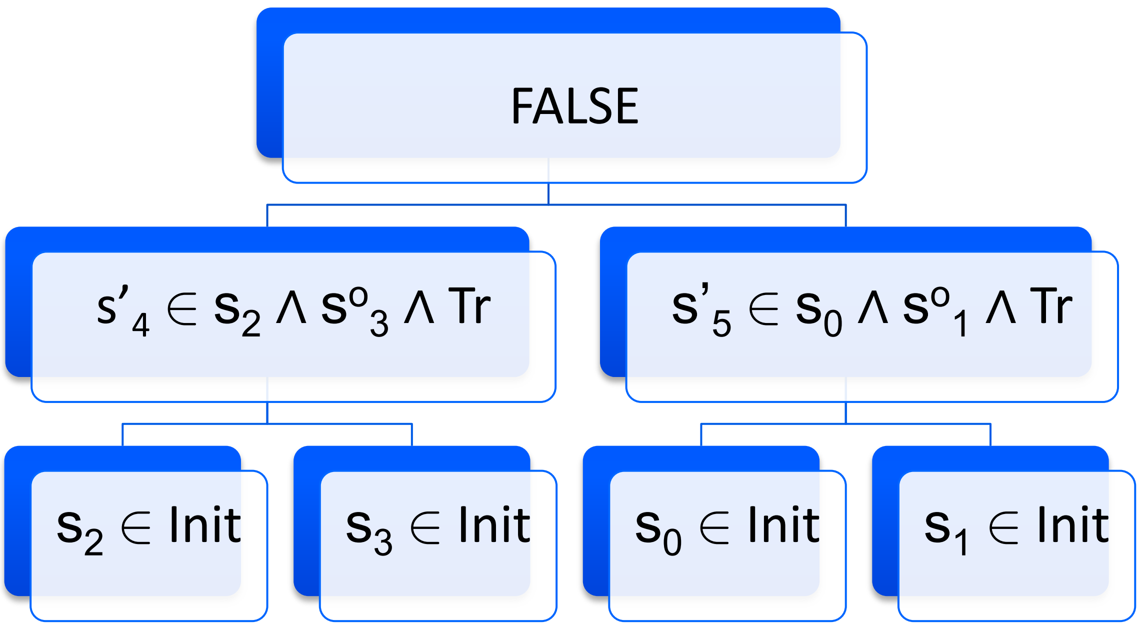
Counterexamples to non-linear CHC

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

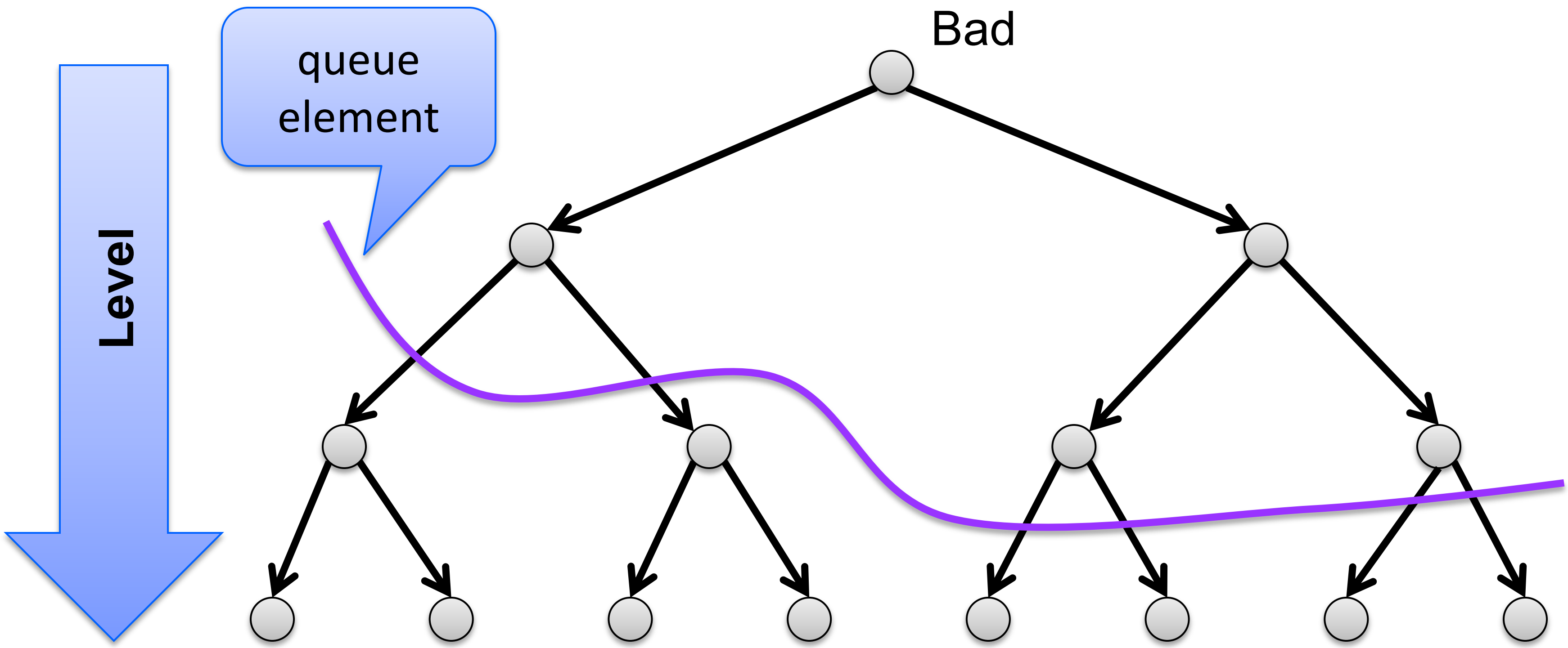
- we call such a derivation a counterexample

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

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

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

Predecessor over the constraint is:

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

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 $c = \langle m, i+1 \rangle \in t$, $m_1 \to m$, $l_0 \wedge m_0^o \wedge m_1'$ is satisfiable, and $l_0 \wedge m_0^o \wedge m_1' \to F_i \wedge F_i^o \wedge Tr \wedge m'$ 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

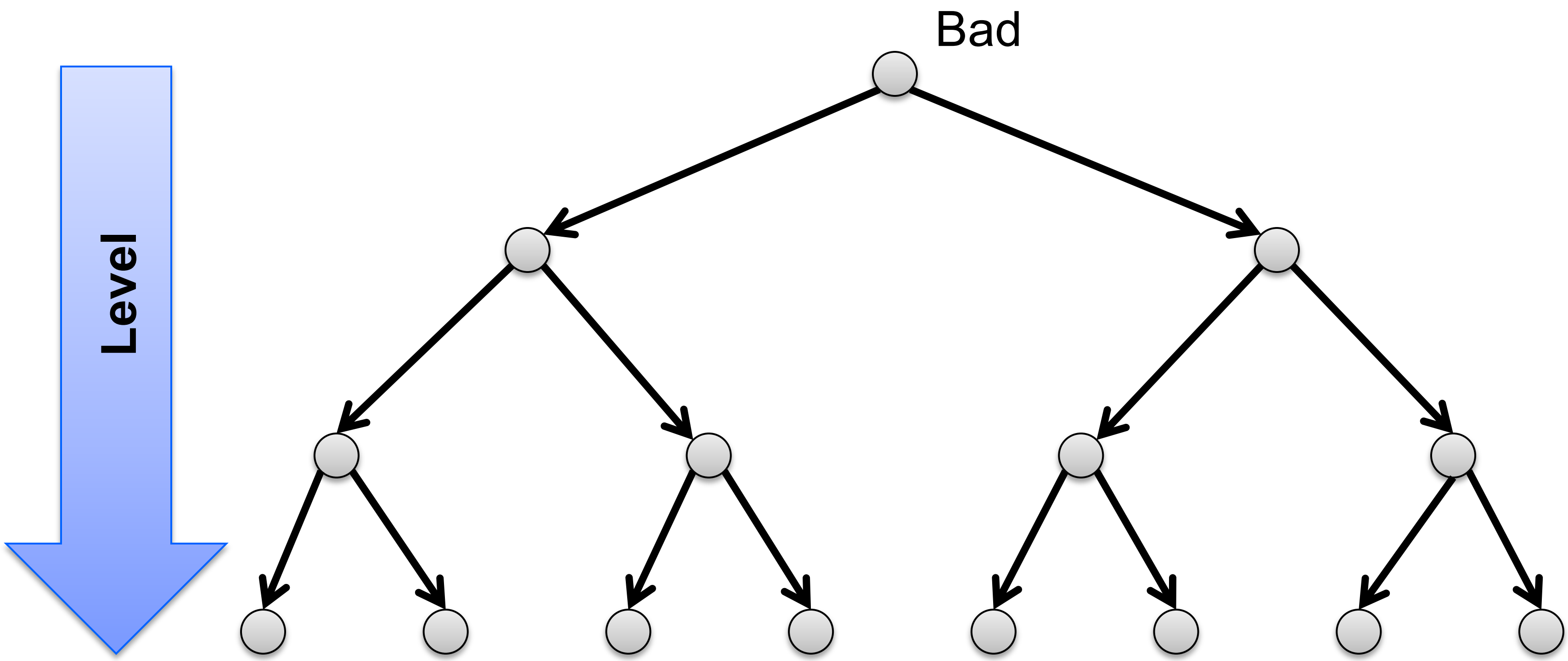
Input: A safety problem $\langle Init(X), Tr(X, X^o, X'), Bad(X)\rangle$.
Output: *Unreachable* or *Reachable*
Data: A cex queue Q, where a cex $c \in Q$ is a pair $\langle m, i\rangle$, m is a cube over state variables, and $i \in \mathbb{N}$. A level N. A set of reachable states REACH. A trace $F_0, F_1, \ldots$
Notation: $\mathcal{F}(A, B) = Init(X') \vee (A(X) \wedge B(X^o) \wedge Tr)$, and $\mathcal{F}(A) = \mathcal{F}(A, A)$
Initially: $Q = \emptyset$, $N = 0$, $F_0 = Init$, $\forall i > 0 \cdot F_i = \emptyset$, REACH $= Init$
Require: $Init \to \neg Bad$
repeat

- **Unreachable** If there is an $i < N$ s.t. $F_i \subseteq F_{i+1}$ **return** *Unreachable*.
- **Reachable** If REACH $\wedge\, Bad$ is satisfiable, **return** *Reachable*.
- **Unfold** If $F_N \to \neg Bad$, then set $N \leftarrow N + 1$ and $Q \leftarrow \emptyset$.
- **Candidate** If for some m, $m \to F_N \wedge Bad$, then add $\langle m, N\rangle$ to Q.
- **Successor** If there is $\langle m, i+1\rangle \in Q$ and a model M $M \models \psi$, 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 $M \models \psi$, 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 $M \models \psi$, 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 $0 \le j \le i + 1$.
- **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 $0 \le i < N$ and a clause $(\varphi \vee \psi) \in F_i$, if $\varphi \notin F_{i+1}$, $\mathcal{F}(\phi \wedge F_i) \to \phi'$, then add φ to F_j, for all $j \le i + 1$.

until ∞;

UNIVERSITY OF WATERLOO

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 $M \models \psi$, where $\psi = \mathcal{F}(\vee \textsc{Reach}) \wedge m'$. Then, add s to $\textsc{Reach}$, 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 $M \models \psi$, where $\psi = \mathcal{F}(F_i, \vee\textsc{Reach}) \wedge m'$. Then, add s to Q, where $s \in \mathrm{MBP}(\{X^o, X'\}, \psi)$.

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

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**)

CHC-COMP: CHC Solving Competition

First edition on July 13, 2018 at HVCS@FLOC

Constrained Horn Clauses (CHC) is a fragment of First Order Logic (FOL) that is sufficiently expressive to describe many verification, inference, and synthesis problems including inductive invariant inference, model checking of safety properties, inference of procedure summaries, regression verification, and sequential equivalence. The CHC competition (CHC-COMP) will compare state-of-the-art tools for CHC solving with respect to performance and effectiveness on a set of publicly available benchmarks. The winners among participating solvers are recognized by measuring the number of correctly solved benchmarks as well as the runtime.

Web: https://chc-comp.github.io/

Gitter: https://gitter.im/chc-comp/Lobby

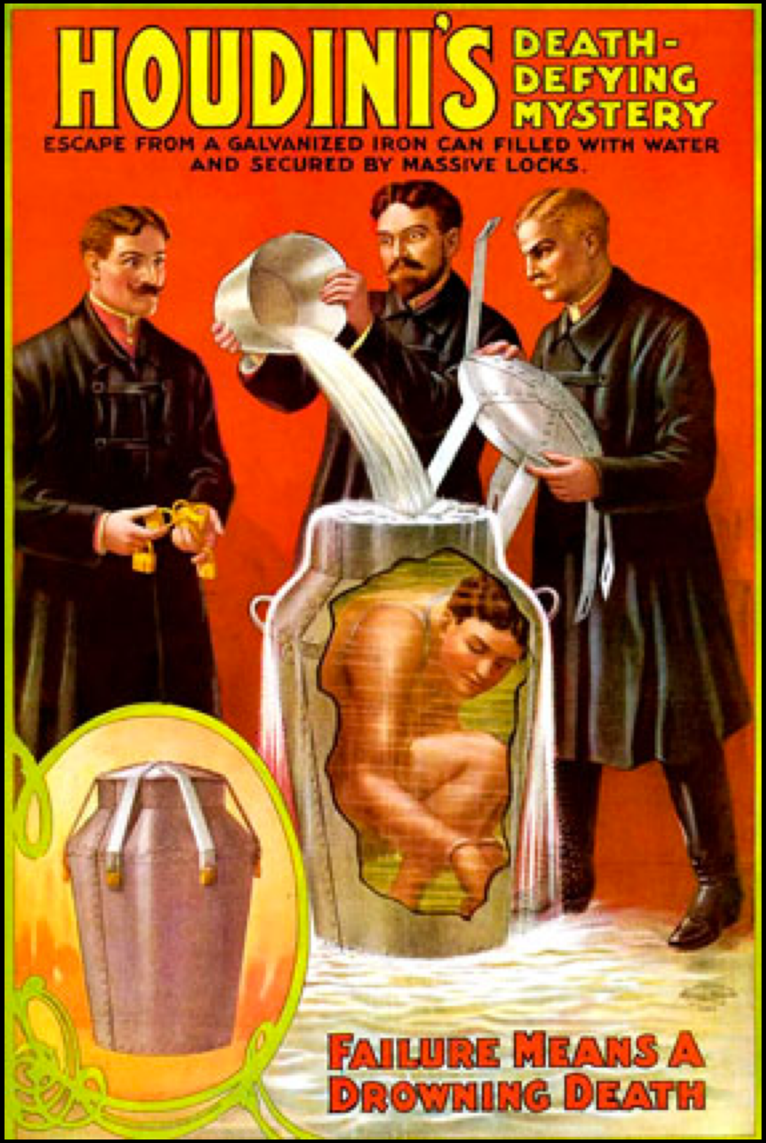
GitHub: https://github.com/chc-comp

Format: https://chc-comp.github.io/2018/format.html

CHC COMP

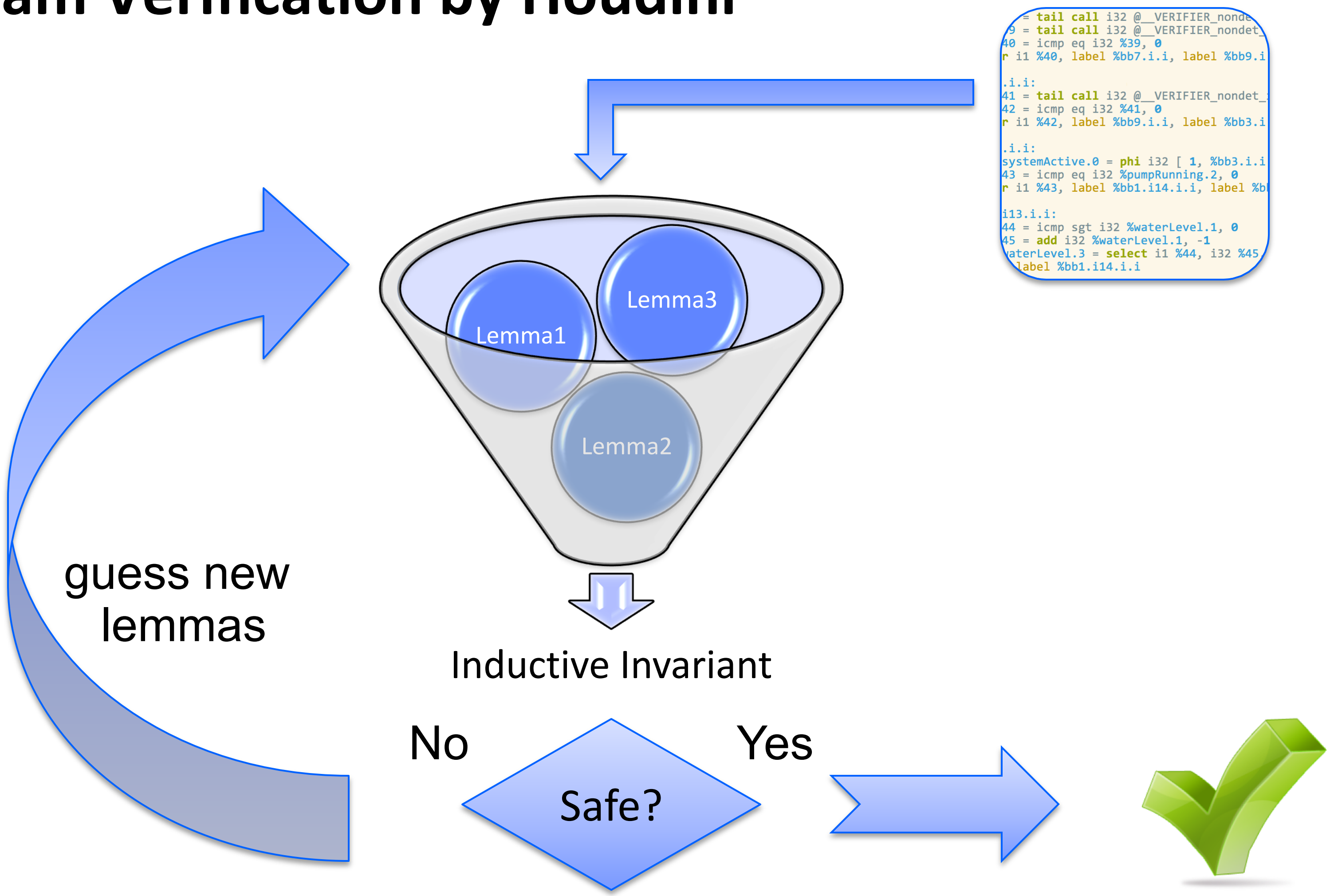
UNIVERSITY OF WATERLOO

CHC VIA MACHINE LEARNING



Cormac Flanagan, K. Rustan M. Leino: Houdini, an Annotation Assistant for ESC/Java. FME 2001: 500-517

Program Verification by Houdini



Finding an Inductive Invariant

Discovering an inductive invariants involves two steps

Step 1: find a candidate inductive invariant **Inv**

Step 2: check whether **Inv** is an inductive invariant

Invariant Inference is the process of automating both of these phases

Finding an Inductive Invariant

Two popular approaches to invariant inference:

Machine Learning based Invariant Synthesis (**MLIS**)

- e.g. ICE: Pranav Garg, Christof Löding, P. Madhusudan, Daniel Neider: ICE: A Robust Framework for Learning Invariants. CAV 2014: 69-87
- referred to as a Black-Box approach

SAT-based Model Checking (**SAT-MC**)

- e.g. IC3: Aaron R. Bradley: SAT-Based Model Checking without Unrolling. VMCAI 2011: 70-87
- referred to as a White-Box approach

Our Goal

Study the Relationship between SAT-MC and MLIS

Or, is there a difference between White-Box and Black-Box?

Our Goal

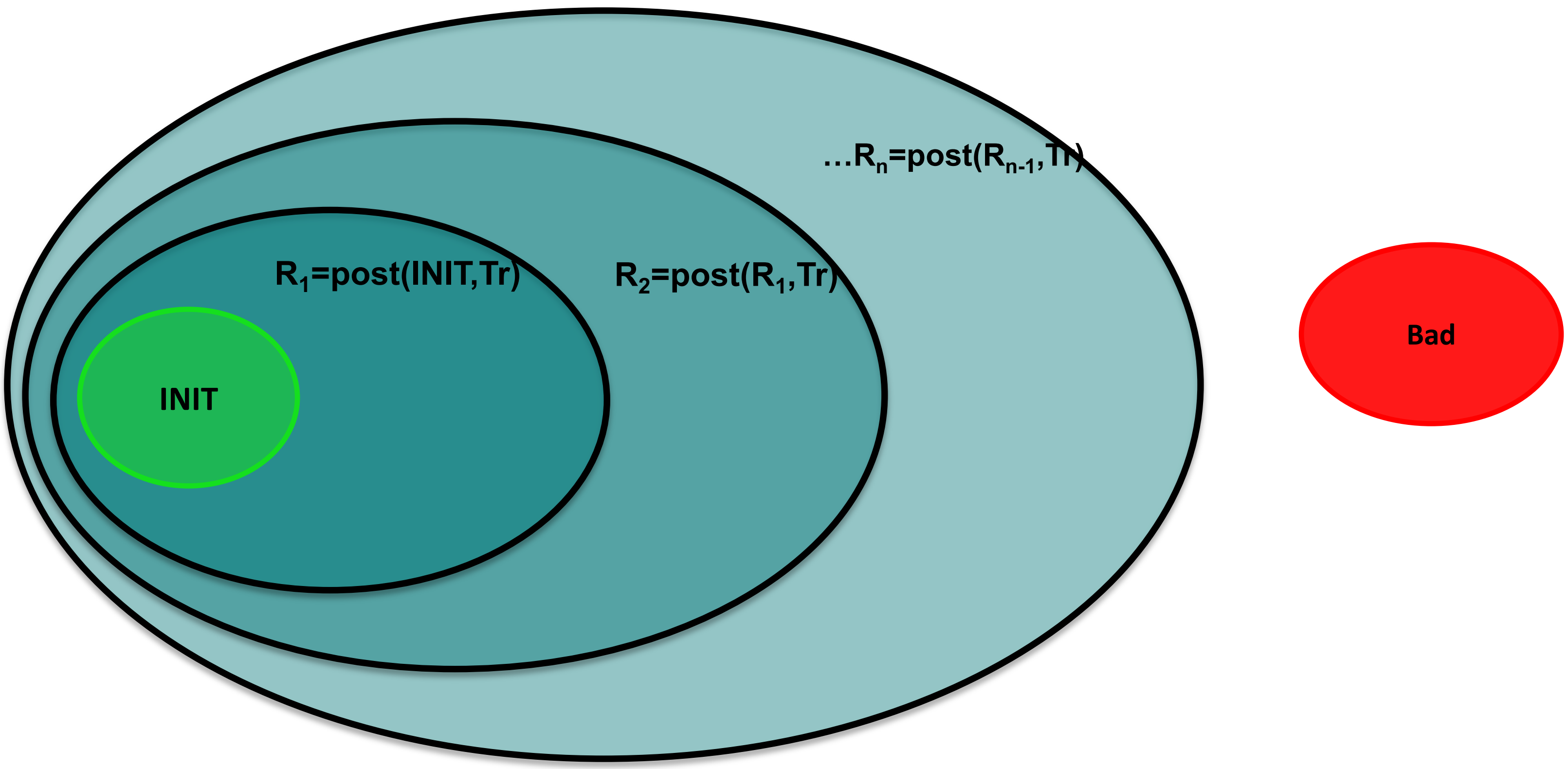
Study the Relationship between SAT-MC and MLIS

Or, is there a difference between White-Box and Black-Box?

- **Study two state-of-the-art algorithms: ICE and IC3**
- **In other words: can we describe IC3 as an instance of ICE?**

UNIVERSITY OF WATERLOO

Reachability Analysis



Reachability Analysis

Computing states reachable from a set of states S using the post operator

$$\begin{cases} post^0(S) = S \\ post^{i+1} = post^i(S) \cup \{t \mid s \in S \wedge (s,t) \in Tr\} \end{cases}$$

Computing states reaching a set of states S using the pre operator

$$\begin{cases} pre^0(S) = S \\ pre^{i+1} = pre^i(S) \cup \{t \mid s \in S \wedge (t,s) \in Tr\} \end{cases}$$

Transitive closure is denoted by post* and pre*

SAT-based Model Checking

Search for a counterexample for a specific length

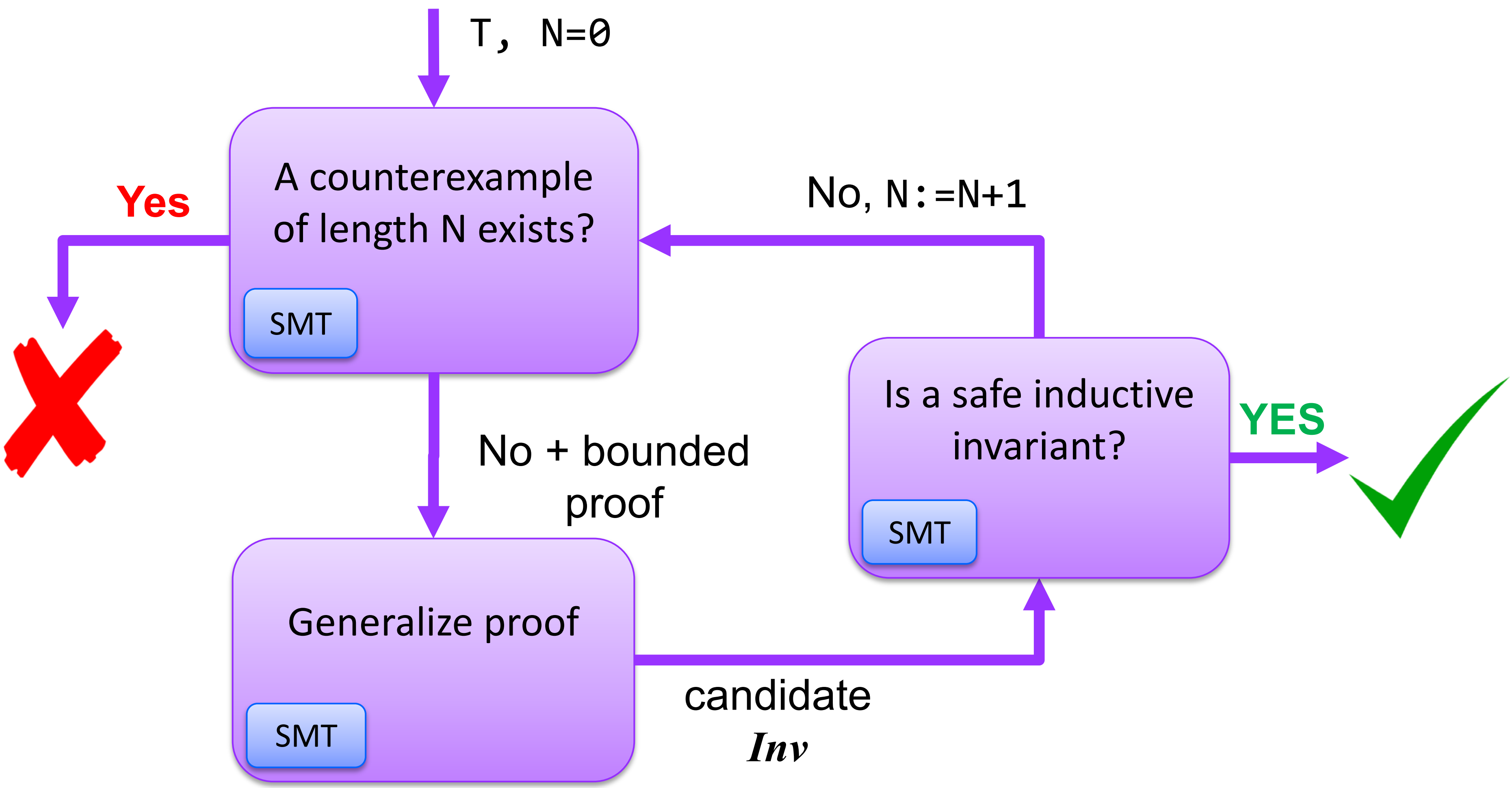
If a counterexample does not exist, generalize the bounded proof into a candidate Inv

Check if Inv is a safe inductive invariant

Referred to as White-Box: Rely on a close interaction between the main algorithm and the decision procedure used

SMT-based Model Checking

Generalizing from bounded proofs



Machine Learning-based Invariant Synthesis

MLIS consists of two entities: Teacher and Learner

Learner comes up with a candidate Inv

- Agnostic of the transition system
- Using machine learning techniques

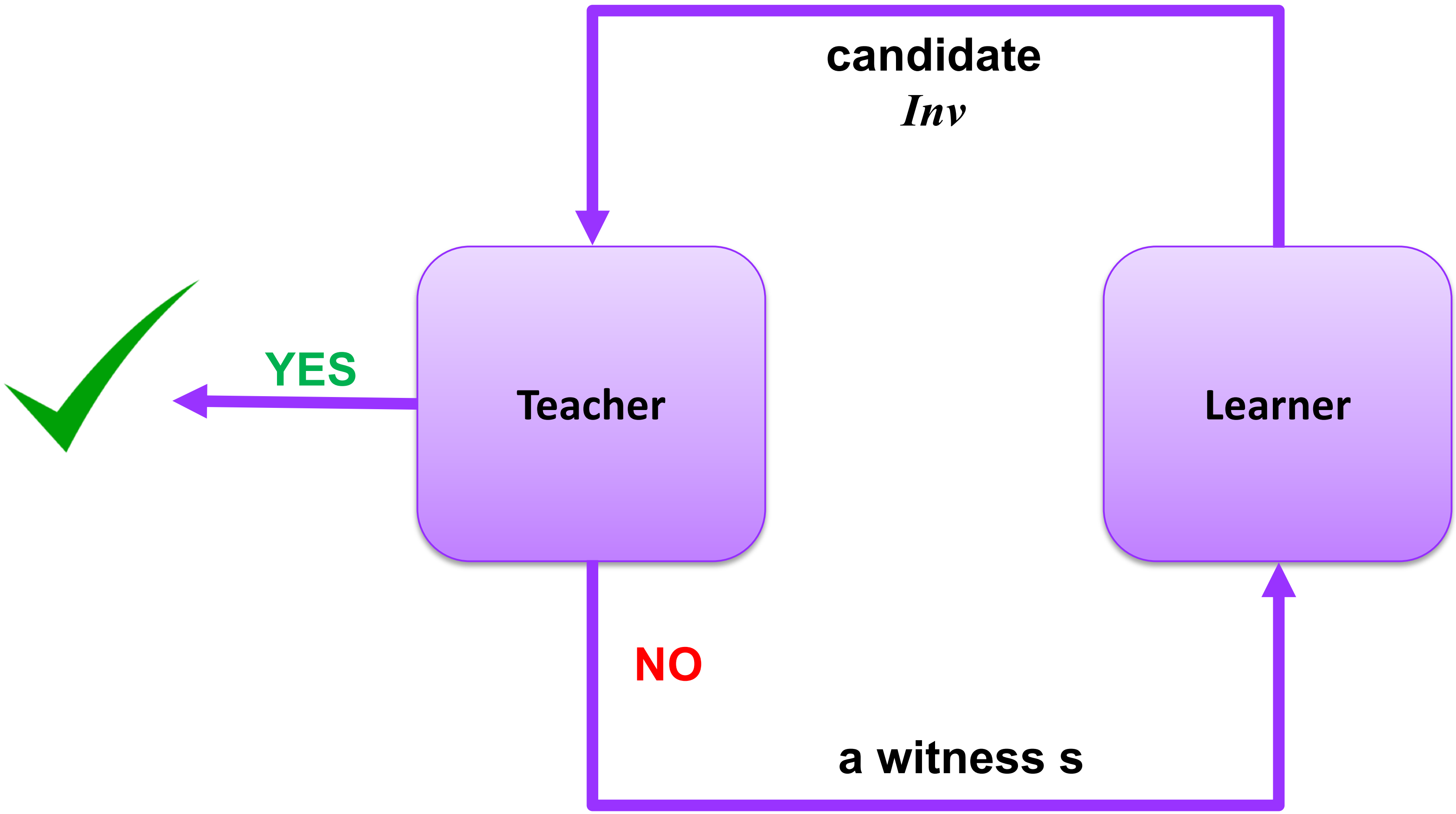
Learner asks the Teacher if Inv is a safe inductive invariant

If not, Teacher replies with a witness: positive or negative

- Aware of the transition system

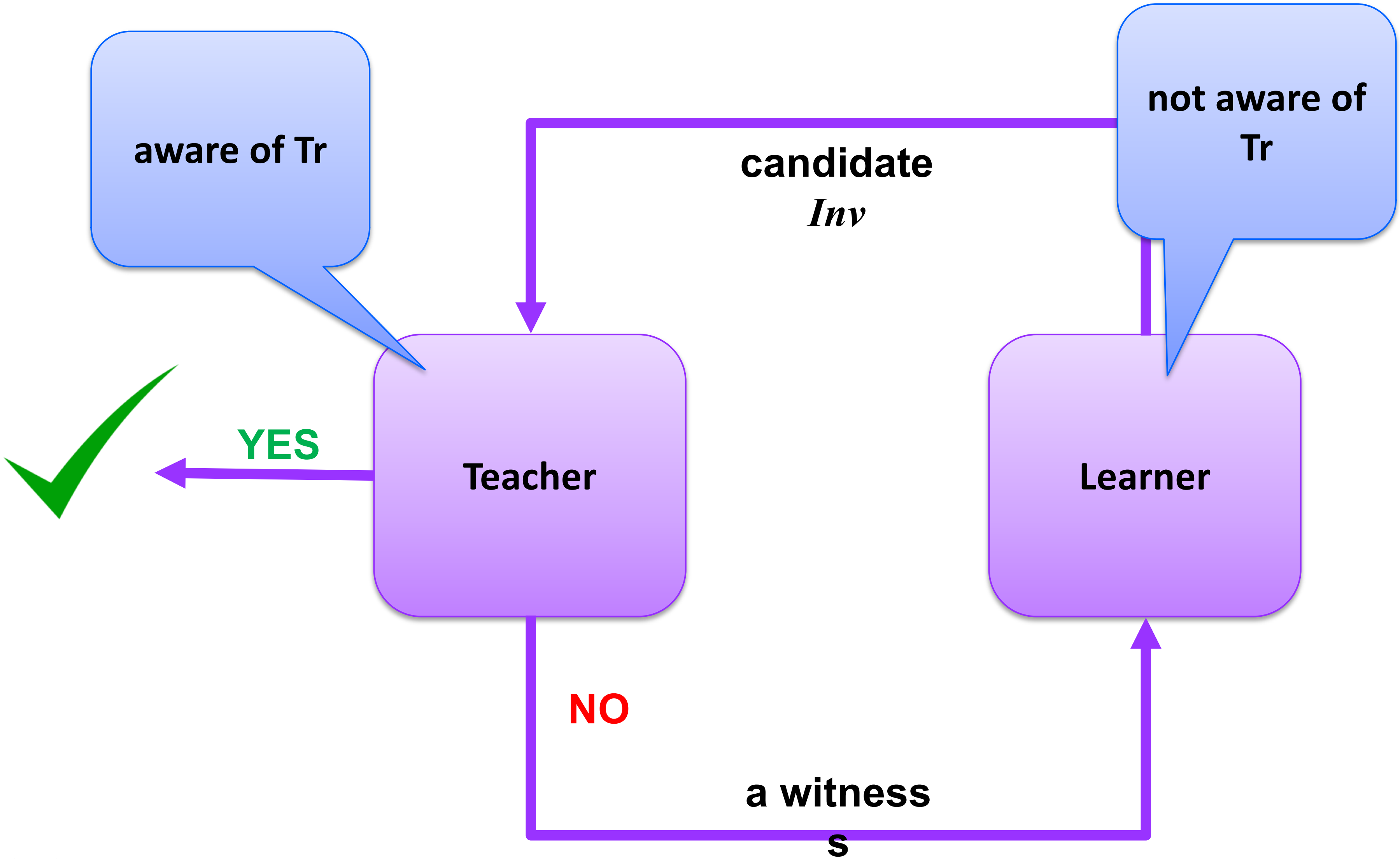
Referred to as Black-Box

Machine Learning-based Invariant Synthesis



UNIVERSITY OF WATERLOO

Machine Learning-based Invariant Synthesis



ICE: MLIS Framework

(Garg et al. CAV 2014)

Given a transition system T=(INIT, Tr, Bad) and a candidate *Inv* generated by the Learner

When the Teacher determines *Inv* is not a safe inductive invariant, a witness is returned:

- E-example: $s \in post^*(INIT)$ but $s \notin Inv$
- C-example: $s \in pre^*(Bad)$ and $s \in Inv$
- I-example: $(s,t) \in T$ such that $s \in Inv$ but $t \notin Inv$

Given a set of states S, the triple (E, C, I) is an **ICE state**

- $E \subseteq S$, $C \subseteq S$, $I \subseteq S \times S$

A set $J \subseteq S$ is **consistent** with ICE state iff

- $E \subseteq J$ and $J \cap C = \emptyset$
- for $(s,t) \in I$, if $s \in J$ then $t \in J$

ICE

(Garg et al. CAV 2014)

Input: A transition system $T = (\mathcal{V}, Init, Tr, Bad)$

$Q \leftarrow \emptyset$ $\quad$ $\textsc{Learner}(T)$; $\textsc{Teacher}(T)$;

repeat

$\quad J \leftarrow \textsc{Learner.SynCandidate}(Q)$;

$\quad \varepsilon \leftarrow \textsc{Teacher.IsInd}(J)$;

$\quad$ **if** $\varepsilon = \perp$ **then return** SAFE;

$\quad Q \leftarrow Q \cup \{\varepsilon\}$;

until ∞;

ICE

(Garg et al. CAV 2014)

Input: A transition system $T = (\mathcal{V},$

$Q \leftarrow \emptyset$ $\quad$ $\textsc{Learner}(T)$; $\textsc{Teacher}(T)$;

repeat

$\quad J \leftarrow \textsc{Learner}.\textsc{SynCandidate}(Q)$;

$\quad \varepsilon \leftarrow \textsc{Teacher}.\textsc{IsInd}(J)$;

$\quad$ **if** $\varepsilon = \bot$ **then return** SAFE;

$\quad Q \leftarrow Q \cup \{\varepsilon\}$;

until ∞;

No requirement for incrementality

J must be consistent with Q

The Learner is passive - has no control over the Teacher

PDR/IC3 – SAT Queries

Trace $[F_0,\ldots,F_N]$, and $Q \subseteq pre^*(Bad)$, a state $s \in Q \cap F_{i+1}$

Strengthening

- $(F_i \wedge \neg s) \wedge T \wedge s'$
- is $(F_i \wedge \neg s\ ) \wedge T \rightarrow \neg s'$ valid?

If this is satisfiable then there exists a state t in F_i that can reach Bad

- This looks like a C-example

In order to "fix" F_i t must be removed

Now check

- $(F_{i-1} \wedge \neg t) \wedge T \wedge t'$

PDR/IC3 – SAT Queries

Trace $[F_0,\ldots,F_N]$, try to push a lemma $c \in F_i$ to F_{i+1}

Pushing

- $(F_i \wedge c) \wedge T \wedge \neg c'$
- is $(F_i \wedge c) \wedge T \rightarrow c'$ valid?

If this is satisfiable then there exists a pair $(s,t) \in T$ s.t. $s \in F_i$ and $t \notin F_{i+1}$

- It looks like an I-example
  - Also, can be either an E- or C-example

In order to "fix" F_i, either s is removed from F_i or t is added to it

- Strengthening vs Weakening

The Problem

IC3 reasons about relative induction

F is inductive relative to G when:

- $INIT \rightarrow F$, and
- $G(V) \land F(V) \land T(V,V') \rightarrow F(V')$

But, in ICE, the Learner (Teacher) asks (answers) about induction

and, the Learner in ICE is passive

- cannot control the Teacher in any way
- No guarantee for incrementality

RICE – ICE + Relative Induction

Input: A transition system $T = (\mathcal{V}, Init, Tr, Bad)$

$Q \leftarrow \emptyset$;

$\textsc{Learner}(T)$; $\textsc{Teacher}(T)$;

repeat

  $(F, G) \leftarrow \textsc{Learner.SynCandAndBase}(Q)$;

  $\varepsilon \leftarrow \textsc{Teacher.IsRelInd}(F, G)$;

  if $\varepsilon = \bot \wedge G = true$ **then return** SAFE;

  $Q \leftarrow Q \cup \{\varepsilon\}$;

until ∞;

G allows the Learner to have some control over the Teacher

When G is true it is a regular inductive check

RICE – ICE + Relative Induction

The Teacher in RICE reacts to queries about relative induction

The Learner can "manipulate" the Teacher using relative induction

RICE is a generalization of ICE where the Learner is an active learning algorithm

RICE – ICE + Relative Induction

The Teacher in RICE reacts to queries about relative induction

Is F inductive relative to G?

If not, a witness is returned:

- E-example: s ∈ post*(INIT) but s $\notin F$
- C-example: s ∈ pre*(Bad) and s $\in F$

- I-example: (s,t) ∈ T such that s $\in F \wedge G$ but t $\notin F$

IC3 AS AN INSTANCE OF RICE

IC3 Learner

The IC3 Learner is active and incremental

Maintains the following:

- a trace [F_0, …, F_N] of candidates
- RICE state Q=(E, C, I)

The Learner must be consistent with the RICE state

E-examples and C-examples may exist when F is inductive relative to G

- The Teacher may return an E-example or C-example when F is inductive relative to G

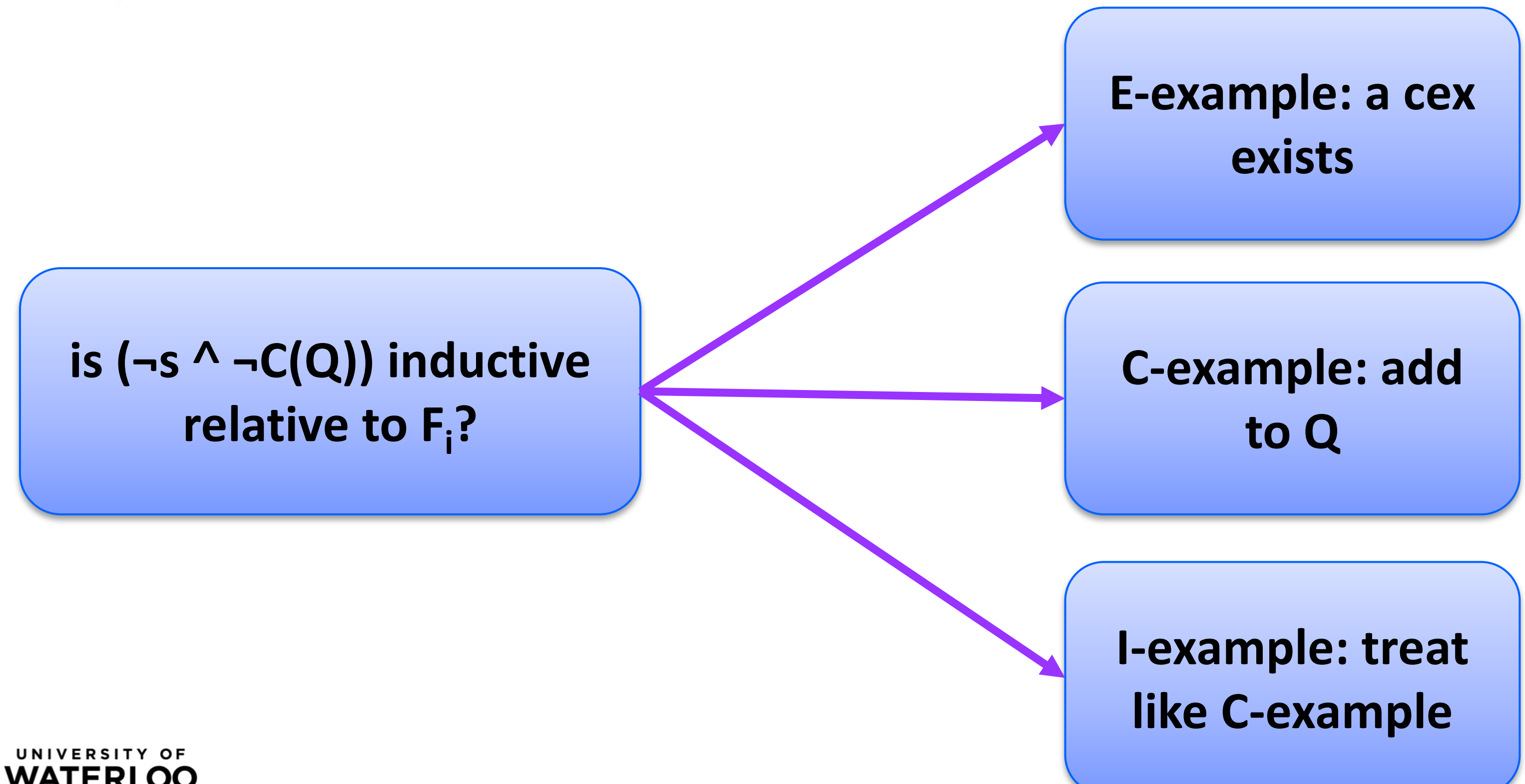
IC3 Learner - Strengthening

INIT → F, and

$G(V) \wedge F(V) \wedge T(V,V') \rightarrow F(V')$

Strengthening:

- a C-example s in F_i
- $(F_i \wedge \neg s \wedge \neg C(Q)) \wedge T \wedge (s \vee C(Q))'$



IC3 Learner - Pushing

INIT → F, and

$G(V) \land F(V) \land T(V,V') \rightarrow F(V')$

Pushing:

- a lemma c in F_i
- $(F_i \land c \land \neg C(Q) \land F_{i+1}) \land T \land (\neg c \lor C(Q) \lor \neg F_{i+1})'$

is $(c \land \neg C(Q) \land F_{i+1})$ inductive relative to F_i?

E-example: do not push and add to Q

C-example: do not push and add to Q

I-example: do not push and add to Q

IC3 Learner - Pushing

E- and C-examples may exist even when relative induction holds

Pushing:

- a lemma c in F_i
- $(F_i \land c \land \neg C(Q) \land F_{i+1}) \land T \land (\neg c \lor C(Q) \lor \neg F_{i+1})'$

is $(c \land \neg C(Q) \land F_{i+1})$ inductive relative to F_i?

E-example: do not push and add to Q

C-example: do not push and add to Q

I-example: do not push and add to Q

IC3 Teacher

Using a general Teacher, the described Learner computes a trace $[F_0, \ldots, F_N]$ such that

- post*(INIT) $\rightarrow F_i \rightarrow$ ¬pre*(Bad)

Generic Teacher is infeasible

- required to look arbitrary far into the future (for E-examples)
- required to look arbitrary far into the past (for C-examples)

Solution: add restrictions on E- and C-examples

IC3 Teacher

Is F inductive relative to G?

If not, a witness is returned:

- C-example: s ∈ pre^m(Bad) and s ∈ F
- I-example: (s,t) ∈ T such that s ∈ $F \wedge G$ but t ∉ F
- E-example: s ∈ $post^0$(INIT) but s ∉ F

Claim: Using this IC3 Teacher and the IC3 Learner results in an algorithm that behaves like (simulates) IC3

What Can We Learn?

Can we lift the restriction that requires E-example to be in INIT only?

- Yes, a variant of IC3, called Quip, does that

There is no “real” weakening mechanism in IC3

- Future work…

Can we introduce other active Learners for MLIS?

Conclusions

An extension of ICE to RICE

- Taking ques from IC3: incrementality, active Learner
- Overcomes a deficiency in ICE

IC3 can benefit from (R)ICE

- Weakening, E-examples, …

CHC-COMP: CHC Solving Competition

First edition on July 13, 2018 at HVCS@FLOC

Constrained Horn Clauses (CHC) is a fragment of First Order Logic (FOL) that is sufficiently expressive to describe many verification, inference, and synthesis problems including inductive invariant inference, model checking of safety properties, inference of procedure summaries, regression verification, and sequential equivalence. The CHC competition (CHC-COMP) will compare state-of-the-art tools for CHC solving with respect to performance and effectiveness on a set of publicly available benchmarks. The winners among participating solvers are recognized by measuring the number of correctly solved benchmarks as well as the runtime.

Web: https://chc-comp.github.io/

Gitter: https://gitter.im/chc-comp/Lobby

GitHub: https://github.com/chc-comp

Format: https://chc-comp.github.io/2018/format.html

CHC COMP