Building Program Verifiers from Compilers and Theorem Provers

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

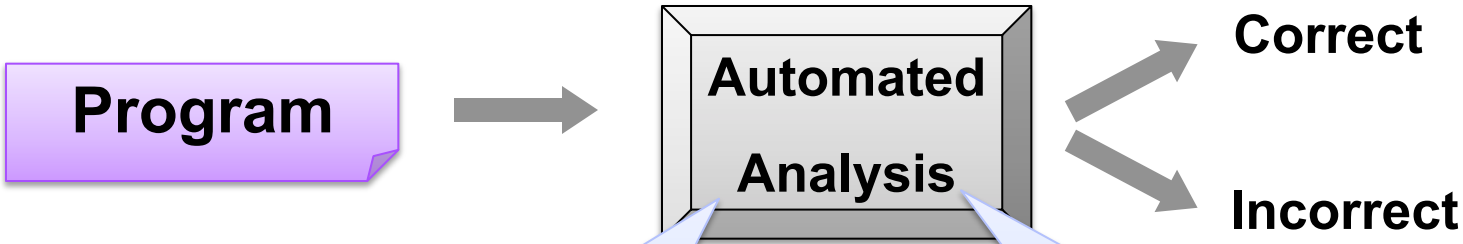
Arie Gurfinkel

based on joint work with Teme Kahsai, Jorge A. Navas, Anvesh Komuravelli, and Nikolaj Bjorner

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0002433

Automated Software Analysis

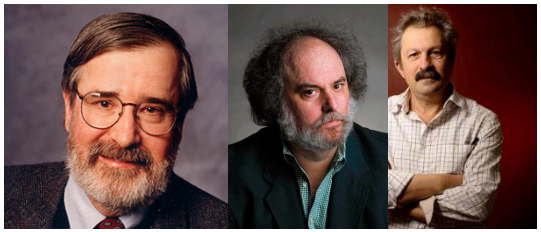


Software Model Checking with Predicate Abstraction

e.g., Microsoft's SDV

Abstract Interpretation with Numeric Abstraction

e.g., ASTREE, Polyspace



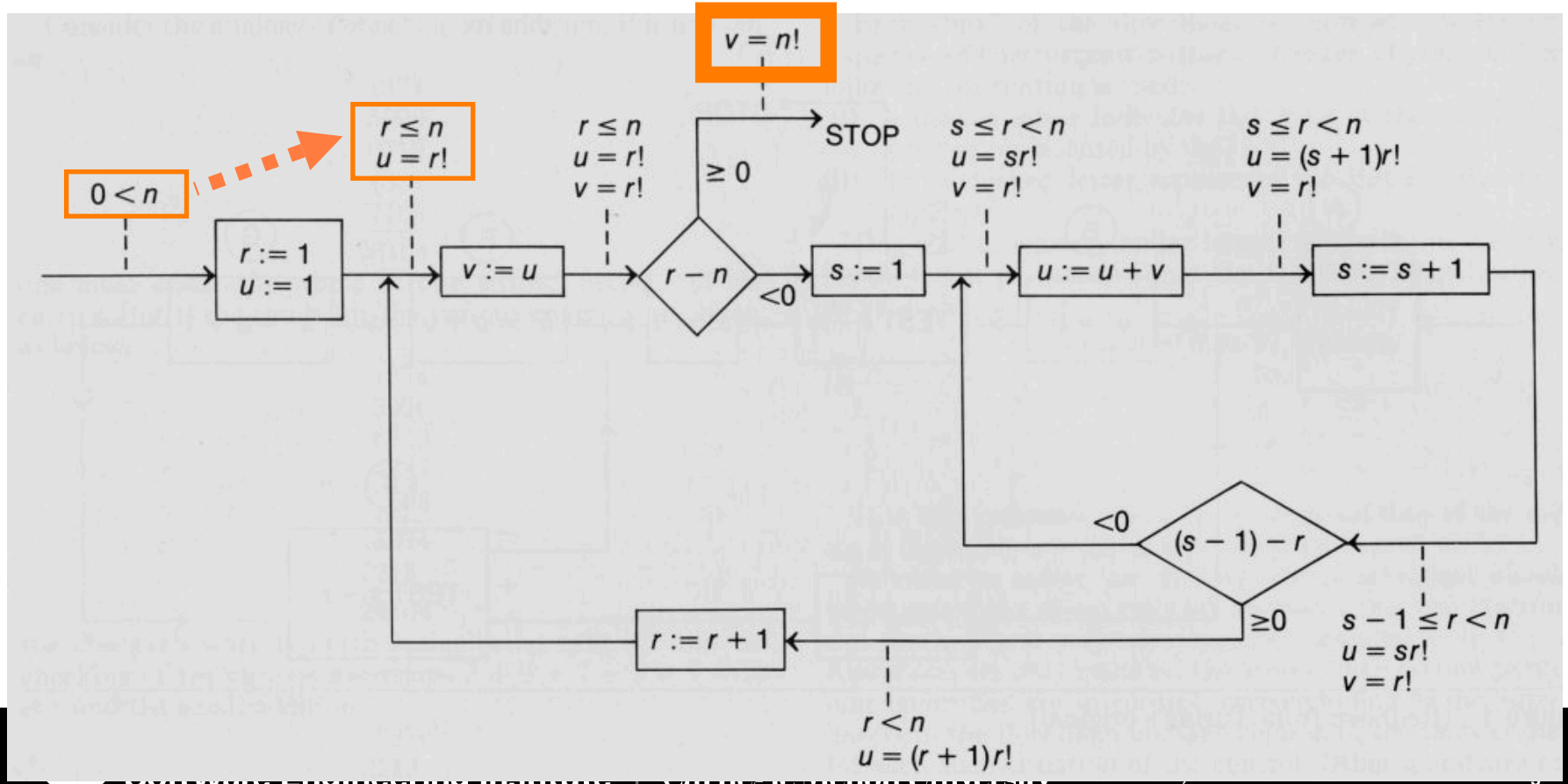
Turing, 1936: “undecidable”

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?

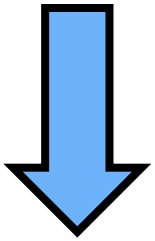
programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.



Three-Layers of a Program Verifier

Compiler

- compiles surface syntax to some machine
- embodies syntax with semantics

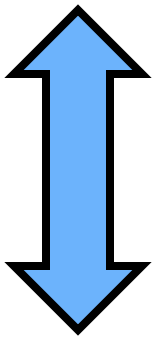


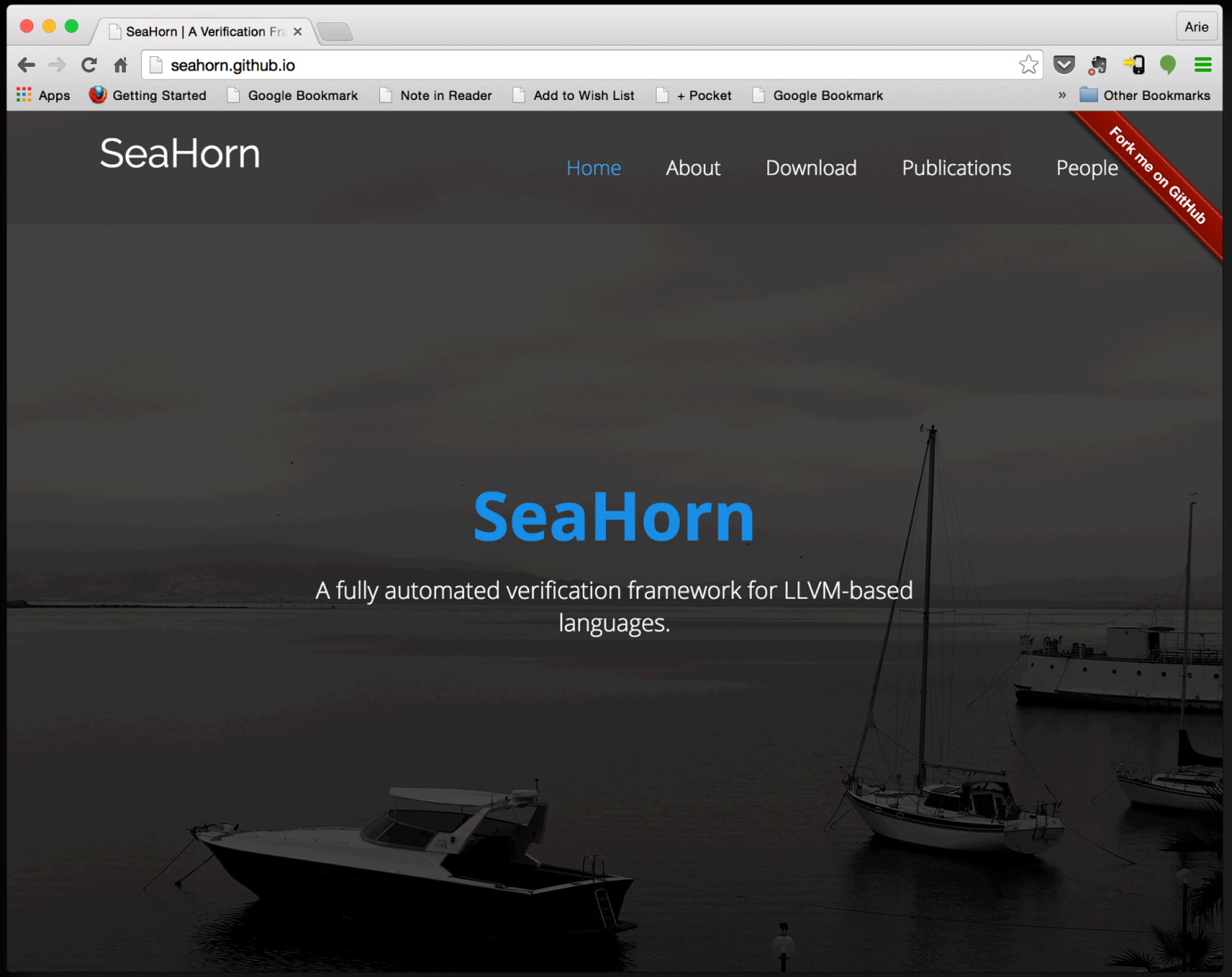
Verification Condition Generator

- transforms a program and a property to 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 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

The Plan

This Lecture

- front-end: verification conditions, constrained horn clauses
- from verification problems to decision problems in logic

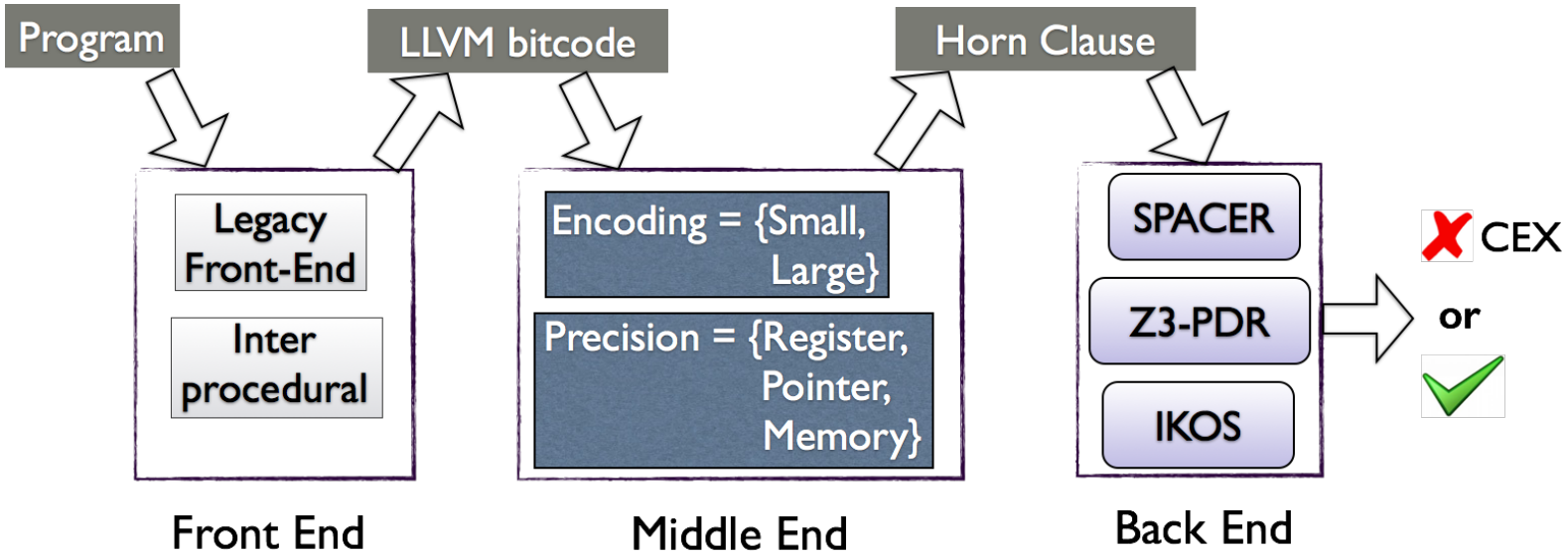
Next Lecture

- back-end: solving verification conditions
- IC3/PDR algorithms and extensions for software verification
- http://arieg.bitbucket.org/pdf/gurfinkel_ssft15.pdf

Labs

- we will play with tools
- get binary distribution from github.com/seahorn/seahorn
- get seahorn-tutorial repo from github.com/seahorn-tutorial
- pre-requisites: recent Linux or OSX + clang-3.6
- can use provided VM to satisfy the pre-requisites

SeaHorn Verification Framework



Distinguishing 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 Usage

`> sea pf FILE.c`

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

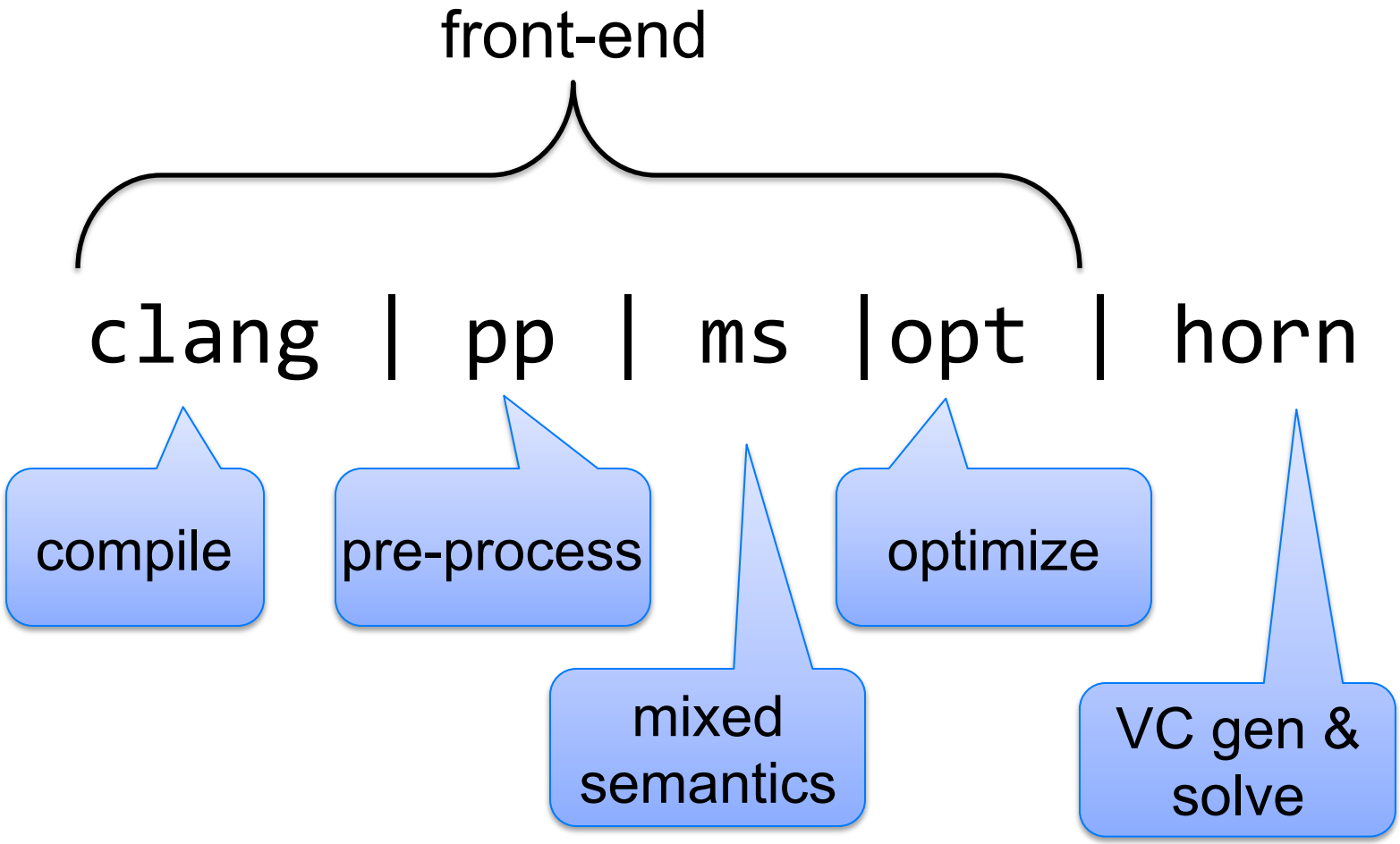
Additional options

- `--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



Constrained Horn Clauses

INTERMEDIATE REPRESENTATION

Constrained Horn Clauses (CHC)

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

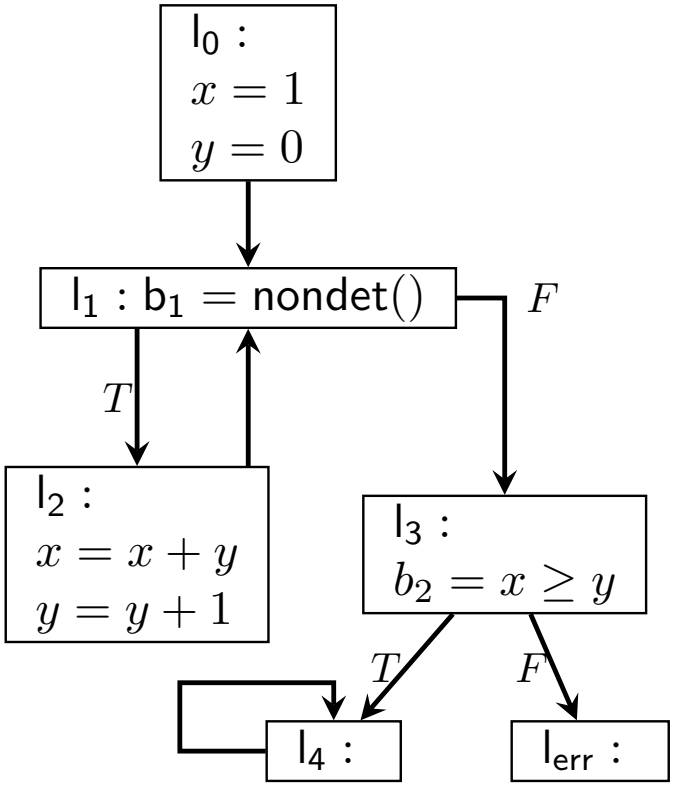
$$\forall V . (\phi \wedge p_1[X_1] \wedge \ldots \wedge p_n[X_n] \rightarrow h[X]),$$

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

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



$\langle 1 \rangle$ p_0.

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

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

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

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

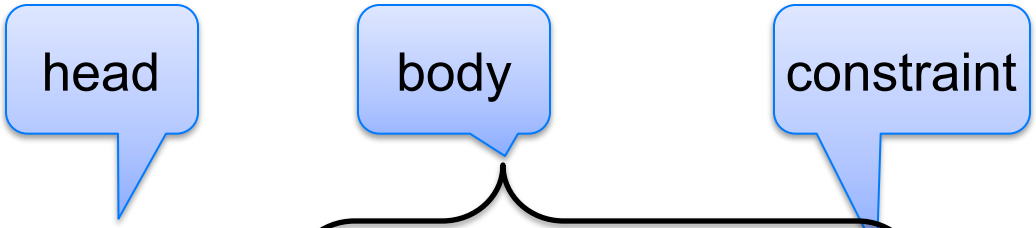
$\langle 6 \rangle$ $\mathsf{p}_4 \leftarrow (x \geq y), \mathsf{p}_3(x, y)$.

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

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

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

CHC Terminology



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

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 **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

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

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

Function Application

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

Recursion

$$\frac{\{\text{Pre}\}\ \text{b = F(a)}\ \{\text{Post}\}\ \vdash \{\text{Pre}\}\ \text{Body}_{\text{F}}\ \{\text{Post}\}}{\{\text{Pre}\}\ \text{b = F(a)}\ \{\text{Post}\}}$$

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))

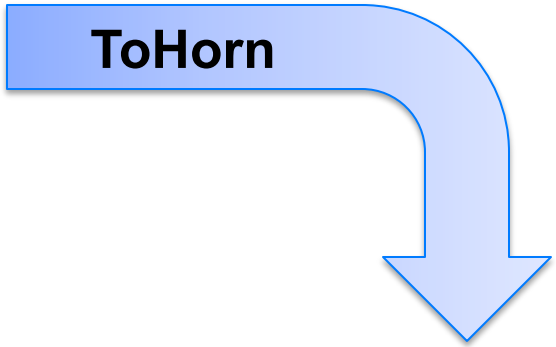
wlp (y = P(E), Q) = p_{pre}(E) $\wedge$ ($\forall$ r. p(E, r) $\rightarrow$ Q[r/y])

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




```
C1: I(x,y,x,y) ← y>=0.
C2: I(x+1,y-1,x_o,y_o) ← I(x,y,x_o,y_o), y>0.
C3: false ← I(x,y,x_o,y_o), y≤0, x≠x_o+y_o
```

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

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

Control Flow Graph

A CFG is a graph of basic blocks

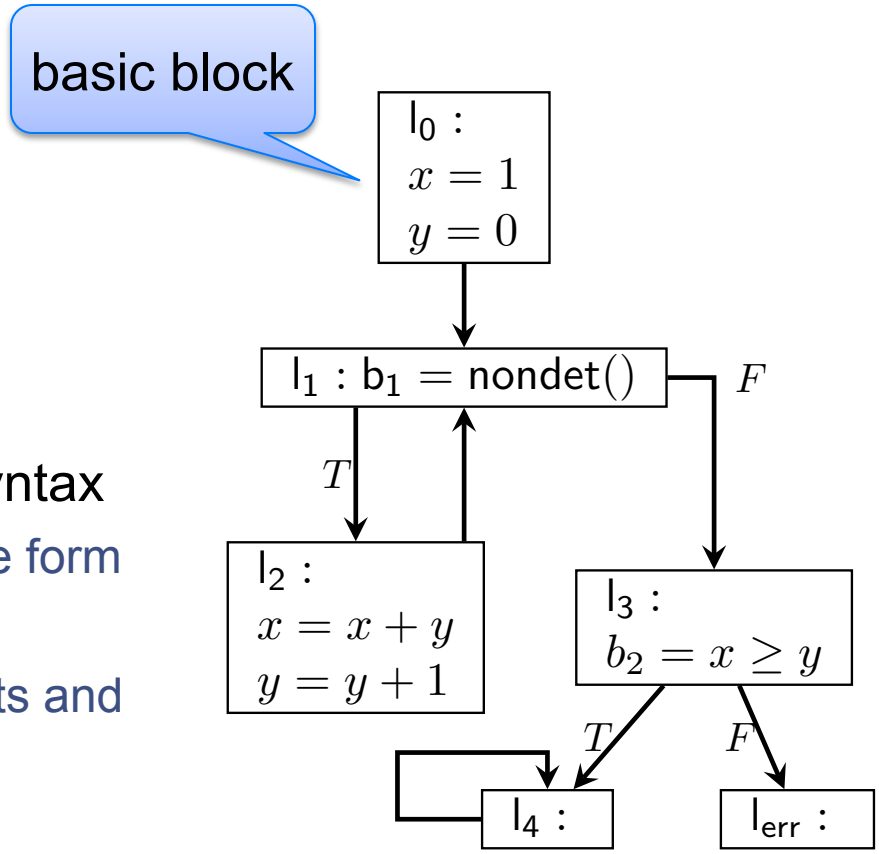
- edges represent different control flow

A CFG corresponds to a program syntax

- where statements are restricted to the form

  L_i:S ; goto L_j

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



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, $p_{en}(x_0,x,w)$ for entry, $p_{ex}(x_0,r)$ for exit

The verification condition is a conjunction of clauses:

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

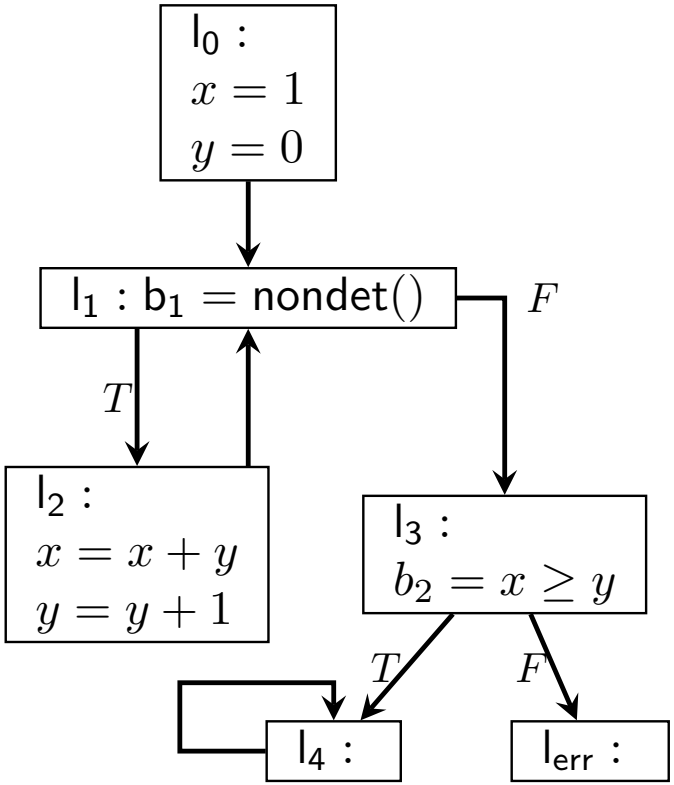
$l_i(x_0,w') \leftarrow l_j(x_0,w) \wedge \neg wlp(S, \neg(w=w'))$, for each statement l_i: S; goto l_j

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

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

Example Horn Encoding

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



$\langle 1 \rangle$ p_0.

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

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

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

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

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

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

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

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

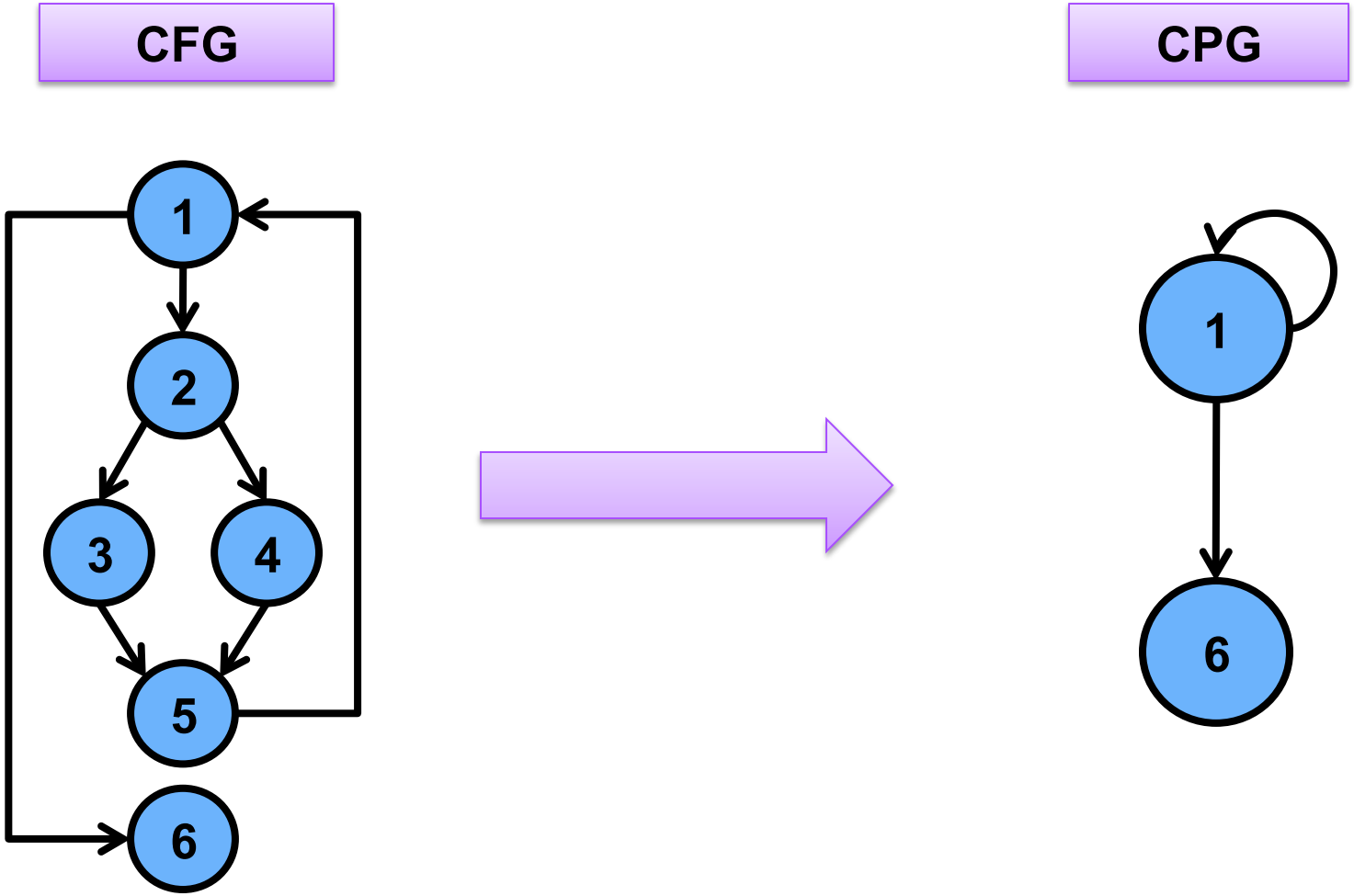
From CFG to Cut Point Graph

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

$$x = PHI (v_0:bb_0, \ldots, v_n:bb_n)) \qquad \text{(phi-assignment)}$$

"x gets v_i if previously executed block was bb_i"

Single Static Assignment: An Example

```
int x, y, n;

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

```
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:
```

val:bb

Large Step Encoding

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

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

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

3: x_1 = x_0 + y_0; goto 5

4: x_2 = x_0 – y_0; goto 5

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

Large Step Encoding: Extract all Actions

$x_1 = x_0 + y_0$
$x_2 = x_0 - y_0$
$y_1 = -1 * y_0$

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

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

3: x_1 = x_0 + y_0  goto 5

4: x_2 = x_0 - y_0  goto 5

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

Example: Encode Control Flow

$x_1 = x_0 + y_0$
$x_2 = x_0 - y_0$
$y_1 = -1 * y_0$

$B_2 \rightarrow x_0 < N$

$B_3 \rightarrow B_2 \wedge y_0 > 0$

$B_4 \rightarrow B_2 \wedge y_0 \leq 0$
$B_5 \rightarrow (B_3 \wedge x_3 = x_1) \vee$
$(B_4 \wedge x_3 = x_2)$

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

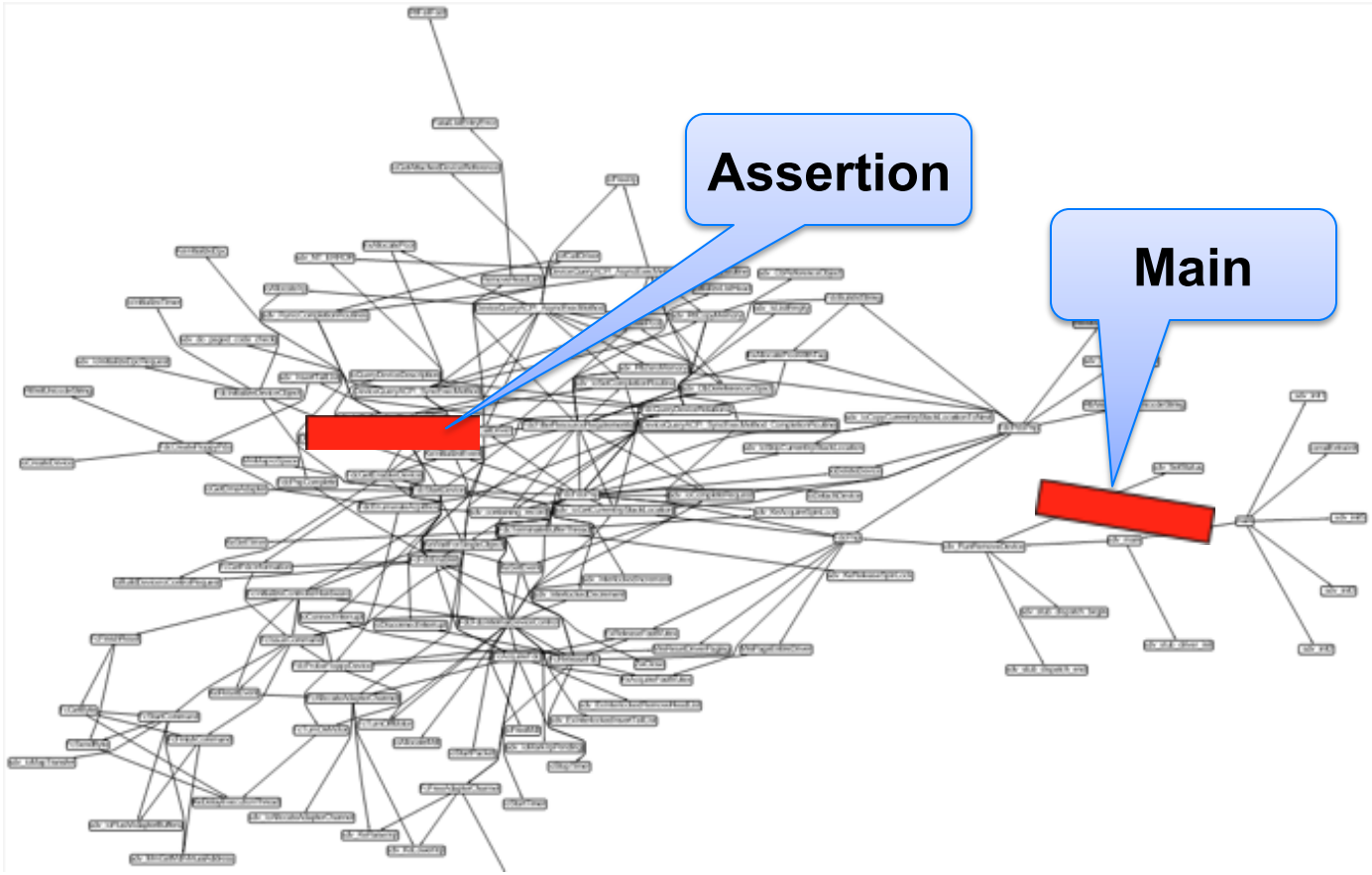
$p_1(x'_0, y'_0) \leftarrow p_1(x_0, y_0), \phi.$

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

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

3: x_1 = x_0 + y_0; goto 5

4: x_2 = x_0 - y_0; goto 5

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

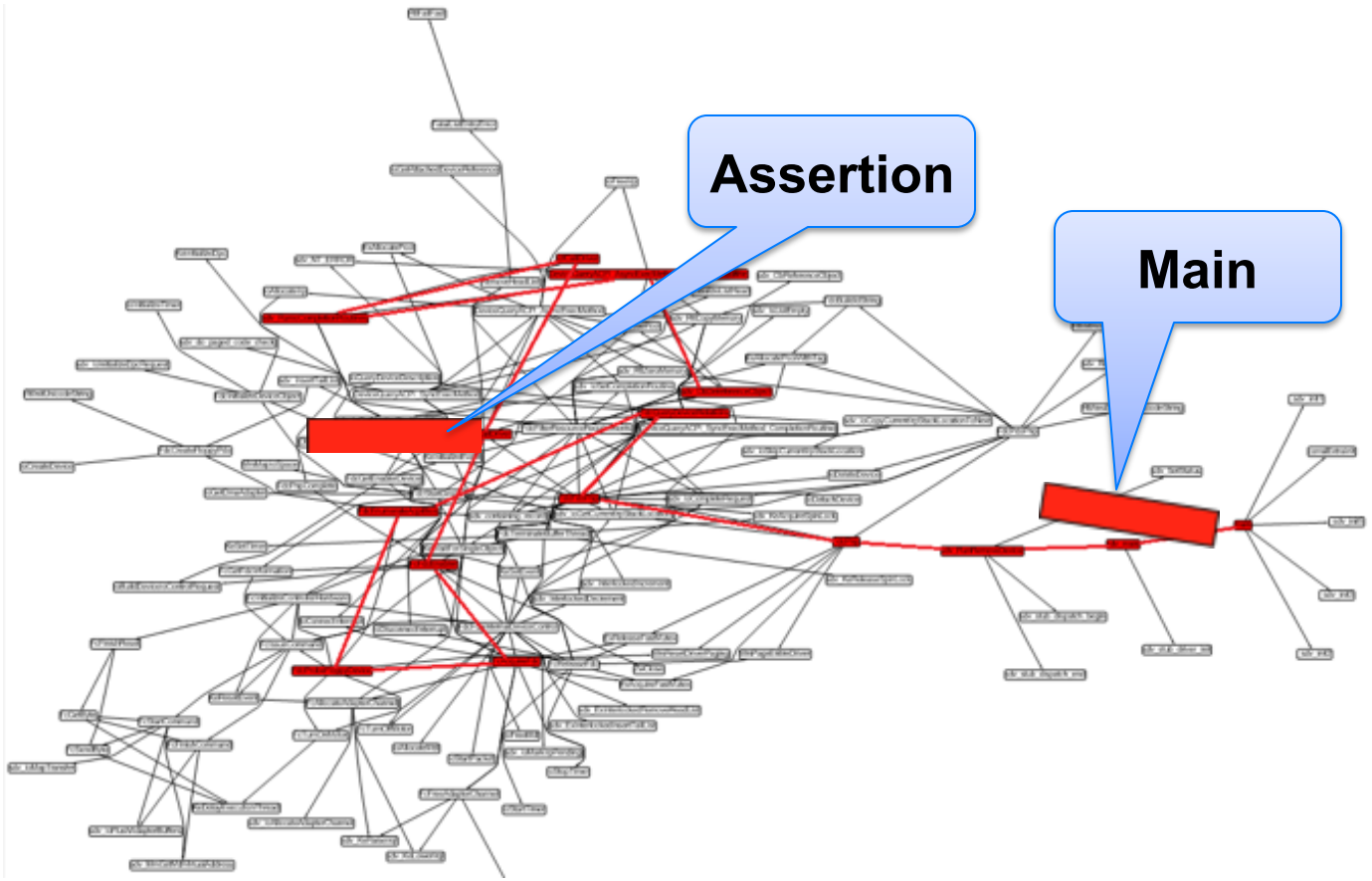
Mixed Semantics

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

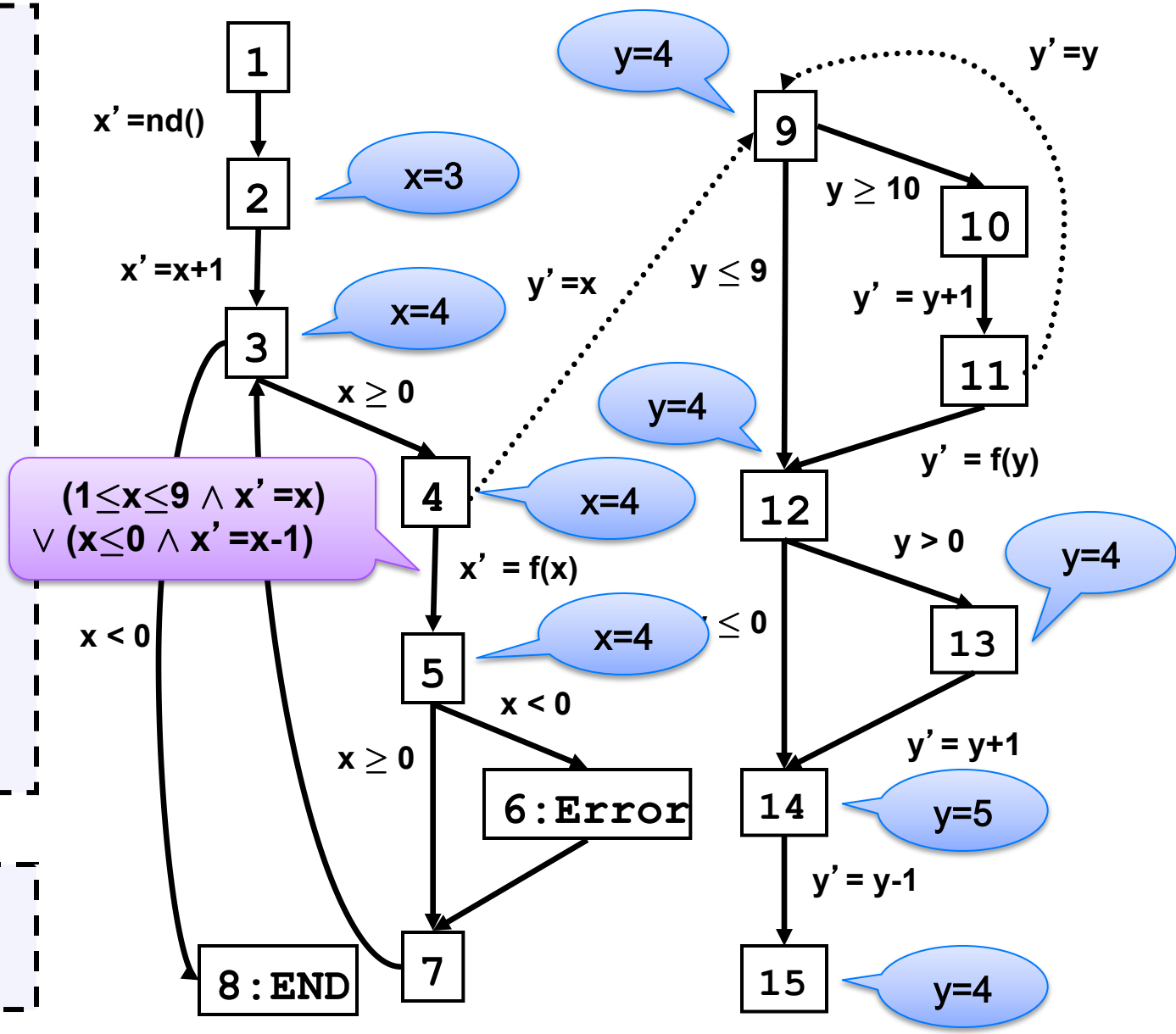
- update top activation record using function summary, or
- enter function body, forgetting history records (i.e., no return!)

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:
```

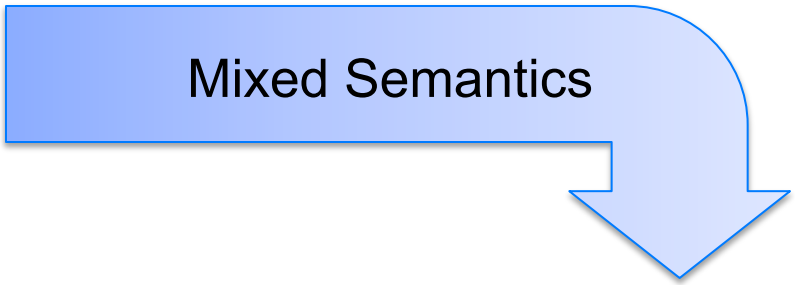


Summary of f(y)

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

Mixed Semantics as Program Transformation

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




```
main_new ()
    if (*) goto p1_entry;
    else p1_new ();
    if (*) goto p1_entry;
    else p1_new ();
    if (¬c1) goto error;
    assume (false);

p1_entry :
    if (*) goto p2_entry;
    else p2_new ();
    if (¬c2) goto error;
p2_entry :
    if (¬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: $\mathcal{F}(X) = \exists\ \boldsymbol{u}\ .\ (X \wedge \rho) \vee \text{Init}$

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

IC3/PDR Algorithm Overview

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

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

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

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

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

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

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

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

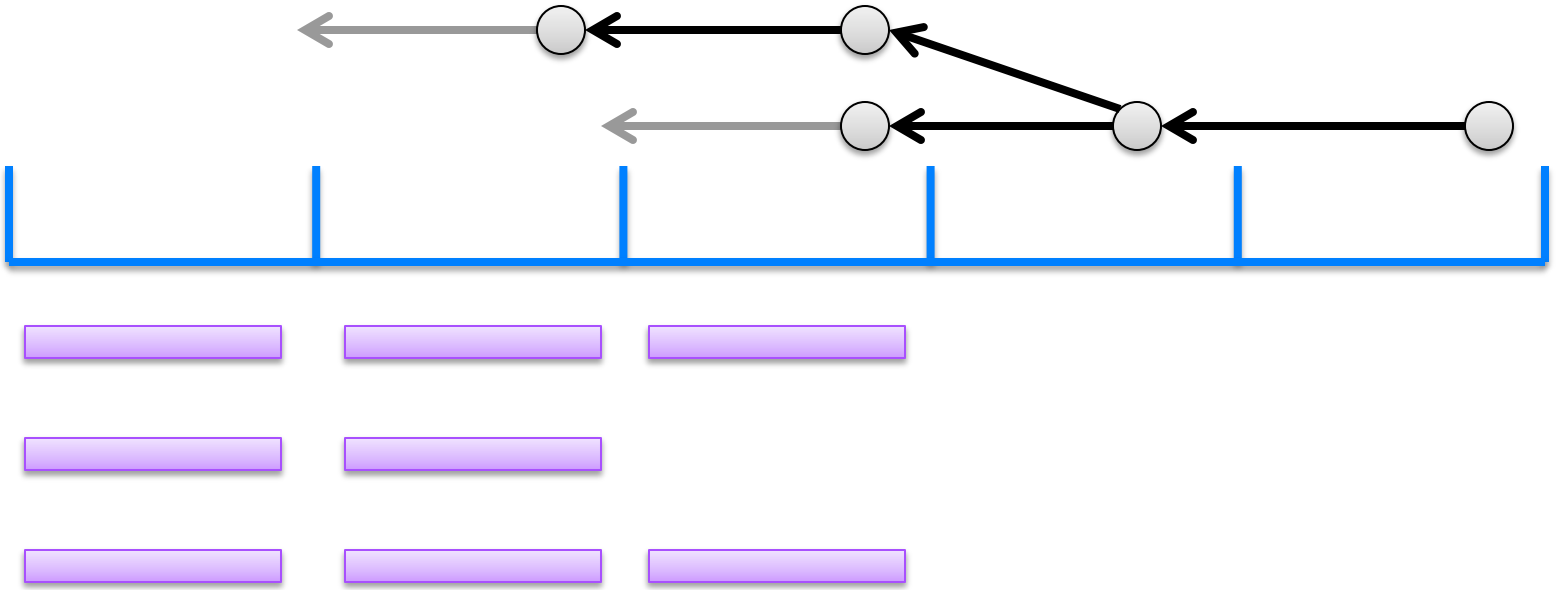
until ∞;

bounded safety

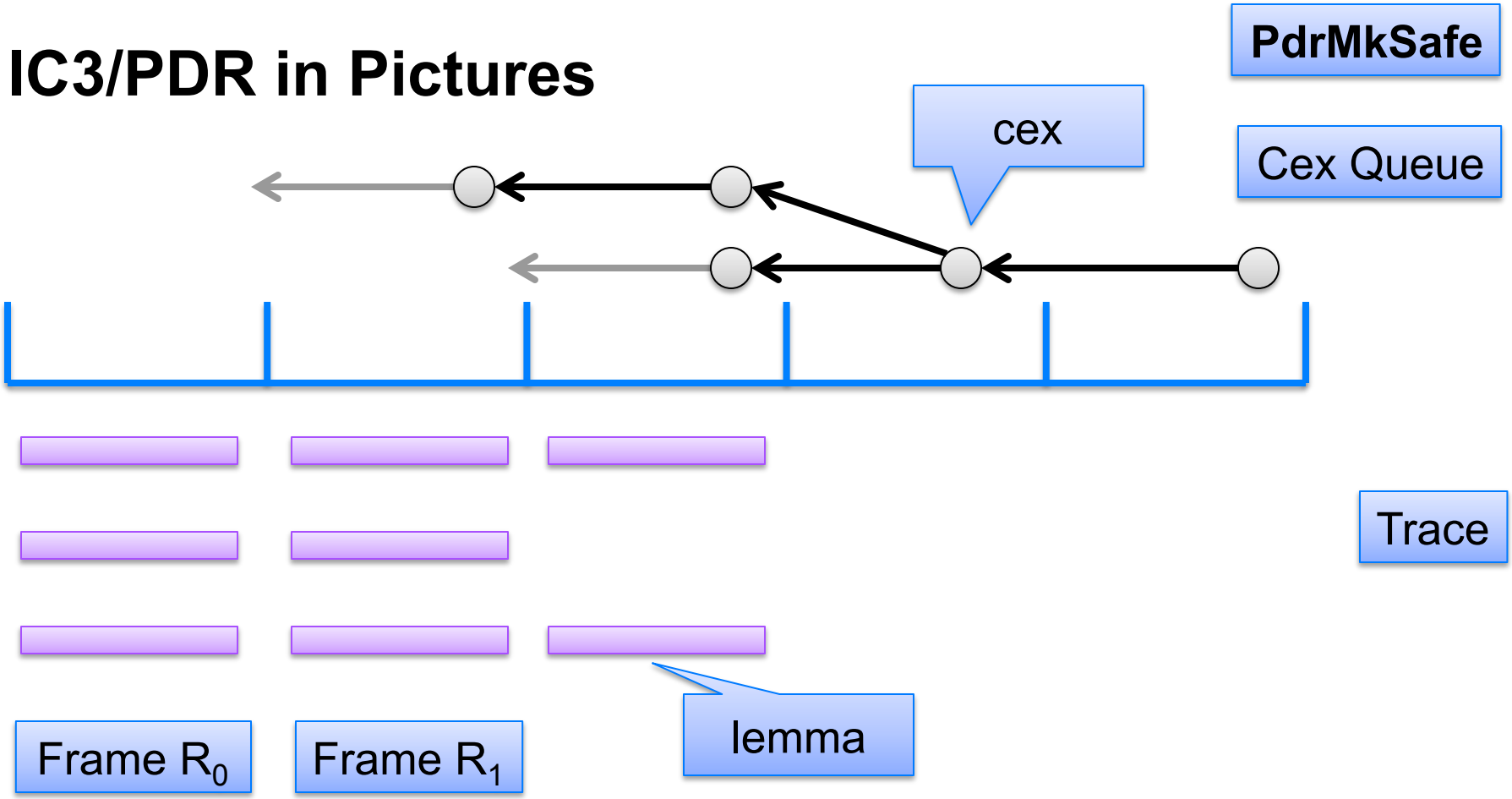
strengthen result

IC3/PDR in Pictures

PdrMkSafe

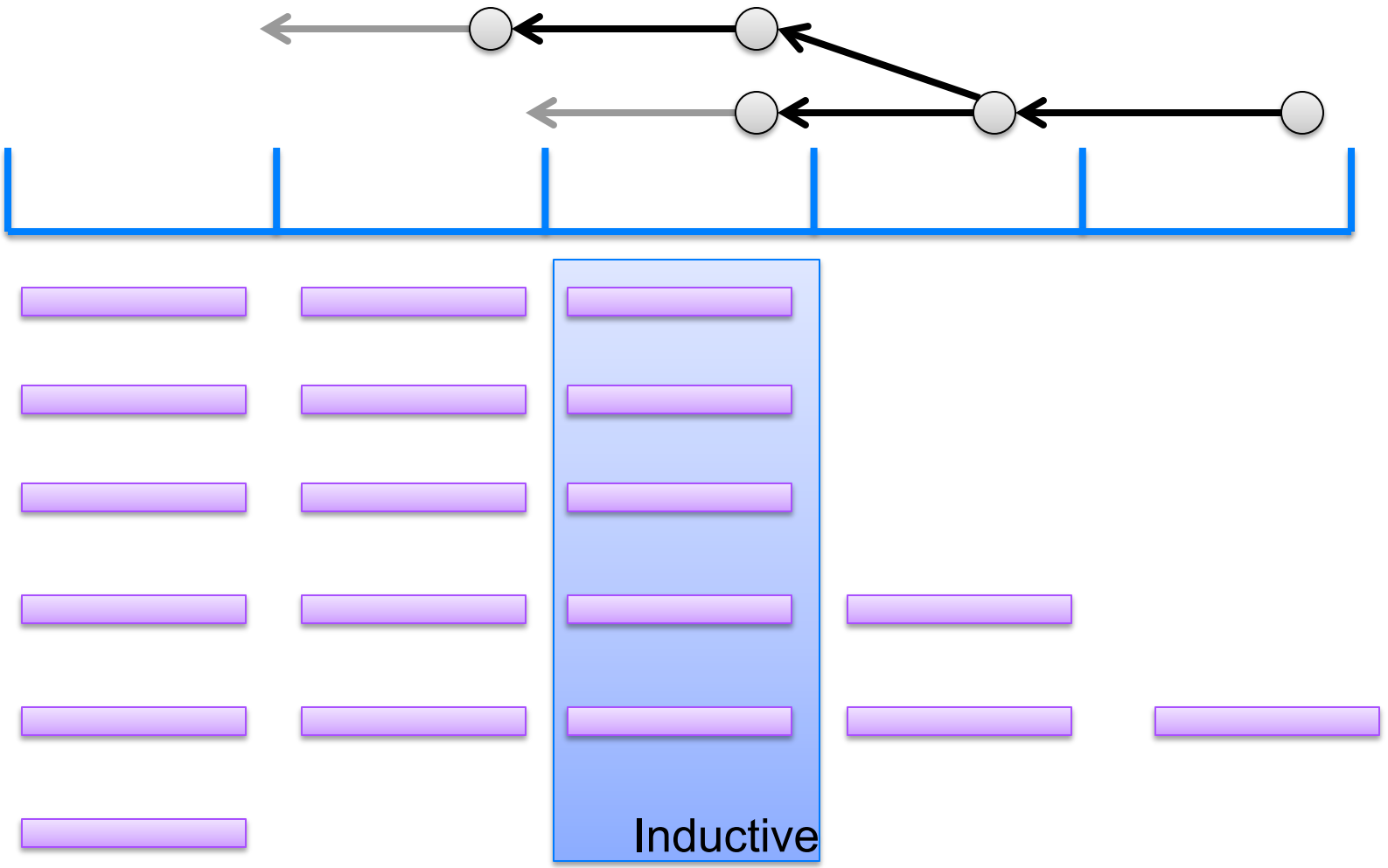


IC3/PDR in Pictures



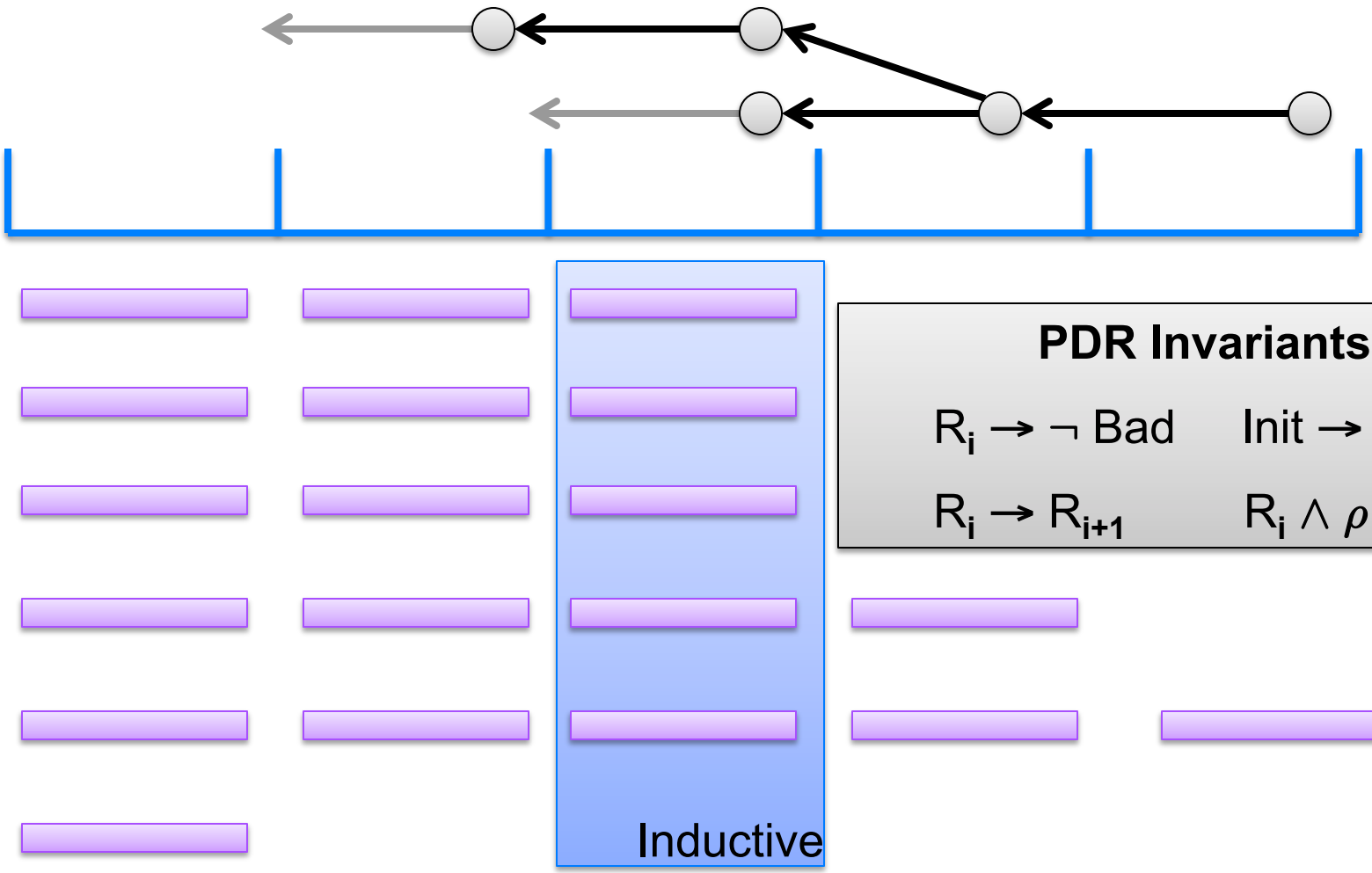
IC3/PDR in Pictures

PdrPush



IC3/PDR in Pictures

PdrPush

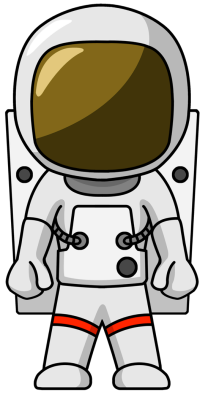


PDR Invariants

$R_i \rightarrow \neg$ Bad Init $\rightarrow R_i$

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

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

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.

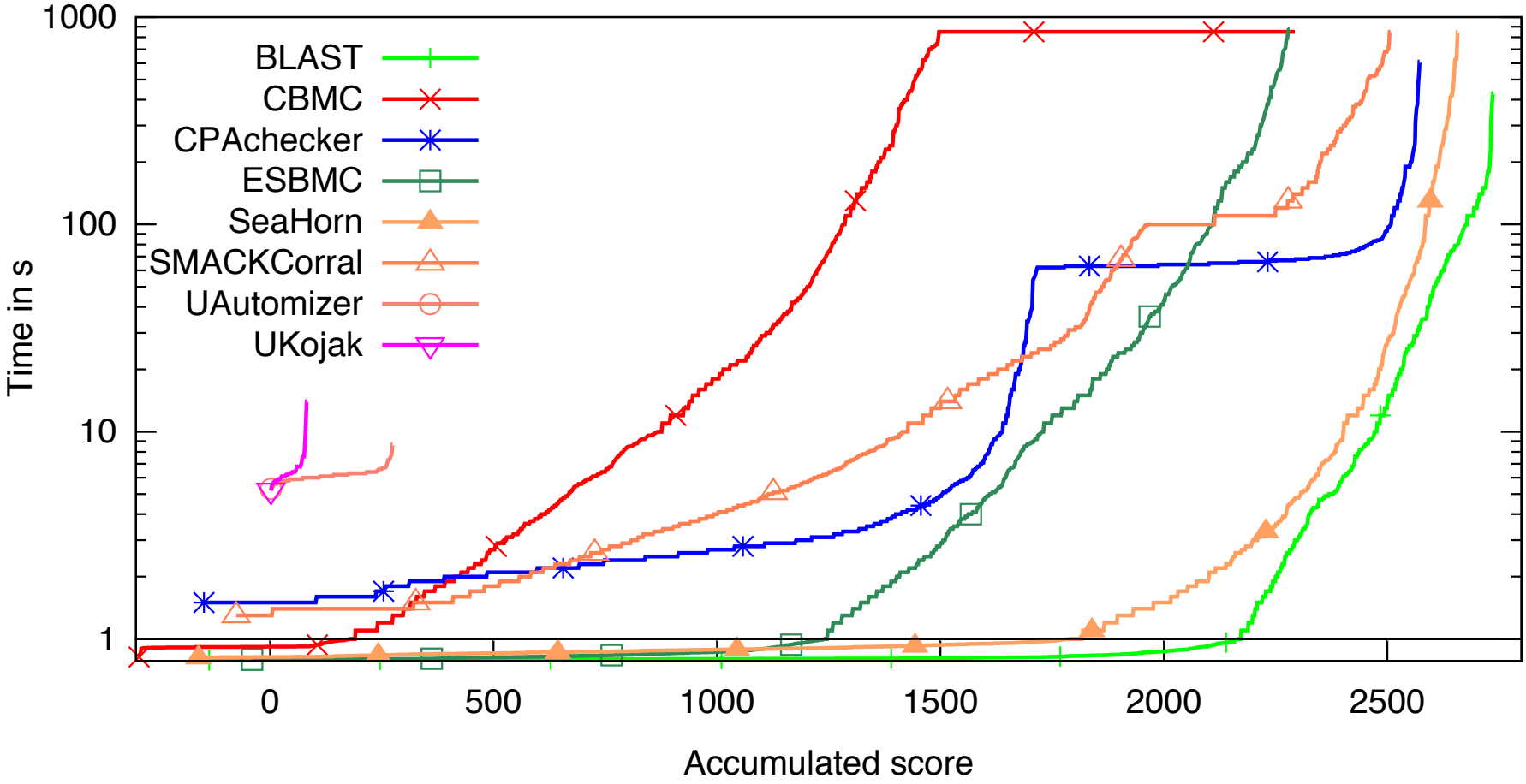
Participants:

- 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

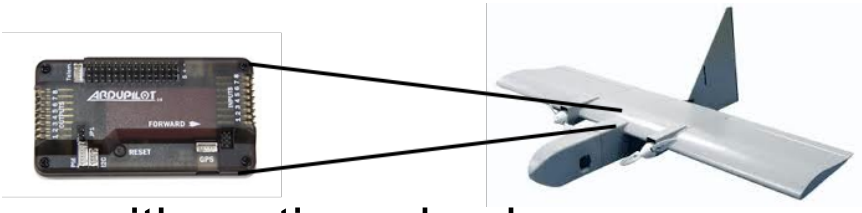
Results for DeviceDriver category



Detecting Buffer Overflow in Auto-pilot software

Show absence of Buffer Overflows in

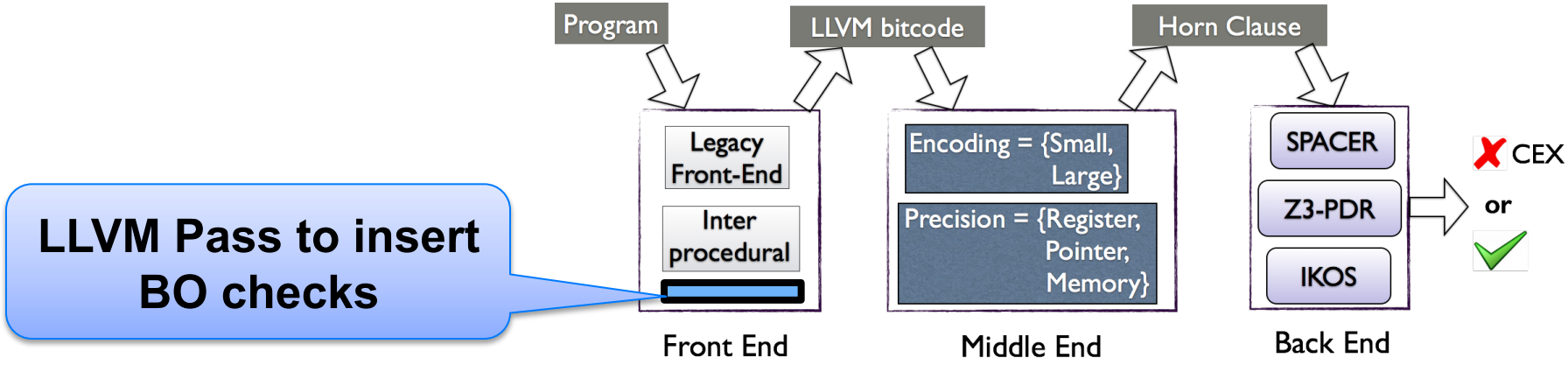
- paparazzi and mnav autopilots



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!



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

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