Building Program Verifiers from Compilers and Theorem Provers

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

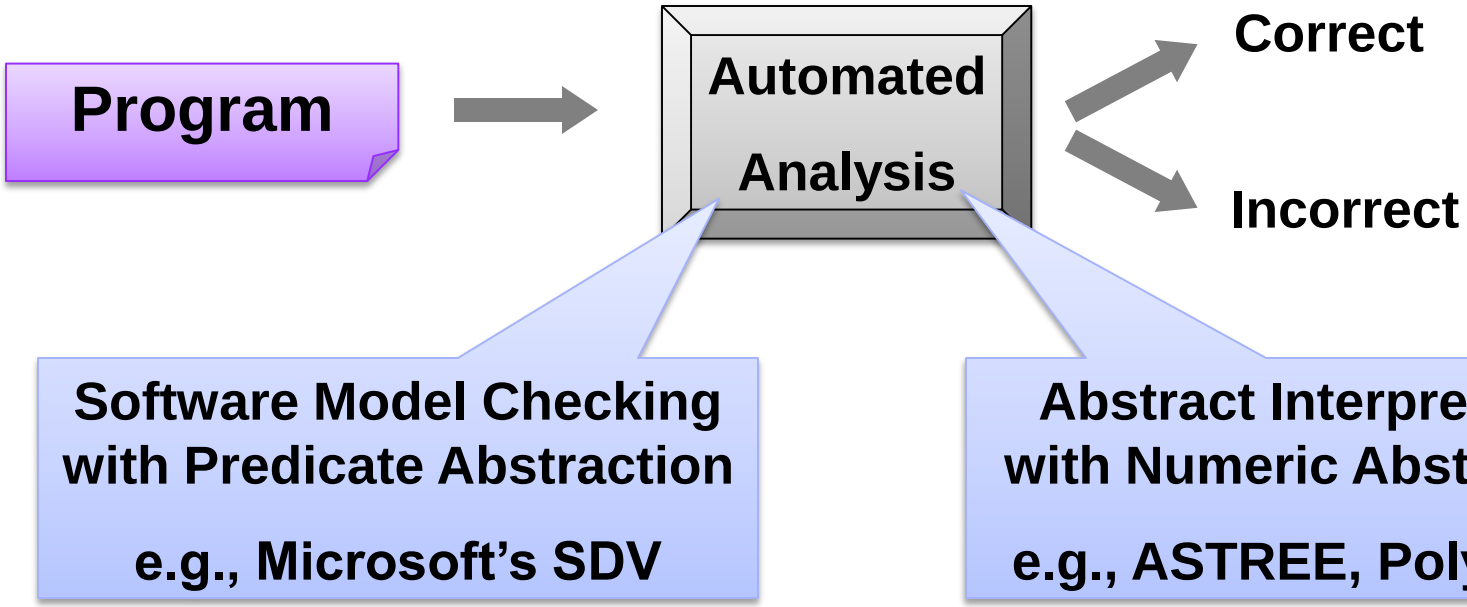
Arie Gurfinkel

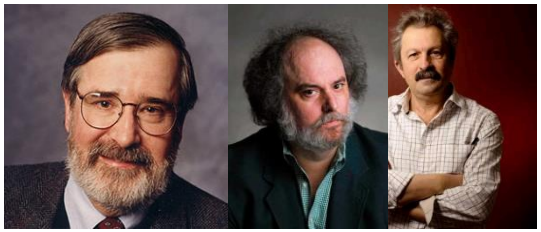
based on joint work with Teme Kahsai, Jorge A. Navas, Anvesh Komuravelli, and Nikolaj Bjorner

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





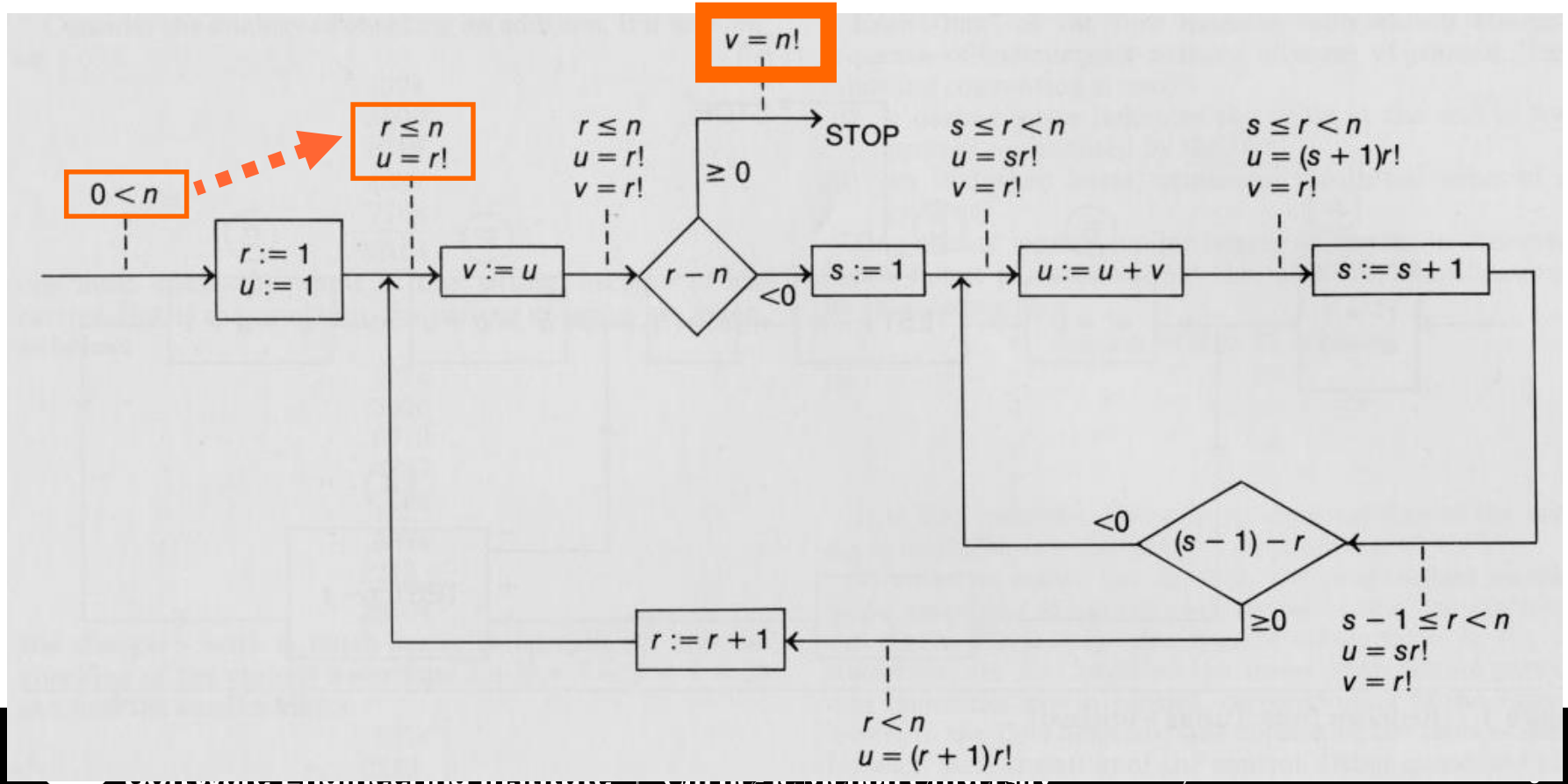
Turing, 1936: “undecidable”

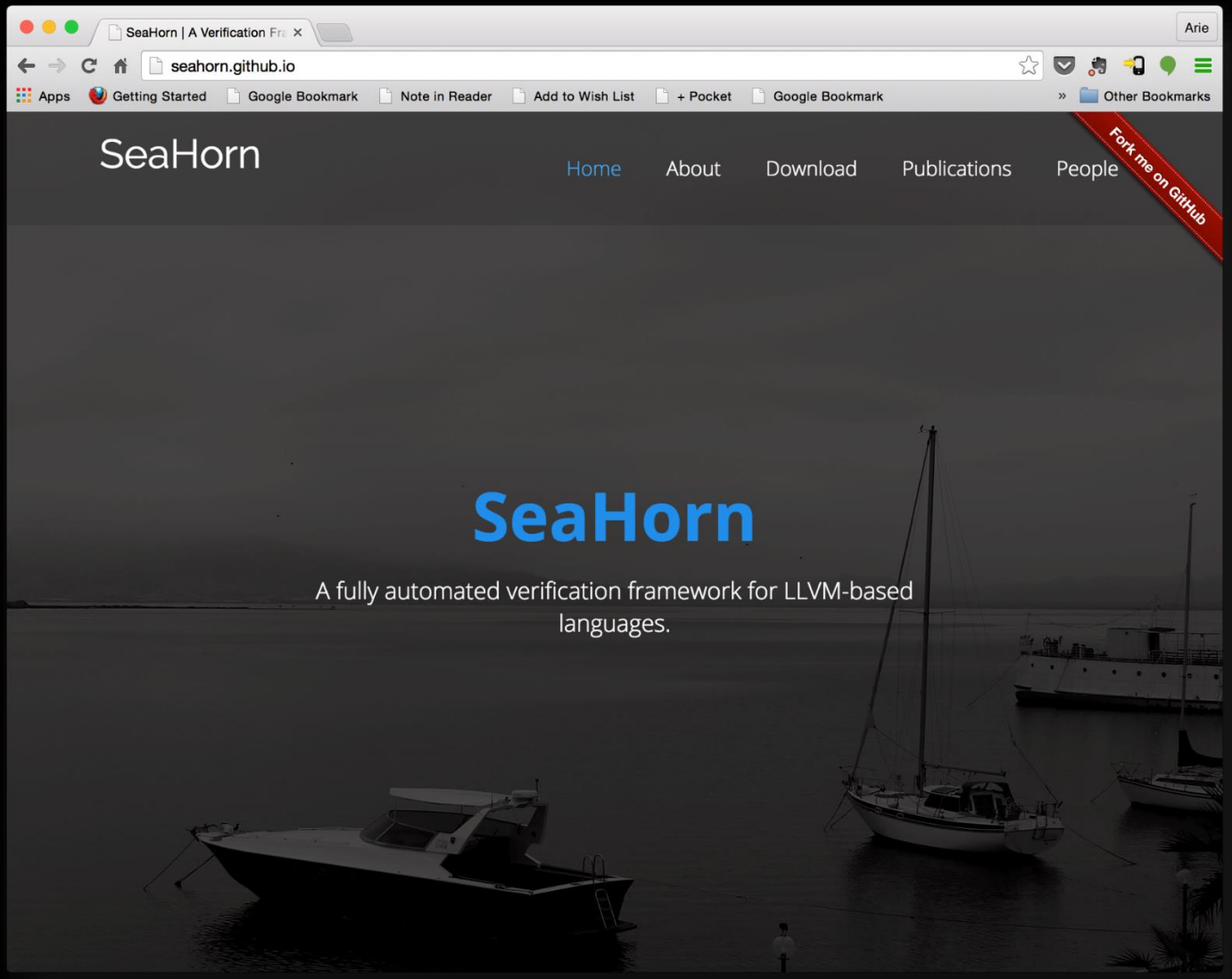
Turing, 1949

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

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

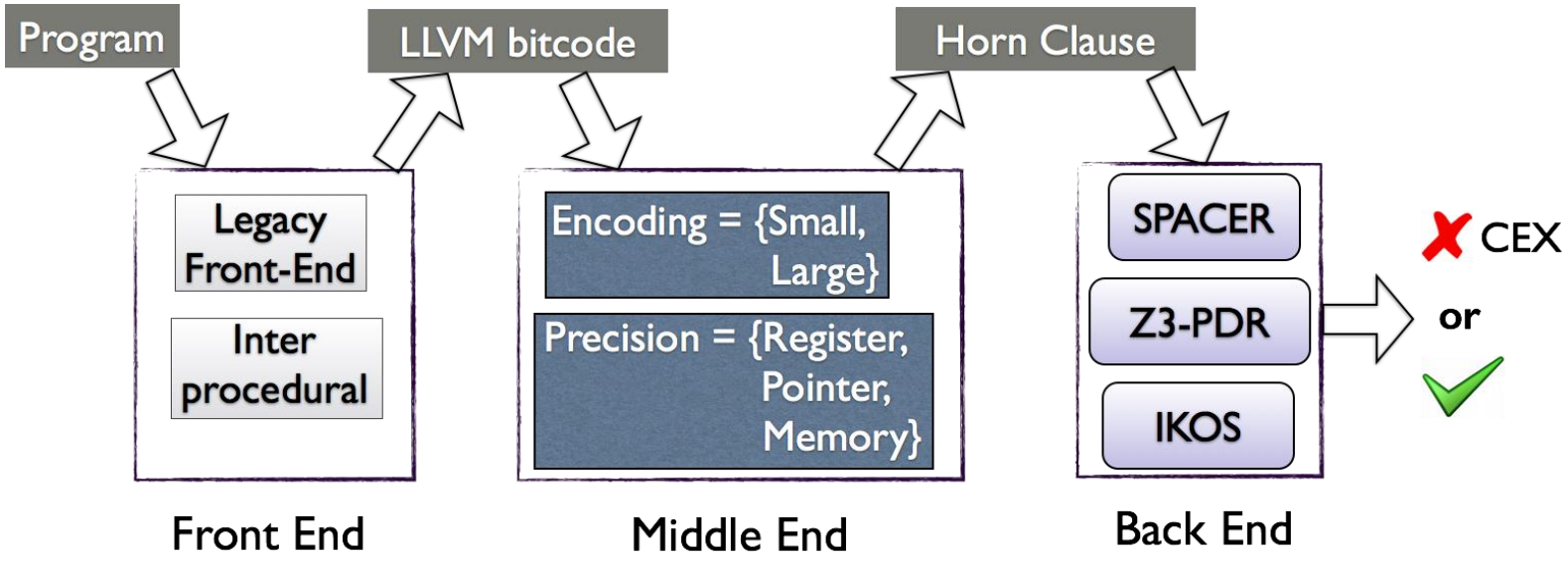
programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.





http://seahorn.github.io

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 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

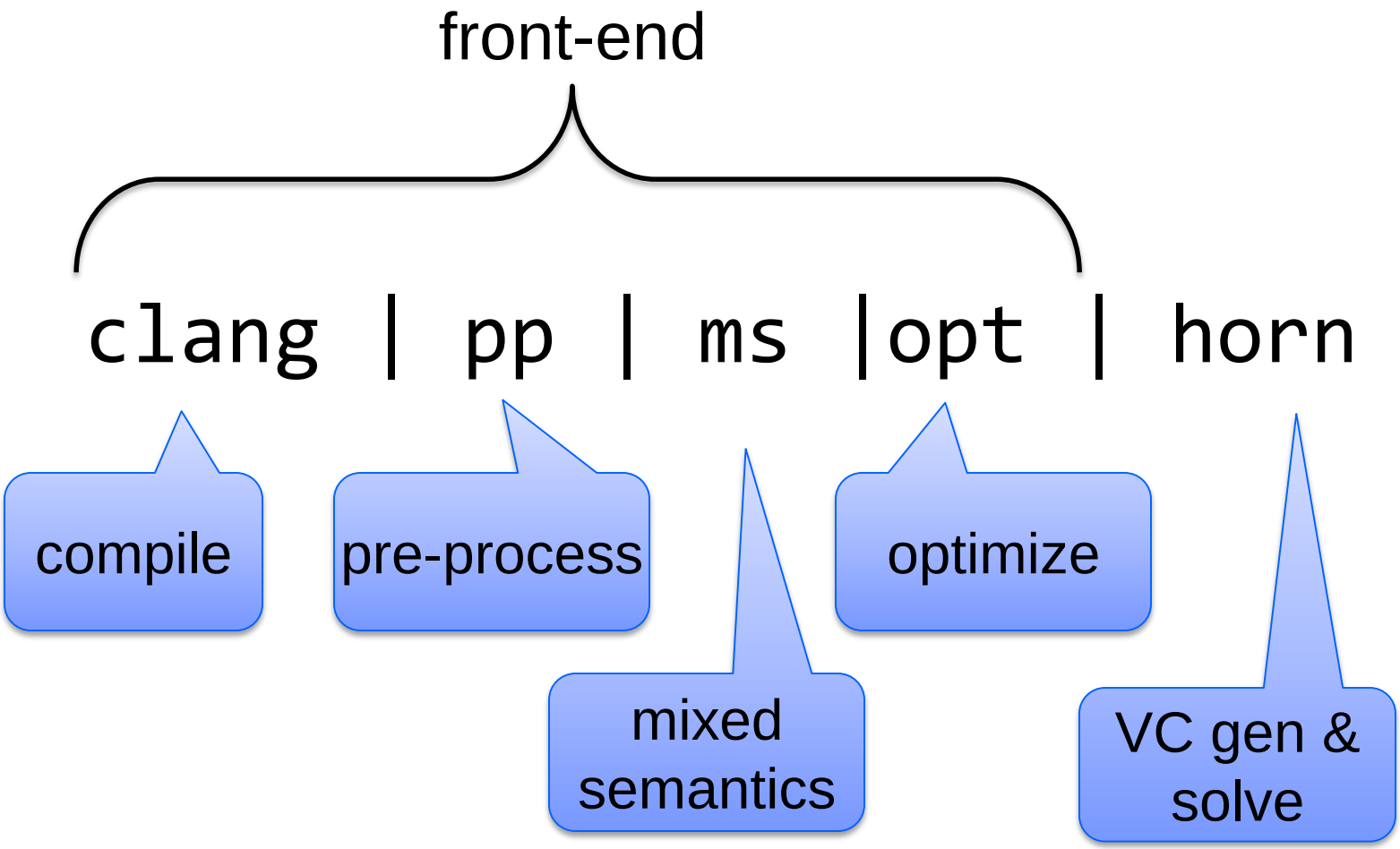
Additional options

- `--cex=trace.xml` outputs a counter-example in SV-COMP'15 format
- `--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

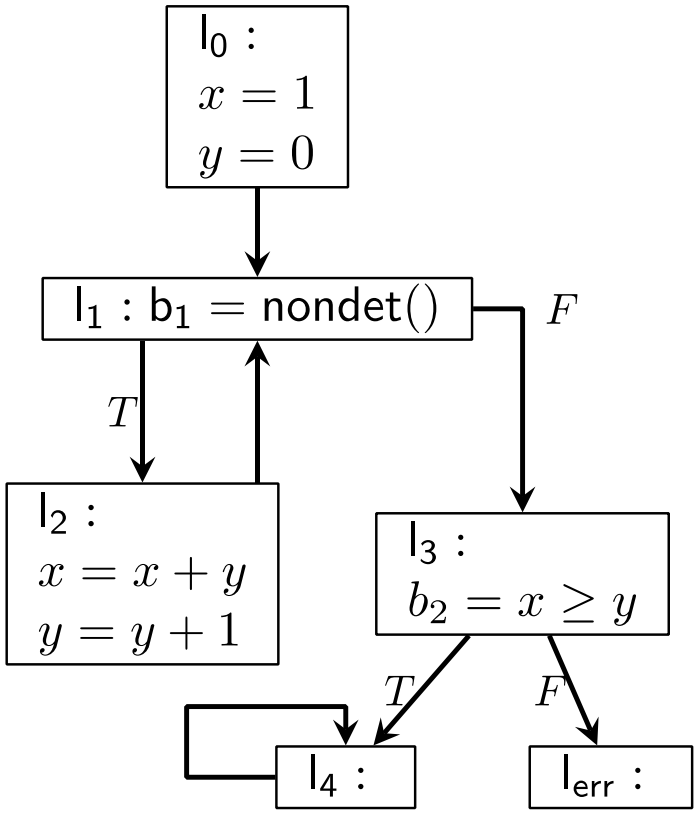
$$8\ V\ .\ (Á\ Æ\ p_1[X_1]\ Æ \ldots Æ\ p_n[X_n] \rightarrow h[X]),$$

where

- A is a background theory (e.g., Linear Arithmetic, Arrays, Bit-Vectors, or combinations of the above)
- Á 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.

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

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

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

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

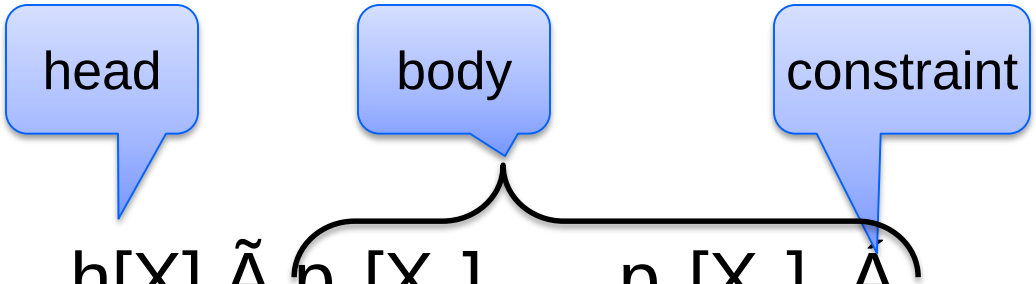
$\langle 6 \rangle$ $\mathsf{p}_4 \leftarrow (x \geq y), \mathsf{p}_3(x, y)$.

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

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

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

CHC Terminology



Rule $h[X] \tilde{A} p_1[X_1],\ldots, p_n[X_n], \acute{A}.$

Query $\text{false} \tilde{A} p_1[X_1],\ldots, p_n[X_n], \acute{A}.$

Fact $h[X] \tilde{A} \acute{A}.$

Linear CHC $h[X] \tilde{A} p[X_1], \acute{A}.$

Non-Linear CHC $h[X] \tilde{A} p_1[X_1],\ldots, p_n[X_n], \acute{A}.$ for $n > 1$

CHC Satisfiability

A **model** of a set of clauses ¦ is an interpretation of each predicate p_i that makes all clauses in ¦ 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 $\tilde{A}_i$ 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

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

Function Application

$$\frac{(\text{Pre}[p/a]) \Rightarrow P \qquad \{P\}\ \text{Body}_F\ \{Q\} \qquad (Q[p,r/a,b]) \Rightarrow \text{Post}}{\{\text{Pre}\}\ b = F(a)\ \{\text{Post}\}}$$

Recursion

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

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 , Pre) **wlp** (P, Post)

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 Æ Q

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

wlp (**while** E **do** S, Q) = I(w) Æ

8w . ((I(w) Æ E) $\rightarrow$ wlp (S, I(w))) Æ ((I(w) Æ :E) $\rightarrow$ Q))

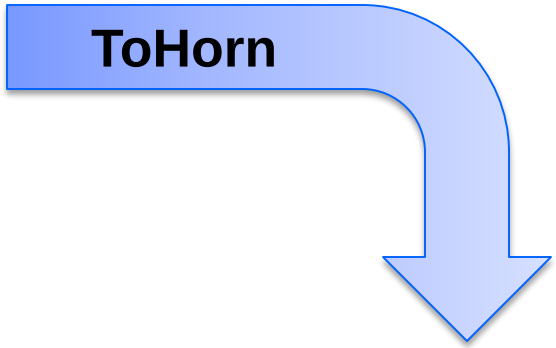
wlp (y = P(E), Q) = p_{pre}(E) Æ (8 r. p(E, r) $\rightarrow$ Q[r/y])

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

ToHorn (Prog) = wlp (Main(), true) Æ 8{P 2 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) Ã y>=0.
C2: I(x+1,y-1,x_o,y_o) Ã I(x,y,x_o,y_o), y>0.
C3: false Ã I(x,y,x_o,y_o), y·0, x≠x_o+y_o
```

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

Dual WLP

Dual weakest liberal pre-condition

dual-wlp (P, Post) = :**wlp** (P, :Post)

s ² **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

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:

$p_{en}(x_0,x)$ Ã $x_0=x$

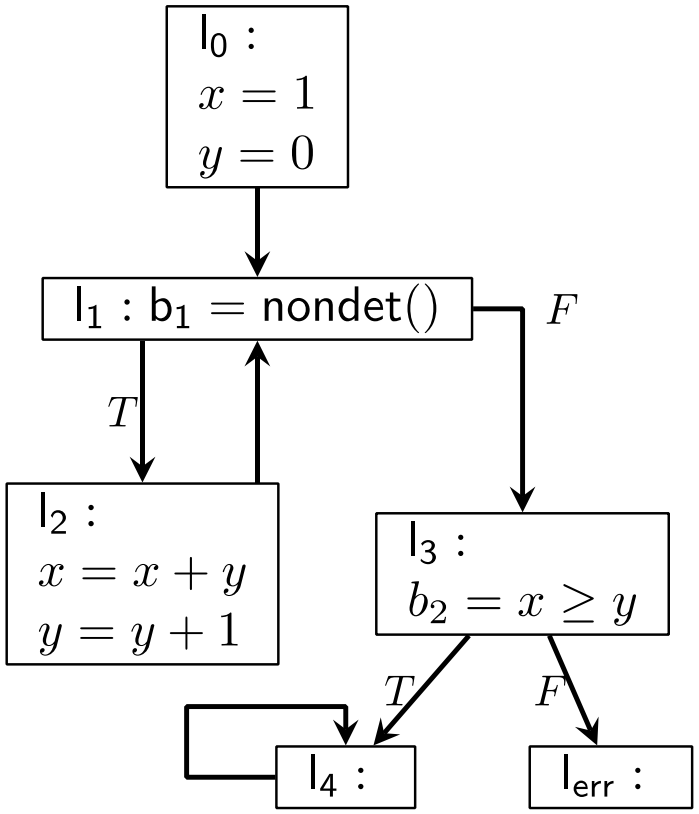
$l_i(x_0,w')$ Ã $l_j(x_0,w)$ Æ :wlp (S, :(w=w')), for each statement l_i: S; goto l_j

$p(x_0,r)$ Ã $p_{ex}(x_0,r)$

false Ã $Main_{ex}(x, ret)$

Example Horn Encoding

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



$\langle 1 \rangle$ p_0.

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

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

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

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

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

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

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

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

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"

Large Step Encoding: Single Static Assignment

```
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:
```

Example: Single Static Assignment

```
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:
```

Example: Large Step Encoding

```
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:
```

Example: Large Step Encoding

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

```
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: Large Step Encoding

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

$B_2 \rightarrow x_0 < N$

$B_3 \rightarrow B_2 \wedge y_0 > 0$

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

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

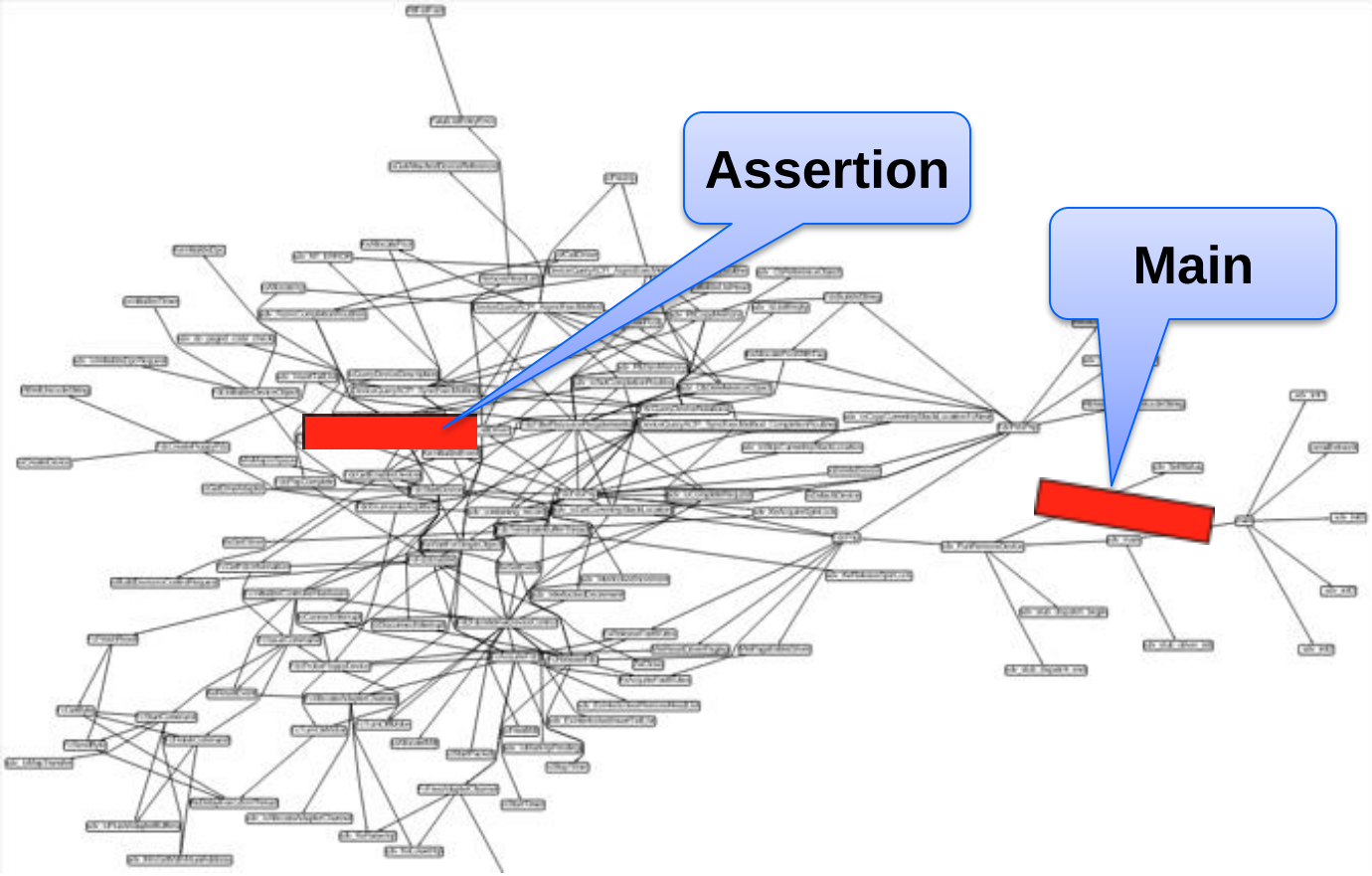
$p_1(x'_0, y'_0)$ Ã $p_1(x_0, y_0)$, Á.

```
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
```

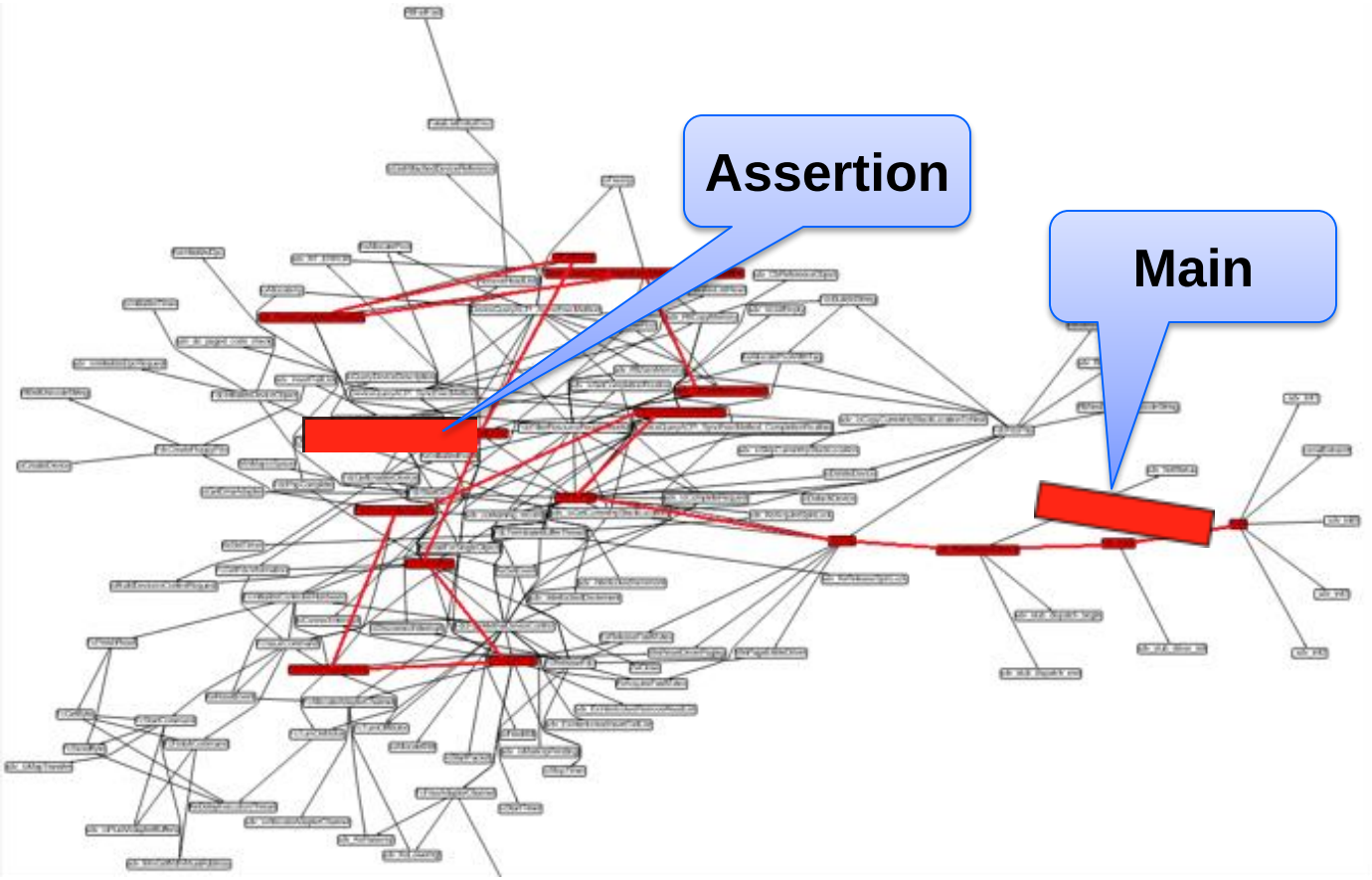
Mixed Semantics

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]
  - (¾, ¾`) 2 ||f|| iff the execution of f on input state ¾ terminates and results in state ¾'
- some execution steps are big, some are small

Non-deterministic executions of function calls

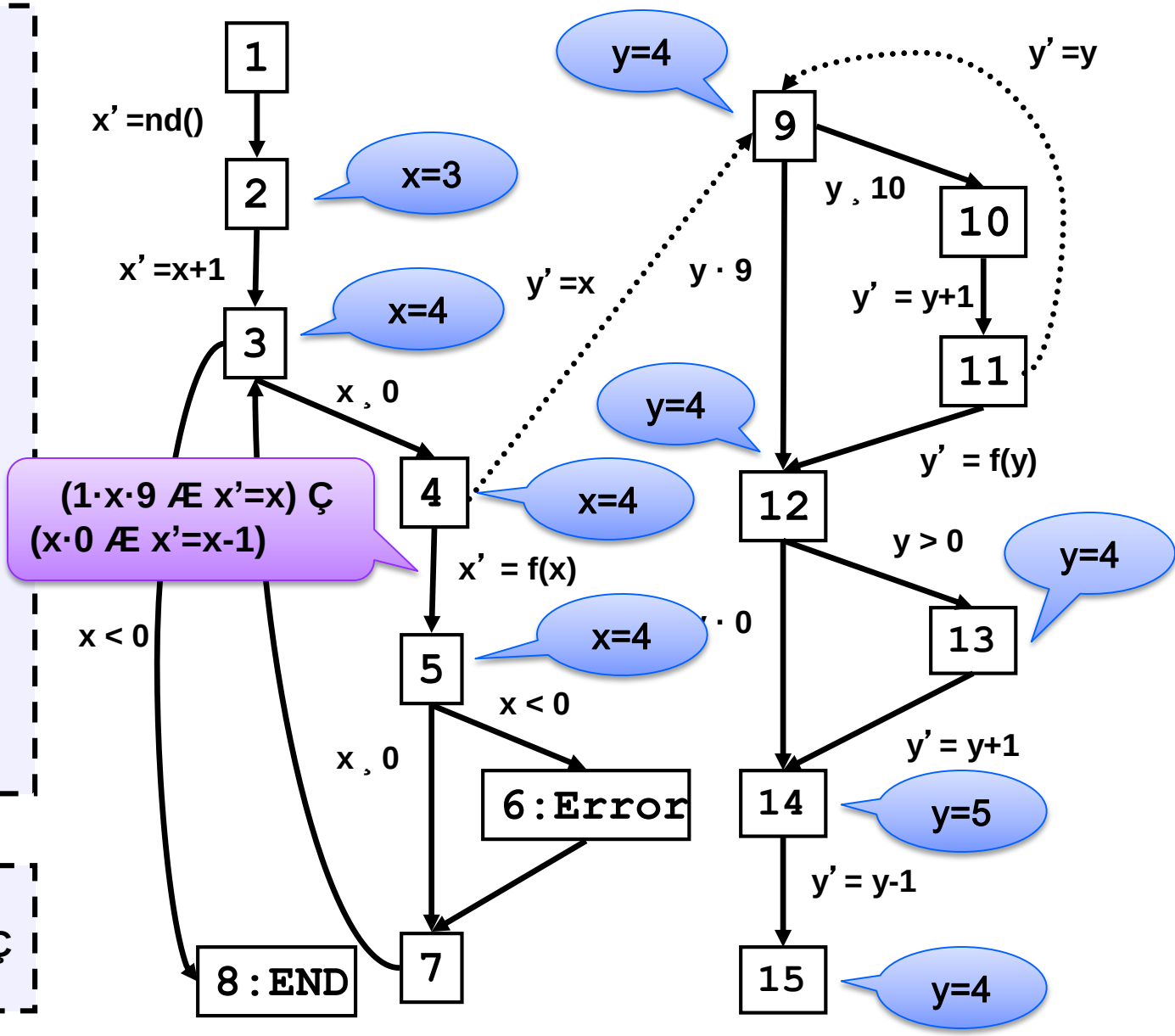
- update top activation record using function summary, or
- enter function body, forgetting history records (i.e., no return!)

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 ² EF (pc=L) , K^m ² EF (pc=L) and K ² EG (pc≠L) , K^m ² EG (pc≠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:
```

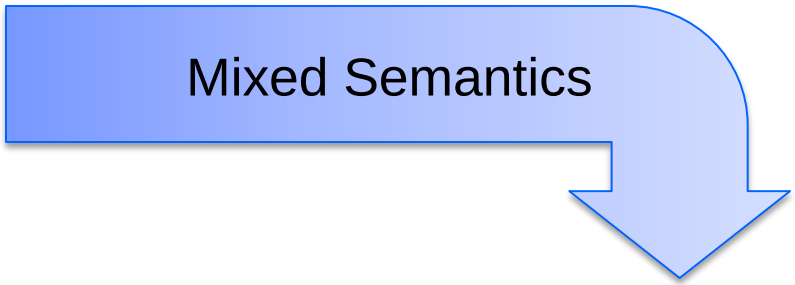


Summary of f(y)

(1·y·9 Æ y' =y) Ç
(y·0 Æ 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);
```

Implementing Mixed Semantics in LLVM

Something about how this can be implemented as a simple transformation in LLVM

in the Lab, show how to do this transformation by hand by modifying the bitcode and using opt to execute the optimization

SOLVING CHC WITH SMT

Programs, Cexs, Invariants

A program *P = (V, Init, ½, Bad)*

- Notation: F(X) = 9 ***u*** . (X Æ ½) Ç 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

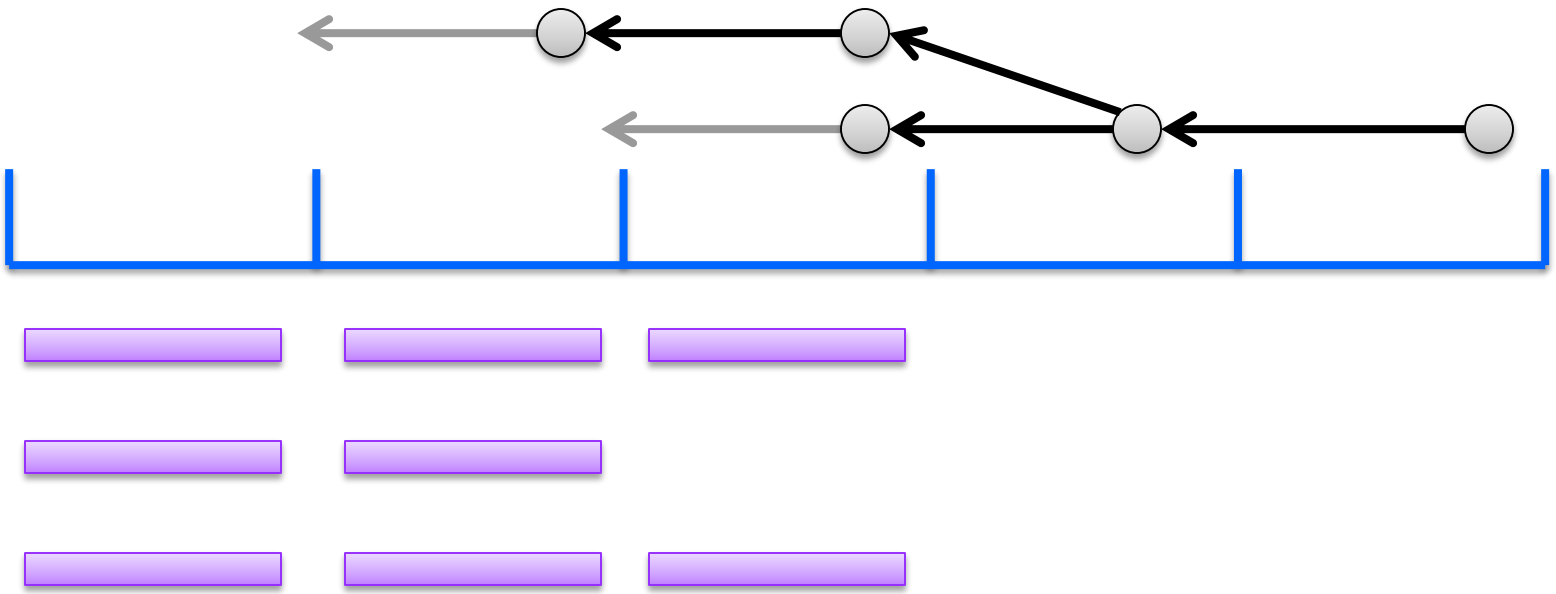
Input: Transition system $T = (Init, Tr, Bad)$

1 $F_0 \leftarrow Init$; $N \leftarrow 0$
2 **repeat**
3 $\quad \boldsymbol{G} \leftarrow \textsc{PdrMkSafe}([F_0, \ldots, F_N], Bad)$
4 $\quad$ **if** $\boldsymbol{G} = [\,]$ **then return** UNSAFE;
5 $\quad \forall 0 \leq i \leq N \cdot F_i \leftarrow \boldsymbol{G}[i]$
6 $\quad F_0, \ldots, F_N \leftarrow \textsc{PdrPush}([F_0, \ldots, F_N])$
$\quad$ // $F_0, \ldots, F_N$ is a safe δ-trace
7 $\quad$ **if** $\exists 0 \leq i \leq N \cdot F_i = \emptyset$ **then return** SAFE;
8 $\quad N \leftarrow N + 1$; $F_N \leftarrow \emptyset$
9 **until** ∞;

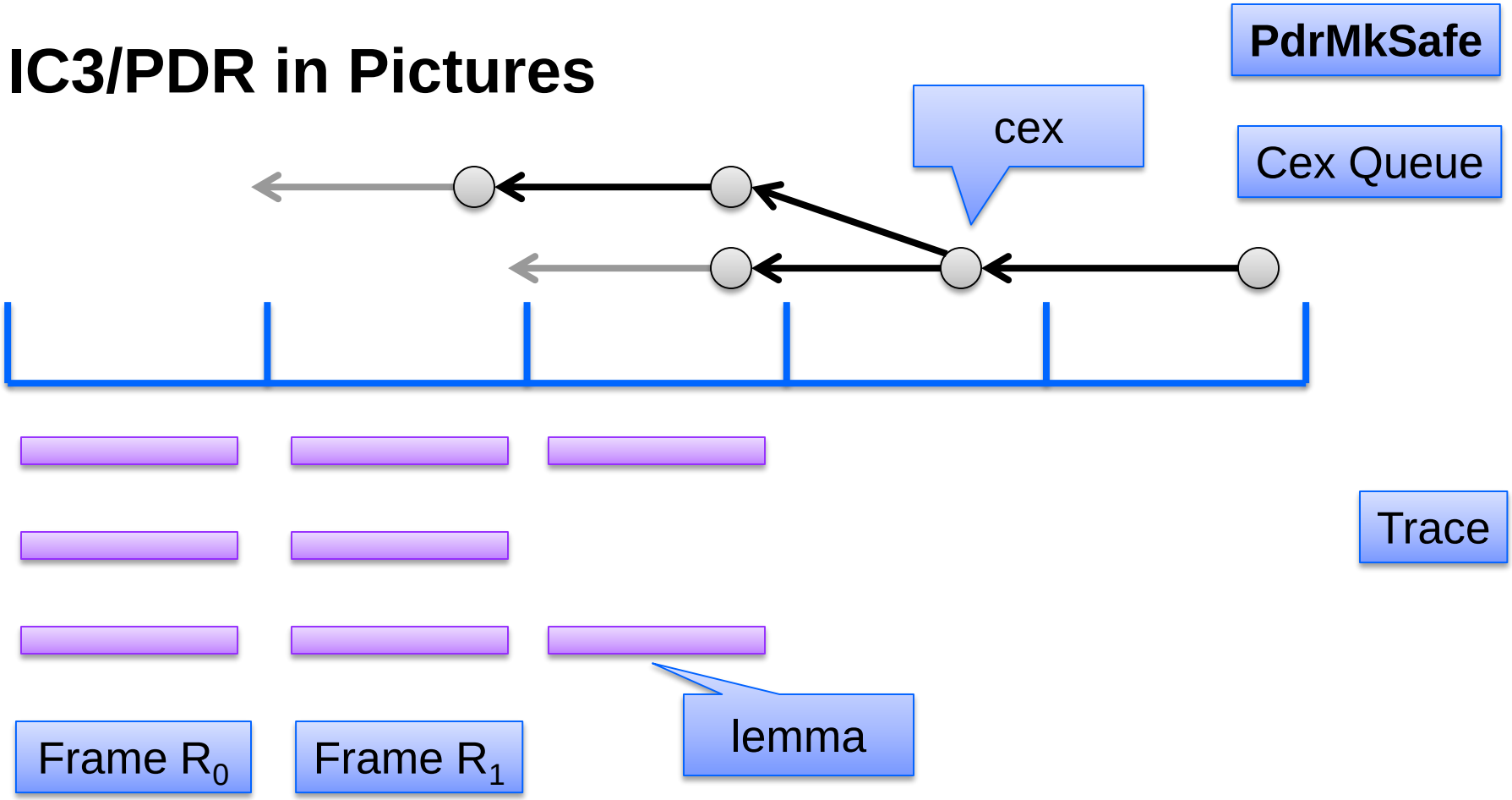
Aaron R. Bradley:SAT-Based Model Checking without Unrolling. VMCAI 2011: 70-87

IC3/PDR in Pictures

PdrMkSafe

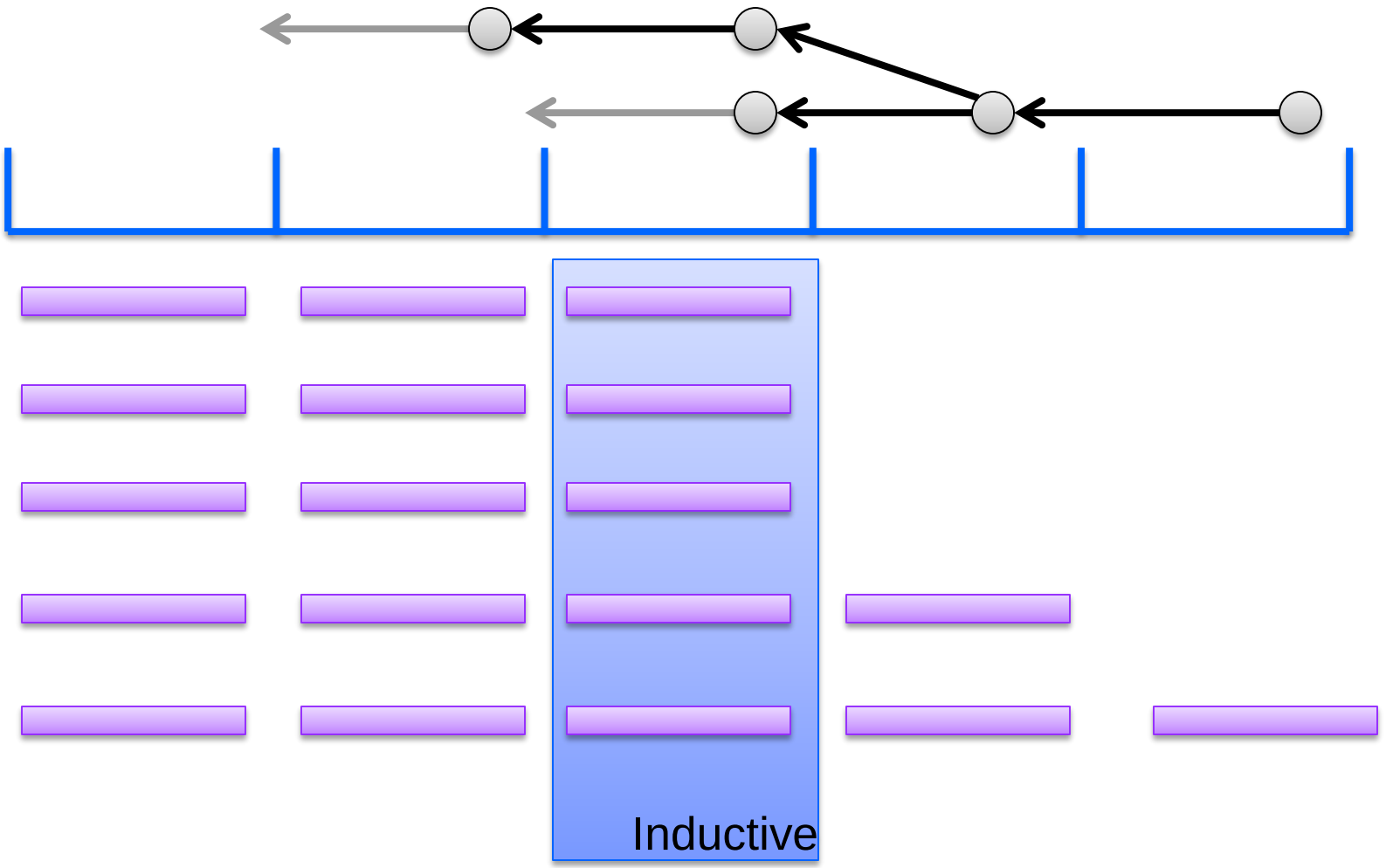


IC3/PDR in Pictures



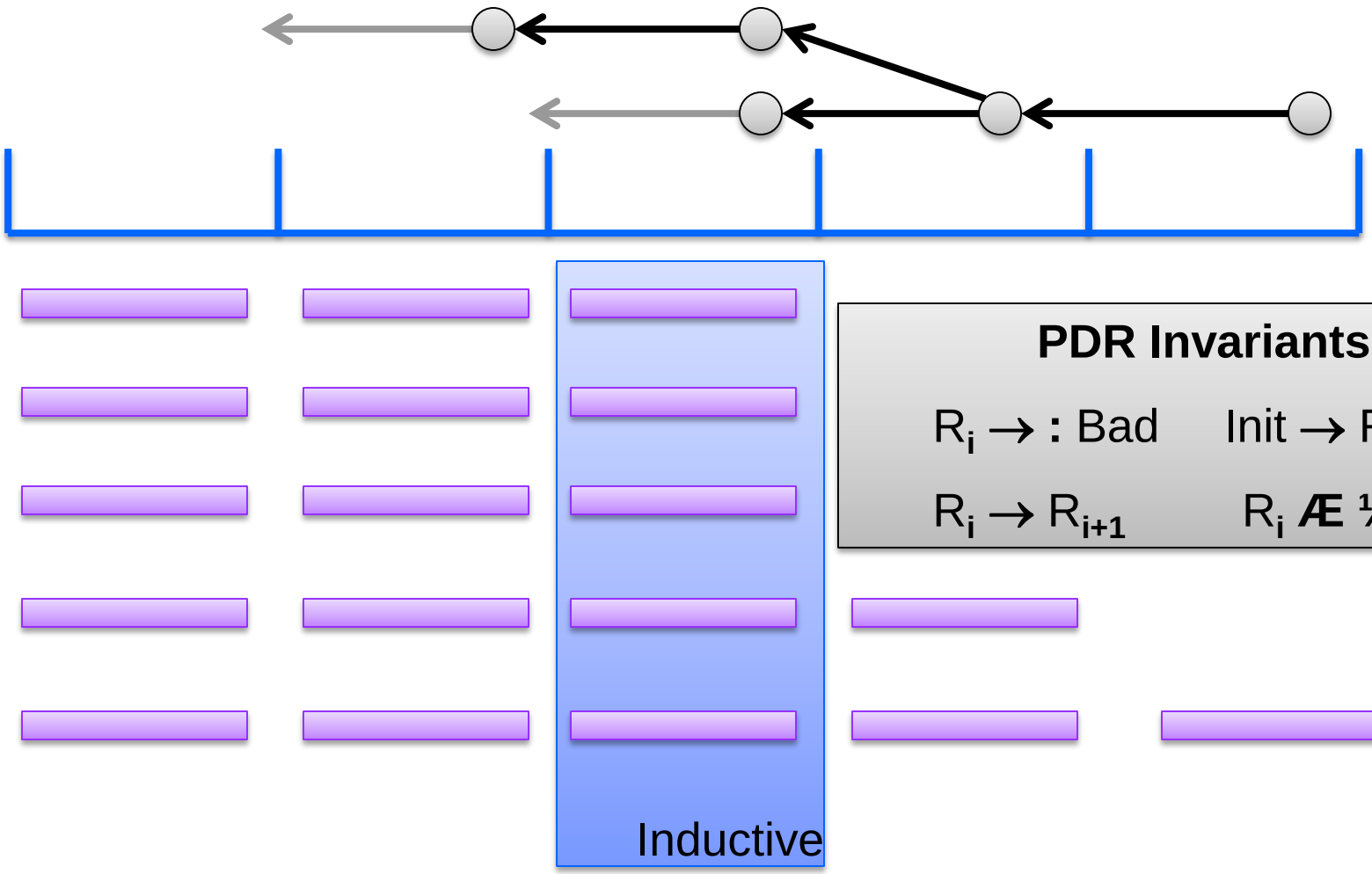
IC3/PDR in Pictures

PdrPush



IC3/PDR in Pictures

PdrPush

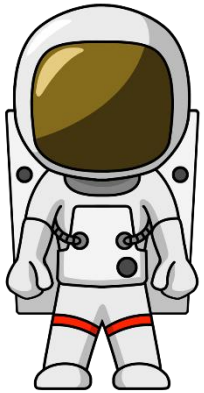


PDR Invariants

$R_i \rightarrow$: Bad Init $\rightarrow R_i$

$R_i \rightarrow R_{i+1}$ R_i Æ ½ $\rightarrow R_{i+1}$

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.

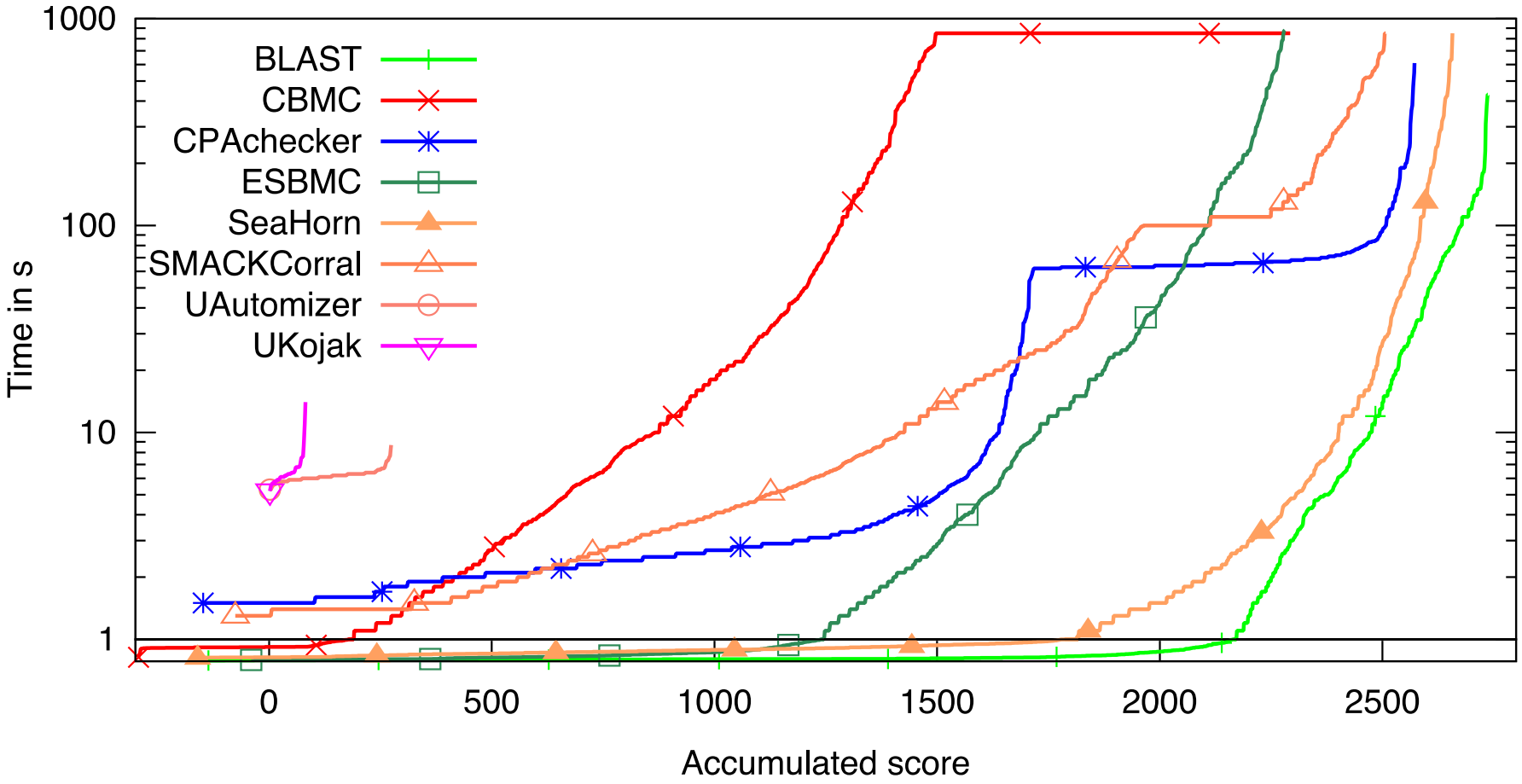
Participants:

- 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



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