Algorithmic Logic-Based Verification with SeaHorn

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Arie Gurfinkel with Teme Kahsai and Jorge A. Navas

based on joint work with Anvesh Komuravelli, and Nikolaj Bjørner

Automated Software Analysis

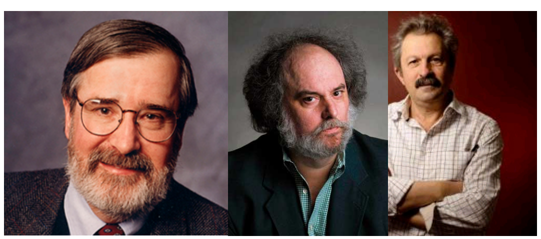
Program → Automated Analysis → Correct / Incorrect

Software Model Checking with Predicate Abstraction

e.g., Microsoft's SDV

Abstract Interpretation with Numeric Abstraction

e.g., ASTREE, Polyspace



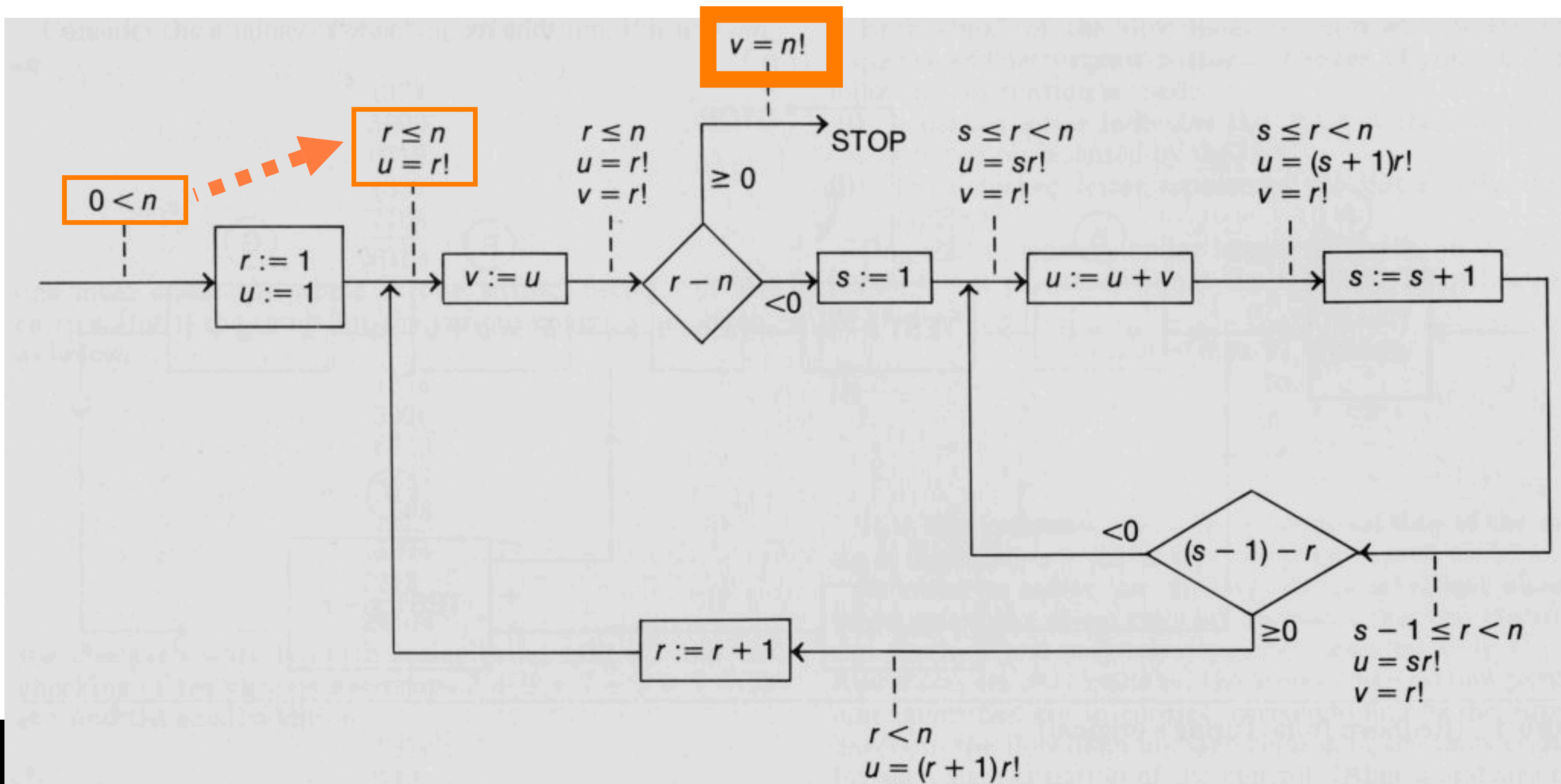
Turing, 1936: “undecidable”

Turing, 1949

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.



Three-Layers of a Program Verifier

Compiler

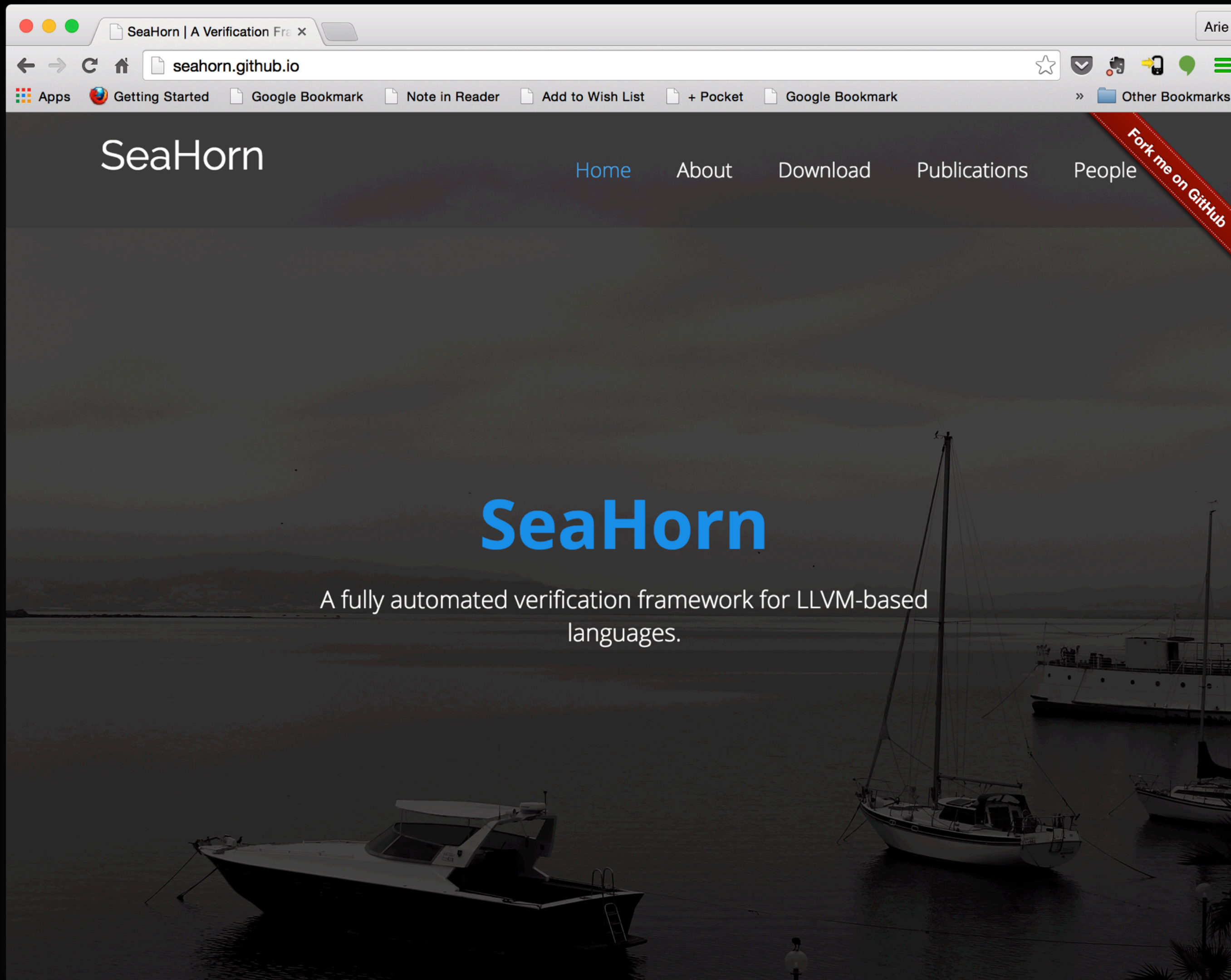
- compiles surface syntax a target machine
- embodies syntax with semantics

Verification Condition Generator

- transforms a program and a property to a verification condition in logic
- employs different abstractions, refinements, proof-search strategies, etc.

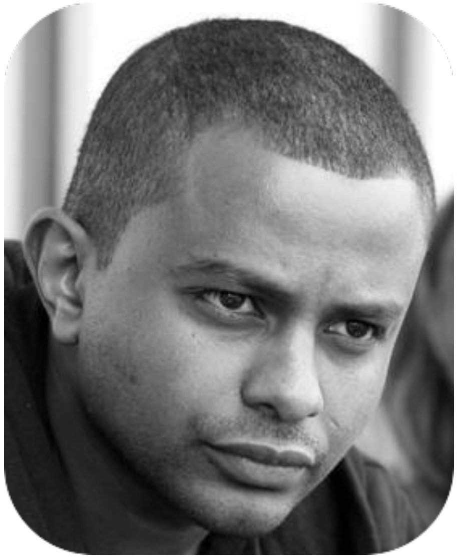
Automated Theorem Prover / Reasoning Engine

- discharges verification conditions
- general purpose constraint solver
- SAT, SMT, Abstract Interpreter, Temporal Logic Model Checker,…



http://seahorn.github.io

SeaHorn Verification Framework



Arie Gurfinkel
Software Engineering Institute
Carnegie Mellon University

Temesghen Kahsai
Carnegie Mellon University
NASA Ames

Jorge A. Navas
SGT
NASA Ames

The Plan

Introduction

Architecture and Usage

Demonstration

Constrained Horn Clauses as an Intermediate Representation

From Programs to Logic

- generating verification conditions

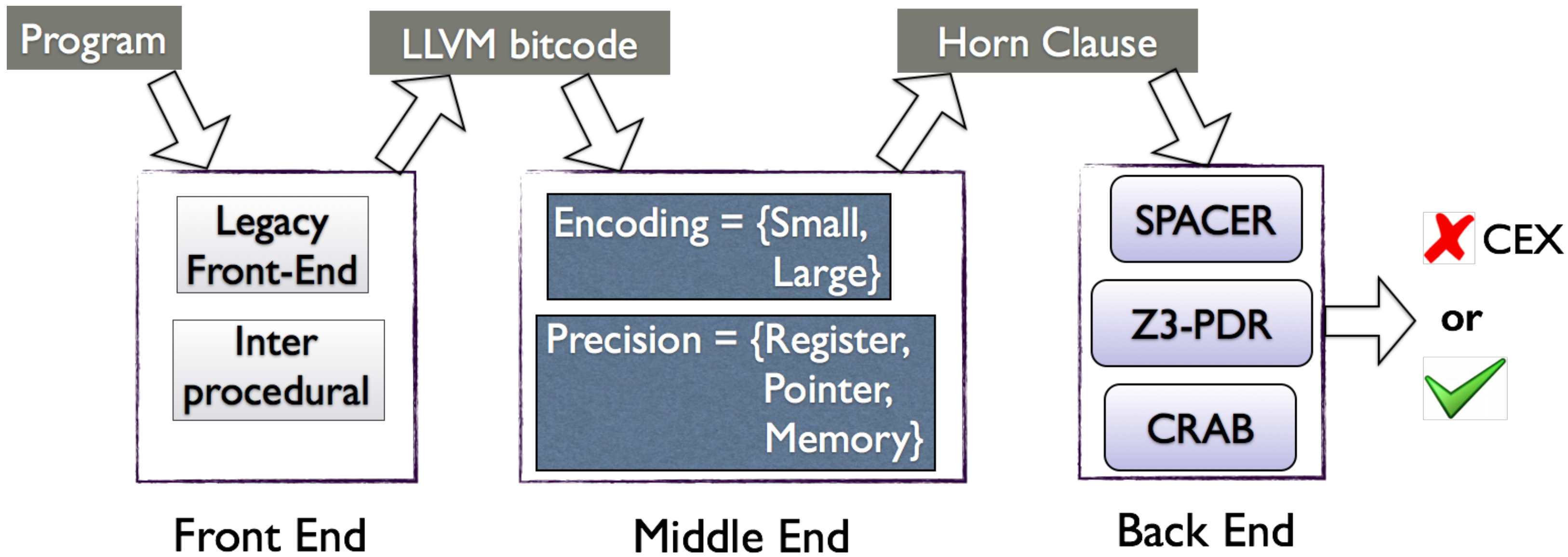
Program Transformations for Verification

Solving Constrained Horn Clauses

- synthesizing inductive invariants and procedure summaries

Conclusion

SeaHorn Verification Framework



Key Features

- LLVM front-end(s)
- Constrained Horn Clauses to represent Verification Conditions
- Comparable to state-of-the-art tools at SV-COMP'15

Goals

- be a state-of-the-art Software Model Checker
- be a framework for experimenting and developing CHC-based verification

Related Tools

CPAChecker

- Custom front-end for C
- Abstract Interpretation-inspired verification engine
- Predicate abstraction, invariant generation, BMC, k-induction

SMACK / Corral

- LLVM-based front-end
- Reduces C verification to Boogie
- Corral / Q verification back-end based on Bounded Model Checking with SMT

UFO

- LLVM-based front-end (partially reused in SeaHorn)
- Combines Abstract Interpretation with Interpolation-Based Model Checking
- (no longer actively developed)

SeaHorn Philosophy

Build a state-of-the-art Software Model Checker

- useful to "average" users
  - user-friendly, efficient, trusted, certificate-producing, …
- useful to researchers in verification
  - modular design, clean separation between syntax, semantics, and logic, …

Stand on the shoulders of giants

- reuse techniques from compiler community to reduce verification effort
  - SSA, loop restructuring, induction variables, alias analysis, …
  - static analysis and abstract interpretation
- reduce verification to logic
  - verification condition generation
  - Constrained Horn Clauses

Build reusable logic-based verification technology

- "SMT-LIB" for program verification

SeaHorn Usage

```
> sea pf FILE.c
```

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

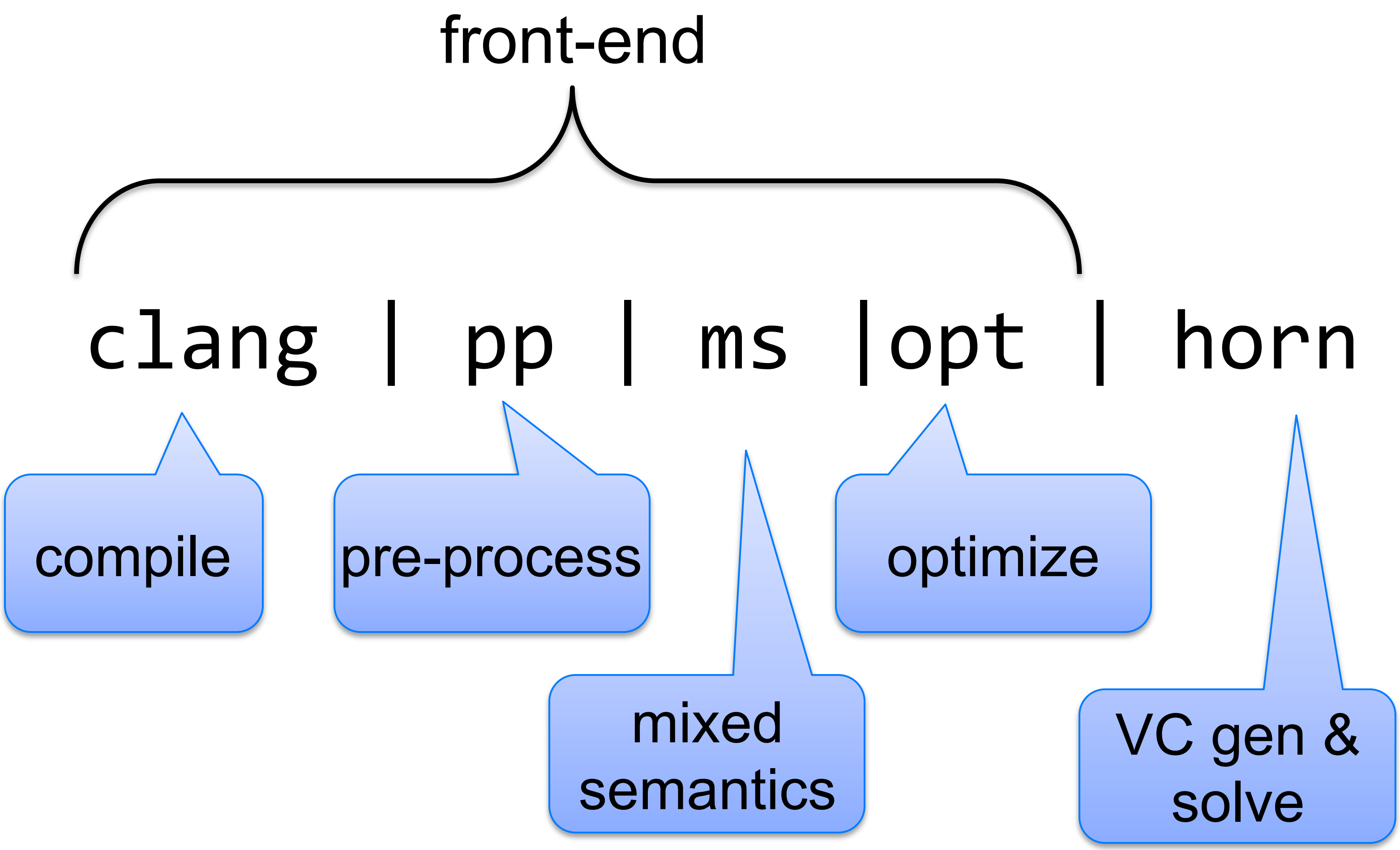
Additional options

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

Additional commands

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

Verification Pipeline



DEMO

From Programming to Modeling

Extend C programming language with 3 modeling features

Assertions

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

```
void assert (_Bool b) { if (!b) abort(); }
```

Non-determinism

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

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

Assumptions

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

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

Constrained Horn Clauses

INTERMEDIATE REPRESENTATION

Constrained Horn Clauses (CHC)

A Constrained Horn Clause (CHC) is a FOL formula of the form

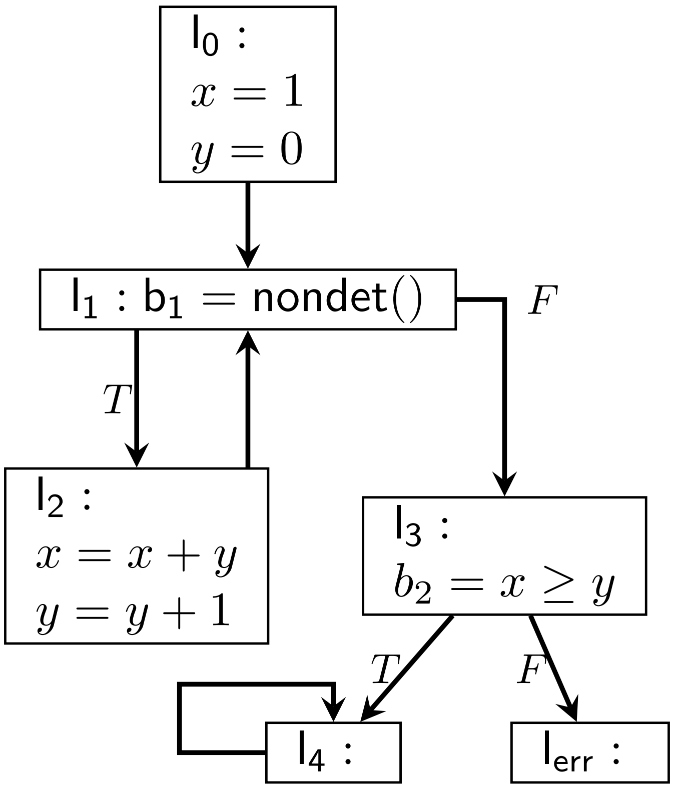
$$\forall V . (\phi \land p_1[X_1] \land \ldots \land p_n[X_n] \rightarrow h[X]),$$

where

- A is a background theory (e.g., Linear Arithmetic, Arrays, Bit-Vectors, or combinations of the above)
- ϕ is a constrained in the background theory A
- $p_1, \ldots, p_n$, h are n-ary predicates
- $p_i[X]$ is an application of a predicate to first-order terms

Example Horn Encoding

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



$\langle 1 \rangle$ p_0.

$\langle 2 \rangle$ $p_1(x, y) \leftarrow$
$p_0, x = 1, y = 0.$

$\langle 3 \rangle$ $p_2(x, y) \leftarrow p_1(x, y)$.

$\langle 4 \rangle$ $p_3(x, y) \leftarrow p_1(x, y)$.

$\langle 5 \rangle$ $p_1(x', y') \leftarrow$
$p_2(x, y),$
$x' = x + y,$
$y' = y + 1.$

$\langle 6 \rangle$ $p_4 \leftarrow (x \geq y), p_3(x, y).$

$\langle 7 \rangle$ $p_{err} \leftarrow (x < y), p_3(x, y).$

$\langle 8 \rangle$ $p_4 \leftarrow p_4.$

$\langle 9 \rangle$ $\bot \leftarrow p_{err}.$

CHC Terminology

head | body | constraint

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

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

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

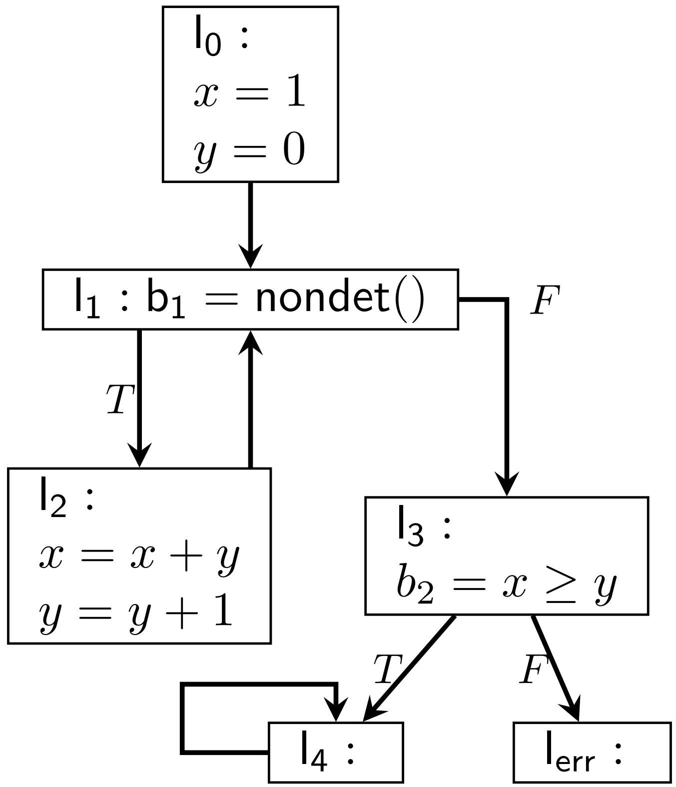
A **model** of a set of clauses Π is an interpretation of each predicate p_i that makes all clauses in Π valid

A set of clauses is **satisfiable** if it has a model, and is unsatisfiable otherwise

A model is **A-definable**, it each p_i is definable by a formula ψ_i in A

Example Horn Encoding

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



$$
\begin{array}{ll}
\langle 1 \rangle & p_0. \\
\langle 2 \rangle & p_1(x, y) \leftarrow \\
& \quad p_0, x = 1, y = 0. \\
\langle 3 \rangle & p_2(x, y) \leftarrow p_1(x, y)\ . \\
\langle 4 \rangle & p_3(x, y) \leftarrow p_1(x, y)\ . \\
\langle 5 \rangle & p_1(x', y') \leftarrow \\
& \quad p_2(x, y), \\
& \quad x' = x + y, \\
& \quad y' = y + 1. \\
\langle 6 \rangle & p_4 \leftarrow (x \geq y), p_3(x, y). \\
\langle 7 \rangle & p_{err} \leftarrow (x < y), p_3(x, y). \\
\langle 8 \rangle & p_4 \leftarrow p_4. \\
\langle 9 \rangle & \bot \leftarrow p_{err}.
\end{array}
$$

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

FROM PROGRAMS TO CLAUSES

Hoare Triples

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

Inductive Loop Invariant

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

Function Application

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

Recursion

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

Weakest Liberal Pre-Condition

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

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

wlp (P, Post)

weakest pre-condition ensuring that executing P ends in Post

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

A Simple Programming Language

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

body  ::= stmt (; stmt)*

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

E     := expression over program variables
```

Horn Clauses by Weakest Liberal Precondition

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

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

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

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

wlp (**while** E **do** S, Q) = $I(w) \wedge$
$\forall w \,.\, ((I(w) \wedge E) \rightarrow wlp(S, I(w))) \wedge ((I(w) \wedge \neg E) \rightarrow Q))$

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) $\wedge$ $\forall \{P \in Prog\}$. ToHorn (P)

Example of a WLP Horn Encoding

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

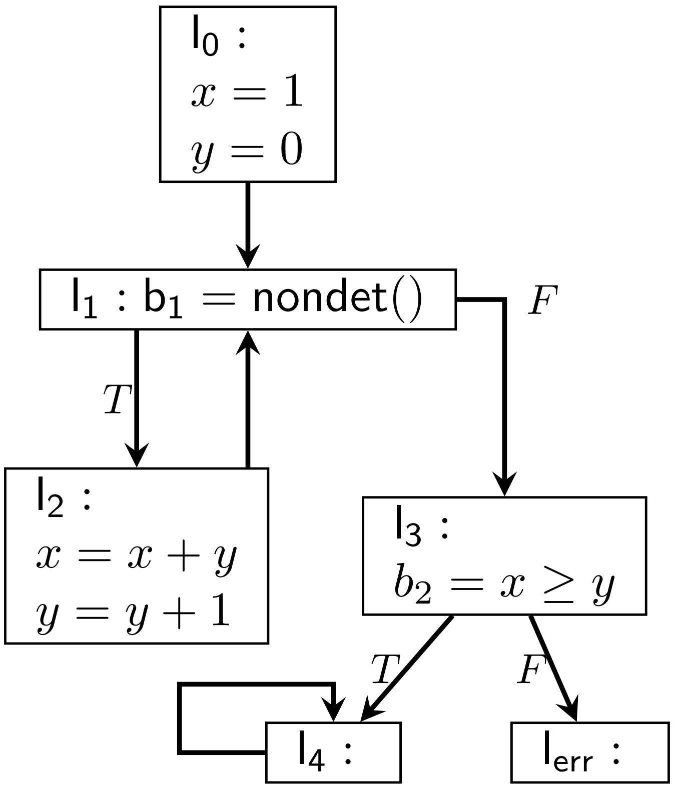
ToHorn

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

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

Example Horn Encoding

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



$\langle 1 \rangle\ \mathsf{p}_0.$

$\langle 2 \rangle\ \mathsf{p}_1(x, y) \leftarrow$
$\qquad \mathsf{p}_0, x = 1, y = 0.$

$\langle 3 \rangle\ \mathsf{p}_2(x, y) \leftarrow \mathsf{p}_1(x, y)\ .$

$\langle 4 \rangle\ \mathsf{p}_3(x, y) \leftarrow \mathsf{p}_1(x, y)\ .$

$\langle 5 \rangle\ \mathsf{p}_1(x', y') \leftarrow$
$\qquad \mathsf{p}_2(x, y),$
$\qquad x' = x + y,$
$\qquad y' = y + 1.$

$\langle 6 \rangle\ \mathsf{p}_4 \leftarrow (x \geq y), \mathsf{p}_3(x, y).$

$\langle 7 \rangle\ \mathsf{p}_{err} \leftarrow (x < y), \mathsf{p}_3(x, y).$

$\langle 8 \rangle\ \mathsf{p}_4 \leftarrow \mathsf{p}_4.$

$\langle 9 \rangle\ \bot \leftarrow \mathsf{p}_{err}.$

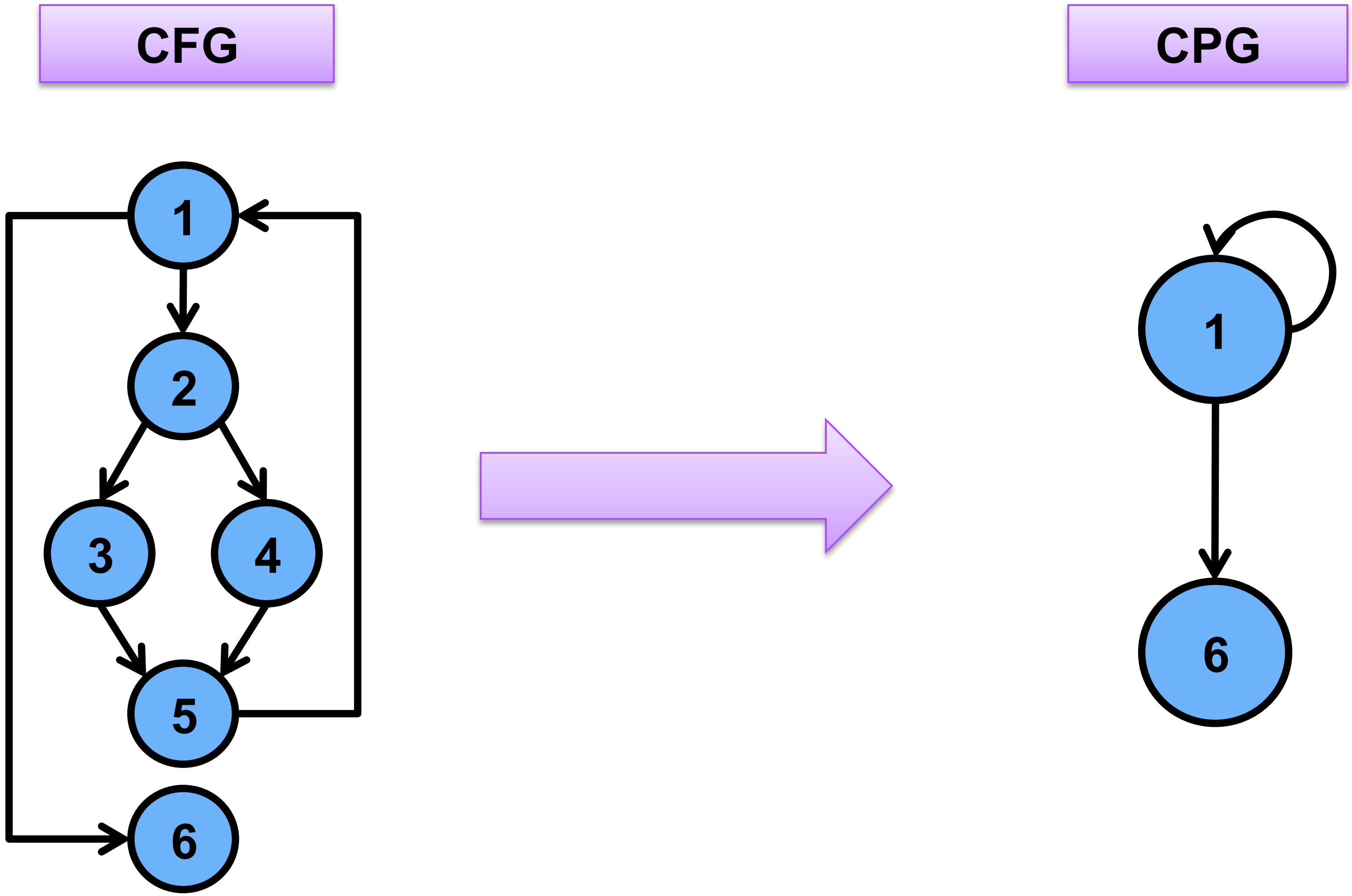
From CFG to Cut Point Graph

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

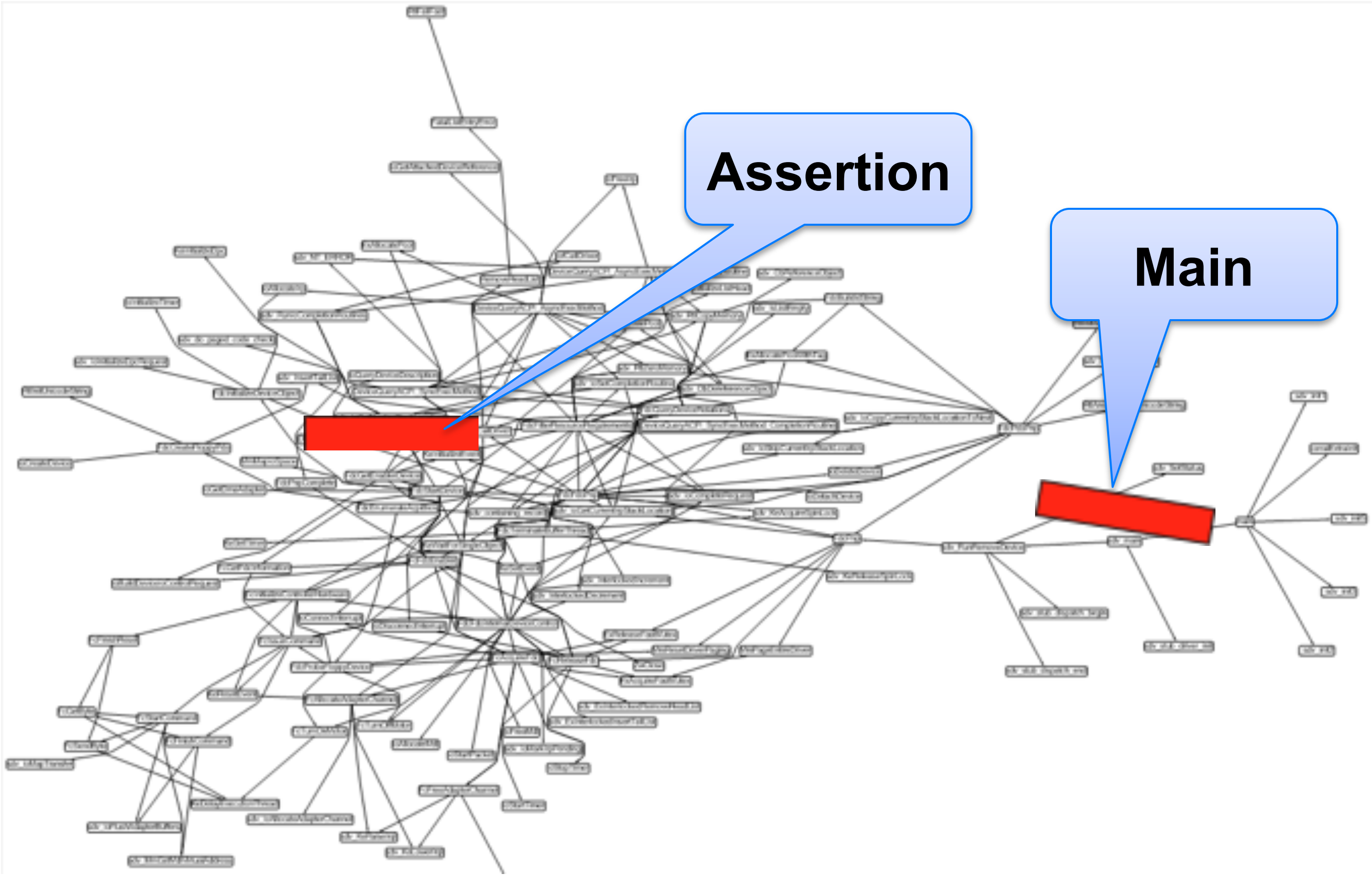
Cut Point Graph Example



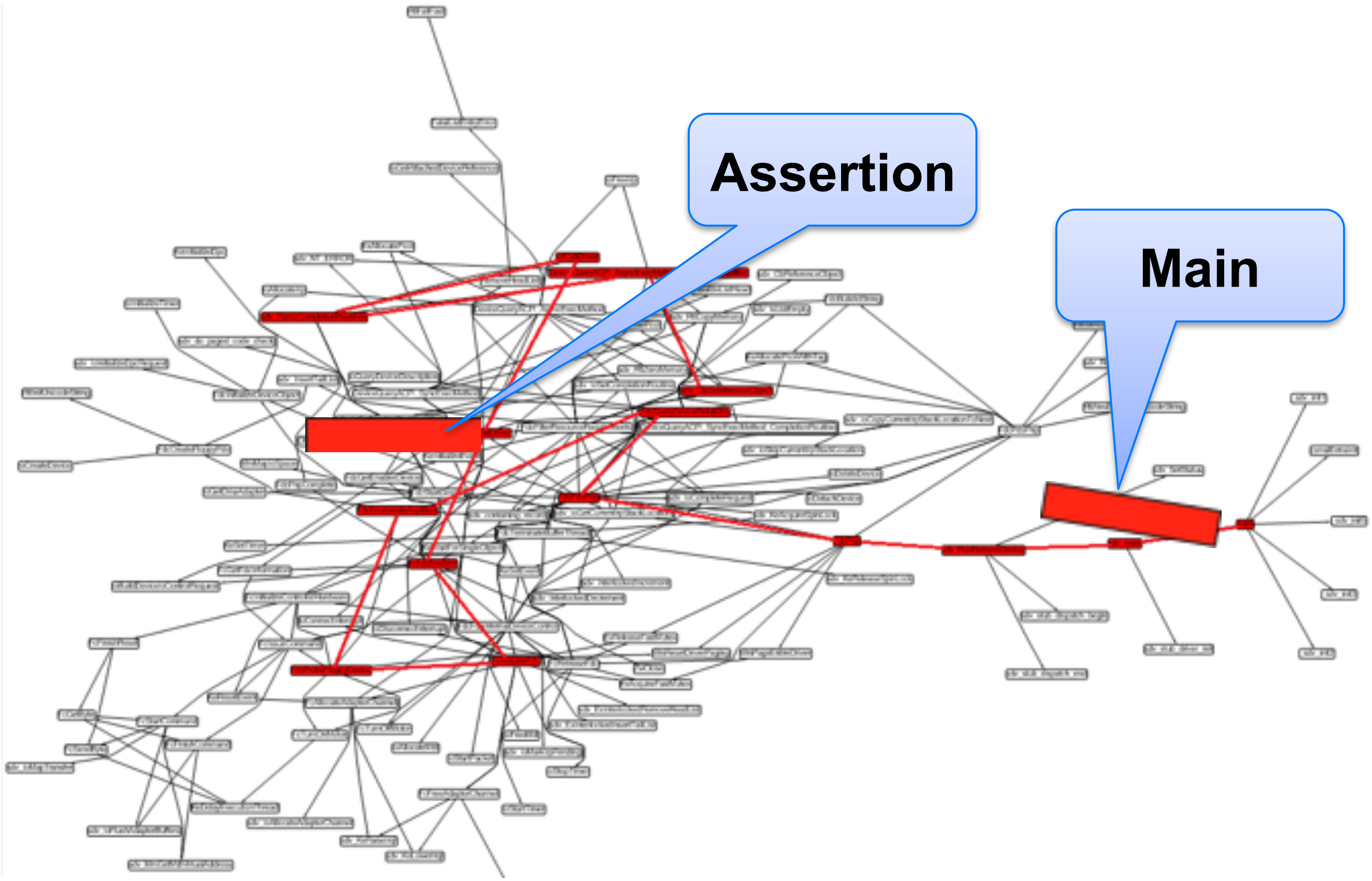
Mixed Semantics

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!)

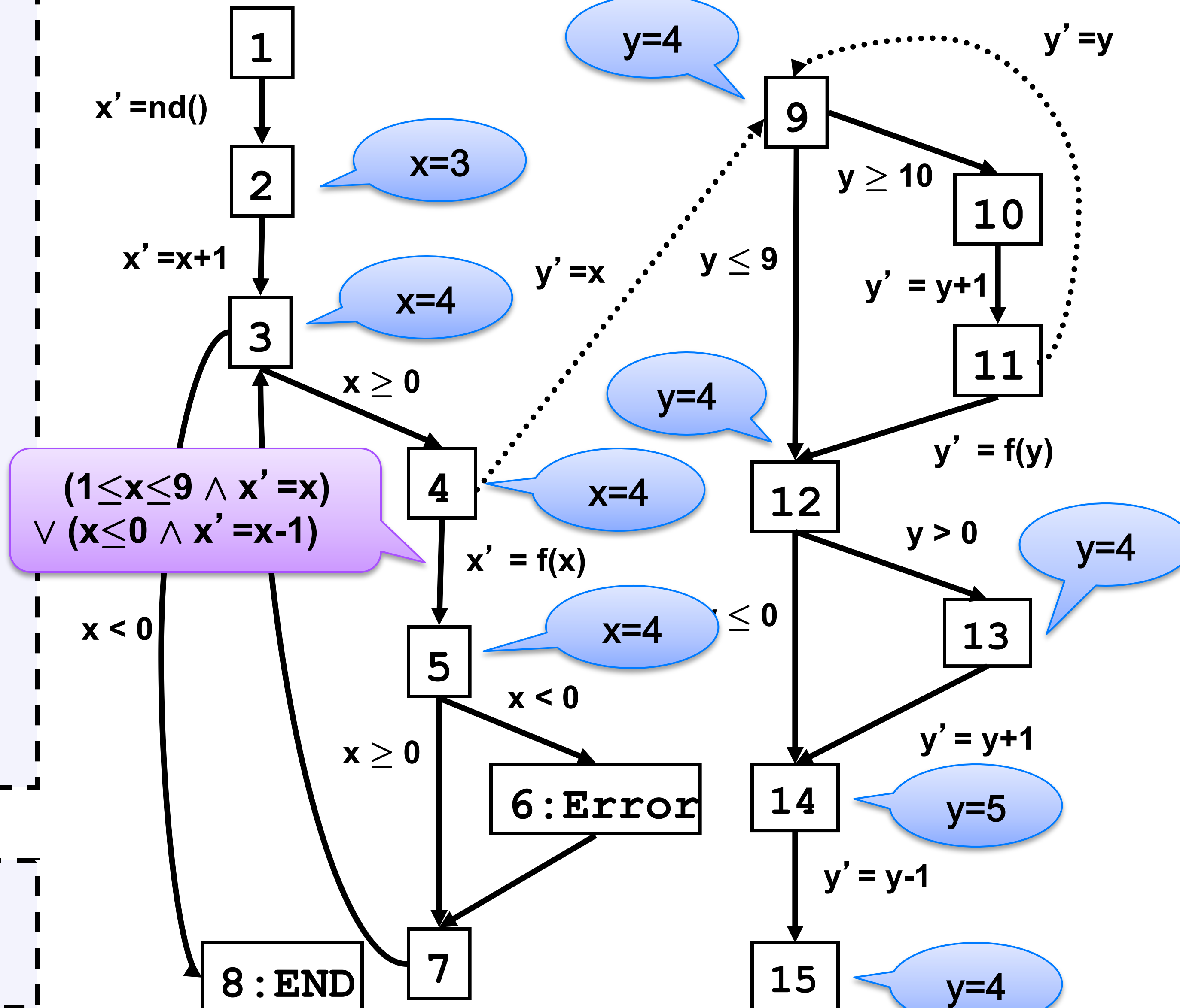
Preserves reachability and non-termination

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)

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

Mixed Semantics as Program Transformation

$main$ ()
  $p1$ (); $p1$ ();
  assert ($c1$);
$p1$ ()
  $p2$ ();
  assert ($c2$);
$p2$ ()
  assert ($c3$);

Mixed Semantics ↓

$main_{new}$ ()
  if (*) goto $p1_{entry}$;
  else $p1_{new}$ ();
  if (*) goto $p1_{entry}$;
  else $p1_{new}$ ();
  if ($\neg c1$) goto $error$;
  assume (false);

$p1_{entry}$:
  if (*) goto $p2_{entry}$;
  else $p2_{new}$ ();
  if ($\neg c2$) goto $error$;
$p2_{entry}$:
  if ($\neg c3$) goto $error$;
  assume (false);
$error$: assert (false);

$p1_{new}$ ()
  $p2_{new}$ ();
  assume ($c2$);
$p2_{new}$ ()
  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

SOLVING CHC WITH SMT

Programs, Cexs, Invariants

A program *P = (V, Init, ρ, Bad)*

- Notation: $\mathcal{F}(X) = \exists \boldsymbol{u} . (X \wedge \rho) \vee Init$

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

$$Init(v_0) \wedge \left(\bigwedge_{i=0}^{N-1} \rho(v_i, v_{i+1}) \right) \wedge Bad(v_N) \not\Rightarrow \bot$$

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

$$Init(u) \Rightarrow Inv(u)$$

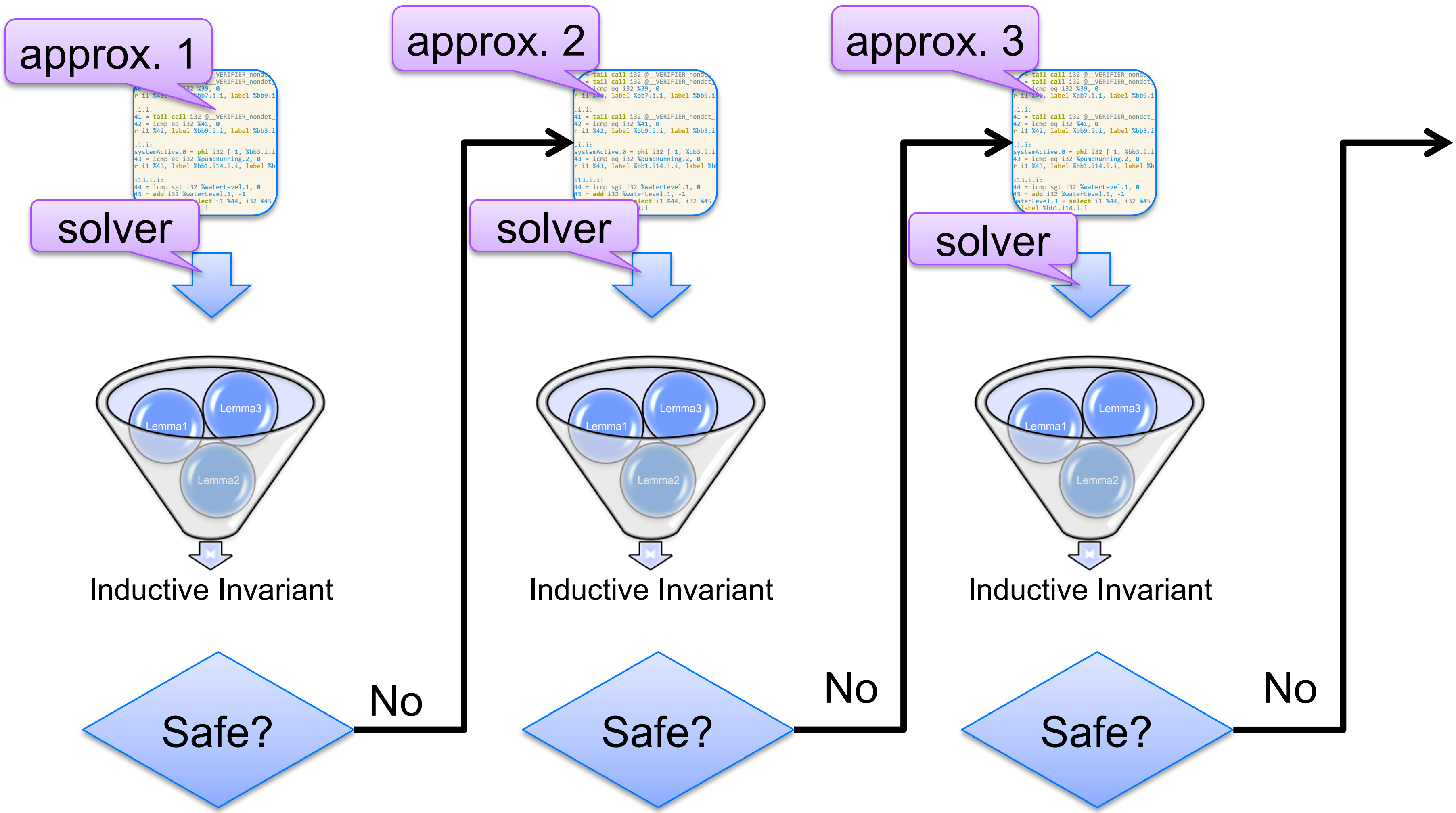
$$Inv(u) \wedge \rho(u, v) \Rightarrow Inv(v)$$

} Inductive

$$Inv(u) \Rightarrow \neg Bad(u)$$

Safe

Verification by Evolving Approximations



IC3/PDR Algorithm Overview

bounded safety

Input: Safety problem $\langle Init(X), Tr(X, X'), Bad(X) \rangle$

$F_0 \leftarrow Init\,; N \leftarrow 0$ **repeat**

$\quad \mathbf{G} \leftarrow \textsc{PdrMkSafe}([F_0, \ldots, F_N], Bad)$

$\quad$ **if** $\mathbf{G} = [\,]$ **then return** $Reachable$;

$\quad \forall 0 \leq i \leq N \cdot F_i \leftarrow \mathbf{G}[i]$

$\quad F_0, \ldots, F_N \leftarrow \textsc{PdrPush}([F_0, \ldots, F_N])$

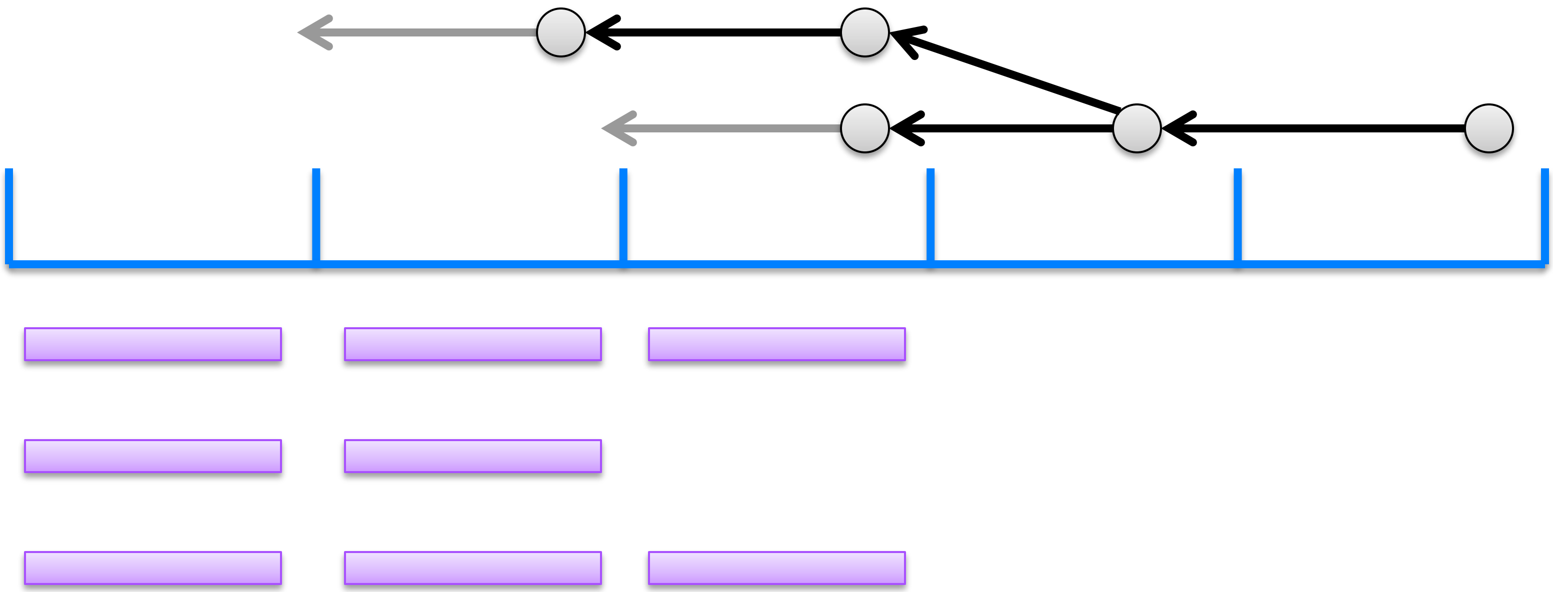
$\quad$ **if** $\exists 0 \leq i < N \cdot F_i = F_{i+1}$ **then return** $Unreachable$;

$\quad N \leftarrow N + 1\,; F_N \leftarrow \emptyset$

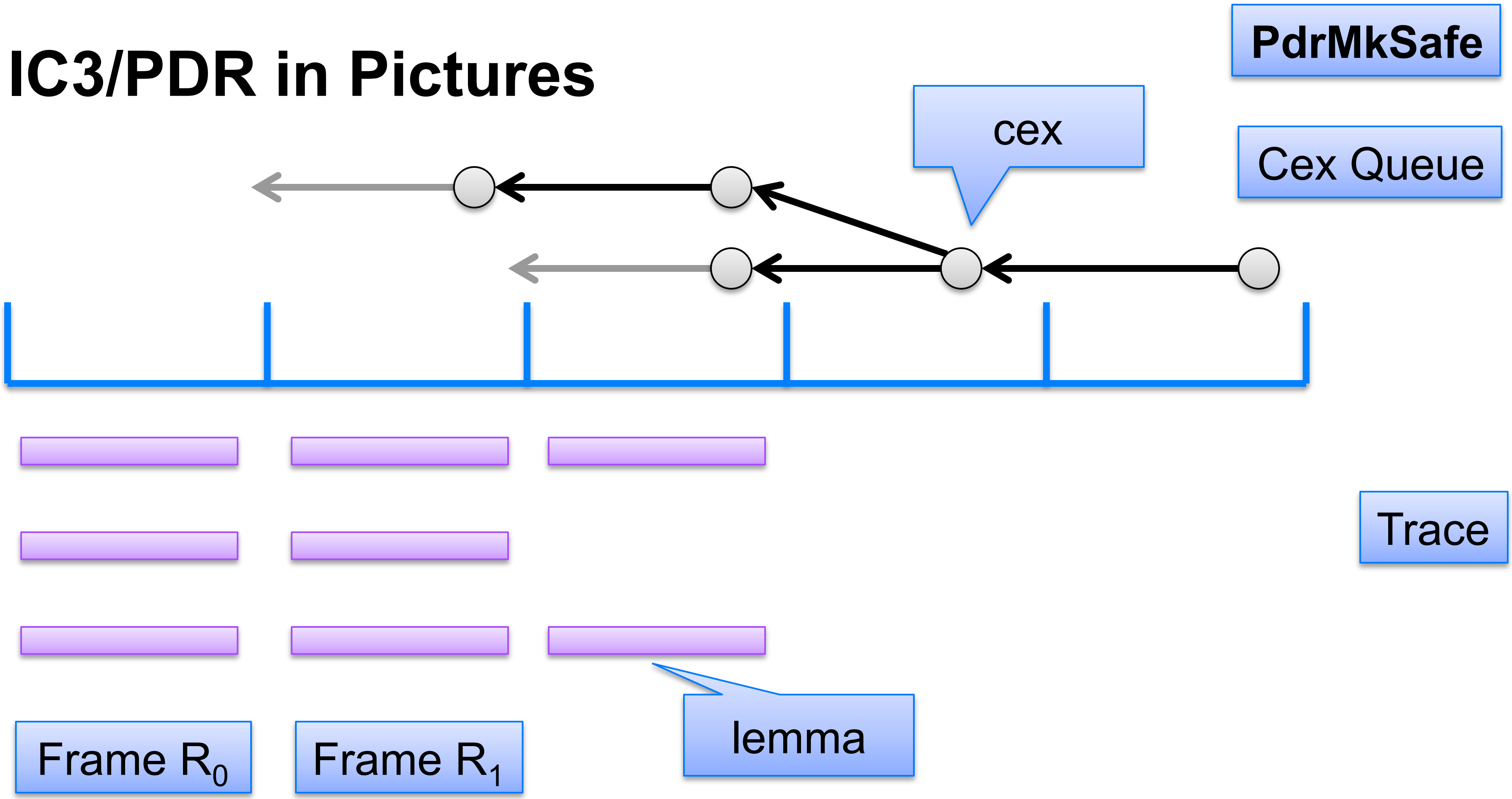
until ∞;

strengthen result

IC3/PDR in Pictures

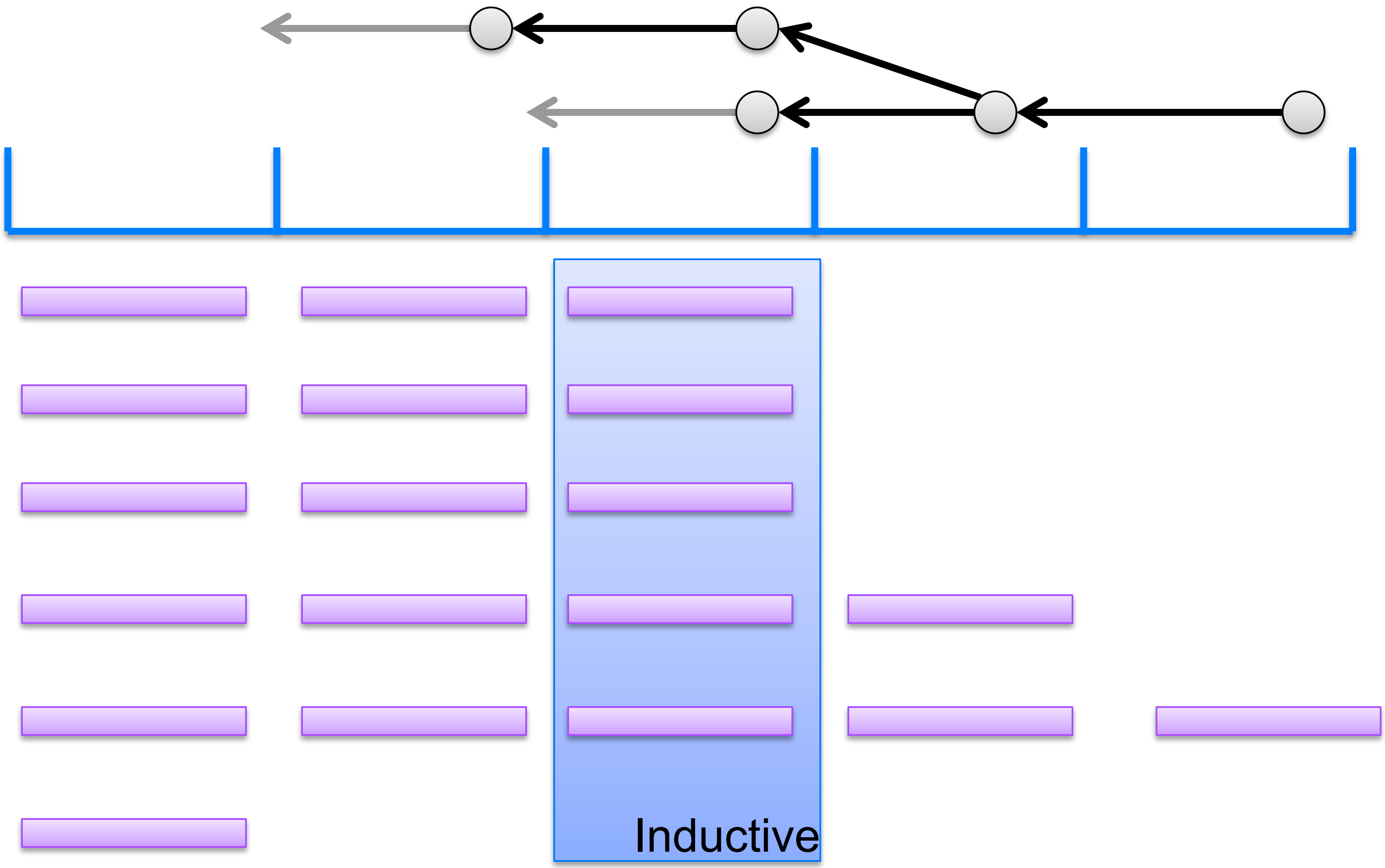


IC3/PDR in Pictures



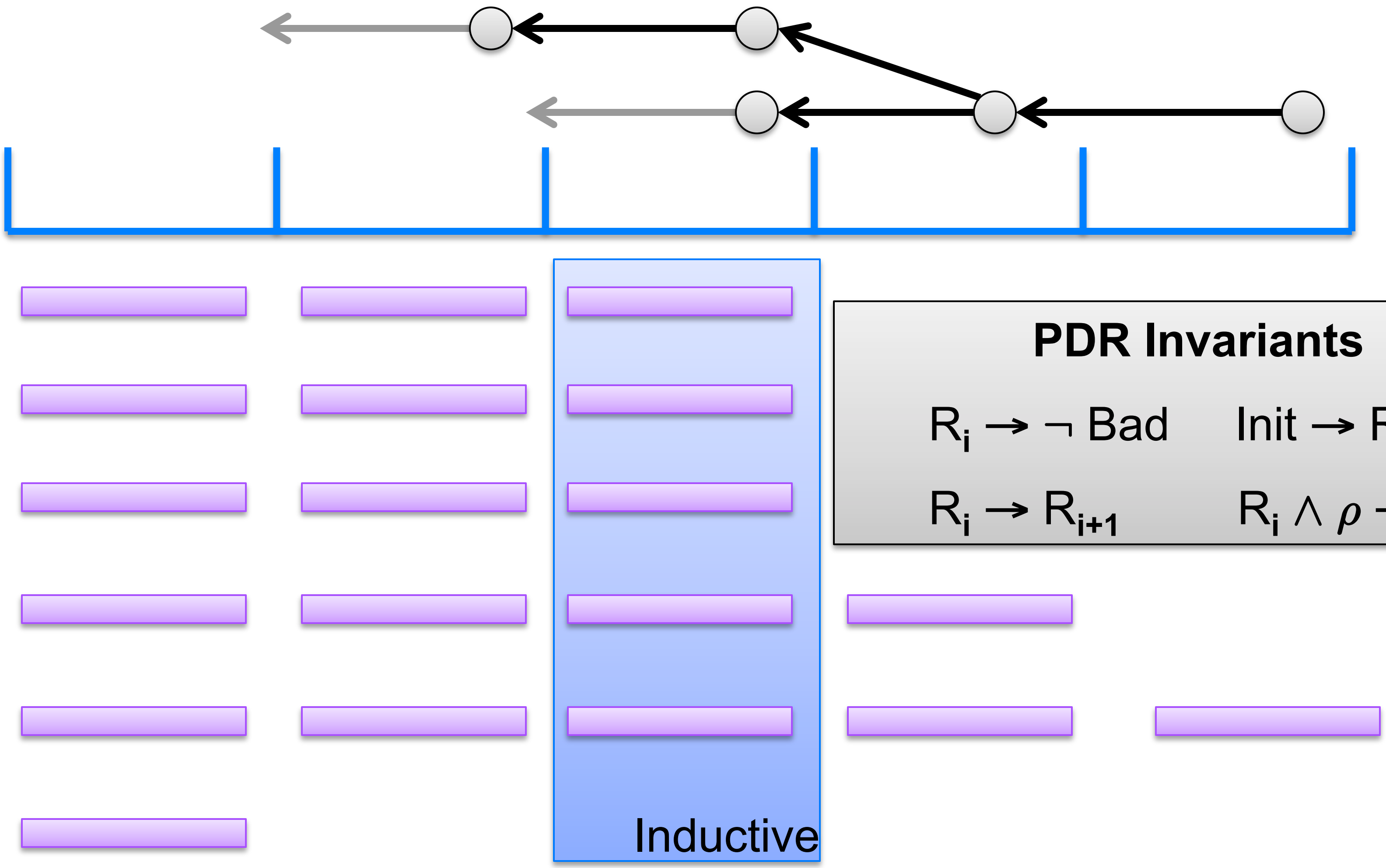
IC3/PDR in Pictures

PdrPush



IC3/PDR in Pictures

PdrPush

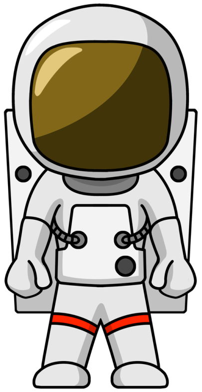


PDR Invariants

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

$R_i \rightarrow R_{i+1}$ $\quad$ $R_i \wedge \rho \rightarrow R_{i+1}$

Spacer: Solving CHC in Z3



Spacer: solver for SMT-constrained Horn Clauses

- stand-alone implementation in a fork of Z3
- http://bitbucket.org/spacer/code

Support for Non-Linear CHC

- model procedure summaries in inter-procedural verification conditions
- model assume-guarantee reasoning
- uses MBP to under-approximate models for finite unfoldings of predicates
- uses MAX-SAT to decide on an unfolding strategy

Supported SMT-Theories

- Best-effort support for arbitrary SMT-theories
  - data-structures, bit-vectors, non-linear arithmetic
- Full support for Linear arithmetic (rational and integer)
- Quantifier-free theory of arrays
  - only quantifier free models with limited applications of array equality

CRAB: Cornucopia of Abstractions

A library of abstract domains build on top of NASA Ikos (Inference Kernel for Open Static Analyzers)

A language-independent intermediate representation

Many abstract domains

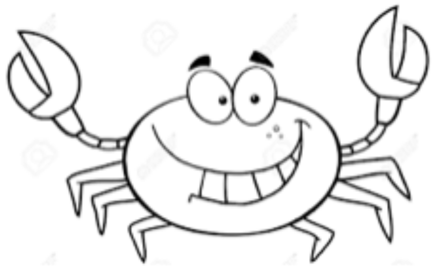
- intervals (with congruences) (with uninterpreted functions)
- zones, dbms, octagons
- pointer analysis with offsets
- array analysis with smashing

Fixpoint iteration library

- precise interleaving between widening and narrowing
- extensible with thresholds

Efficient reusable data-structure

- simple API for integrating new abstract domains



RESULTS

SV-COMP 2015

http://sv-comp.sosy-lab.org/2015/

4th Competition on Software Verification held (here!) at TACAS 2015

Goals

- Provide a snapshot of the state-of-the-art in software verification to the community.
- Increase the visibility and credits that tool developers receive.
- Establish a set of benchmarks for software verification in the community.

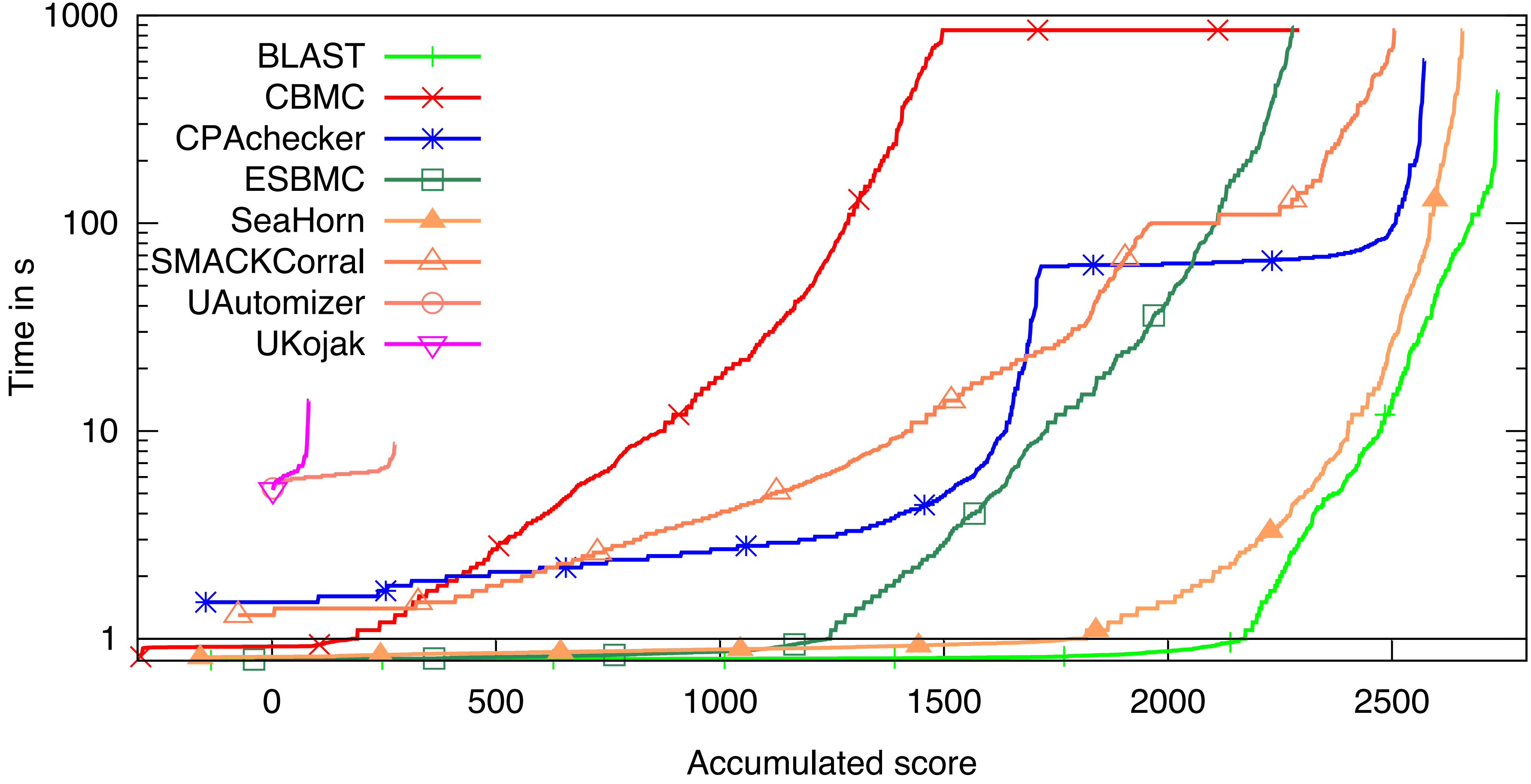
Participants:

- Over 22 participants, including most popular Software Model Checkers and Bounded Model Checkers

Benchmarks:

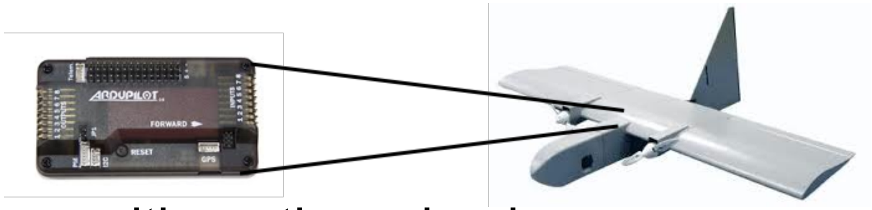
- C programs with error location (programs include pointers, structures, etc.)
- Over 6,000 files, each 2K – 100K LOC
- Linux Device Drivers, Product Lines, Regressions/Tricky examples
- http://sv-comp.sosy-lab.org/2015/benchmarks.php

Results for DeviceDriver category



Detecting Buffer Overflow in Auto-pilot software

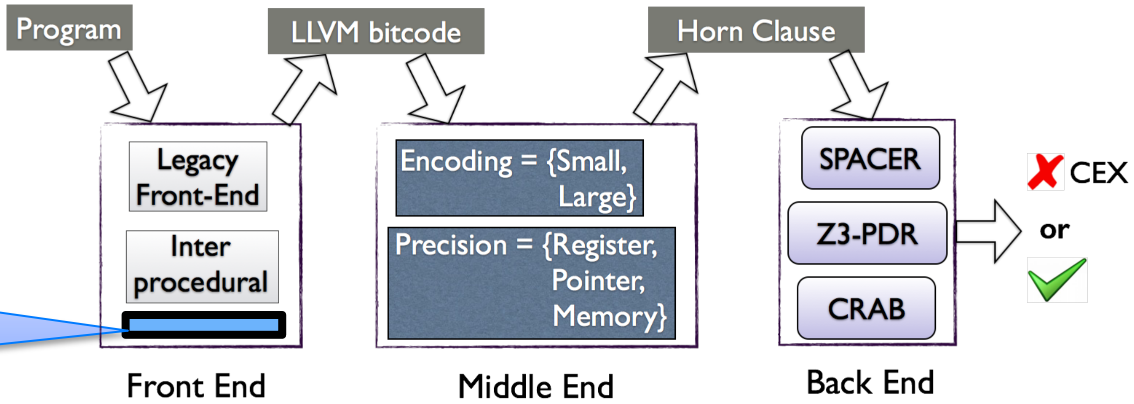
Show absence of Buffer Overflows in
- paparazzi and mnav autopilots



Automatically instrument buffer accesses with runtime checks

Use SeaHorn to validate that run-time checks never fail
- somewhat slower than pure abstract interpretation
- much more precise!

LLVM Pass to insert BO checks



Conclusion

SeaHorn (http://seahorn.github.io)

- a state-of-the-art Software Model Checker
- LLVM-based front-end
- CHC-based verification engine
- a framework for research in logic-based verification

The future

- making SeaHorn useful to users of verification technology
  - counterexamples, build integration, property specification, proofs, etc.
- targeting many existing CHC engines
  - specialize encoding and transformations to specific engines
  - communicate results between engines
- richer properties
  - termination, liveness, synthesis

Available postdoctoral positions

What: development and application of SeaHorn

Where: CMU/NASA Silicon Valley Campus

Contact:

Temegshen Kahsai
temesghen.kahsaiazene@nasa.gov

Arie Gurfinkel arie@cmu.edu

Contact Information

Arie Gurfinkel, Ph. D.
Sr. Researcher
CSC/SSD
Telephone: +1 412-268-5800
Email: info@sei.cmu.edu

U.S. Mail
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
USA

Web
www.sei.cmu.edu
www.sei.cmu.edu/contact.cfm

Customer Relations
Email: info@sei.cmu.edu
Telephone: +1 412-268-5800
SEI Phone: +1 412-268-5800
SEI Fax: +1 412-268-6257