SeaHorn: Software Model Checking with SMT and AI

Arie Gurfinkel

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

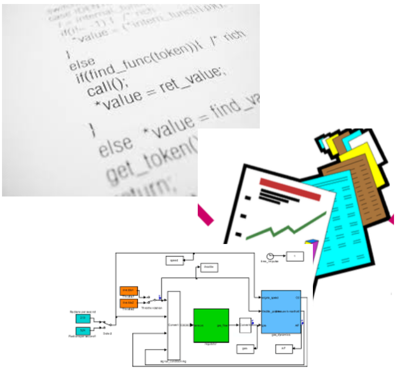
http://ece.uwaterloo.ca/~agurfink

based on work with Teme Kahsai, Jorge Navas, Anvesh Komuravelli, Jeffrey Gennari, Ed Schwartz, and many others …

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 Software Analysis

Model Checking


[Clarke and Emerson, 1981]

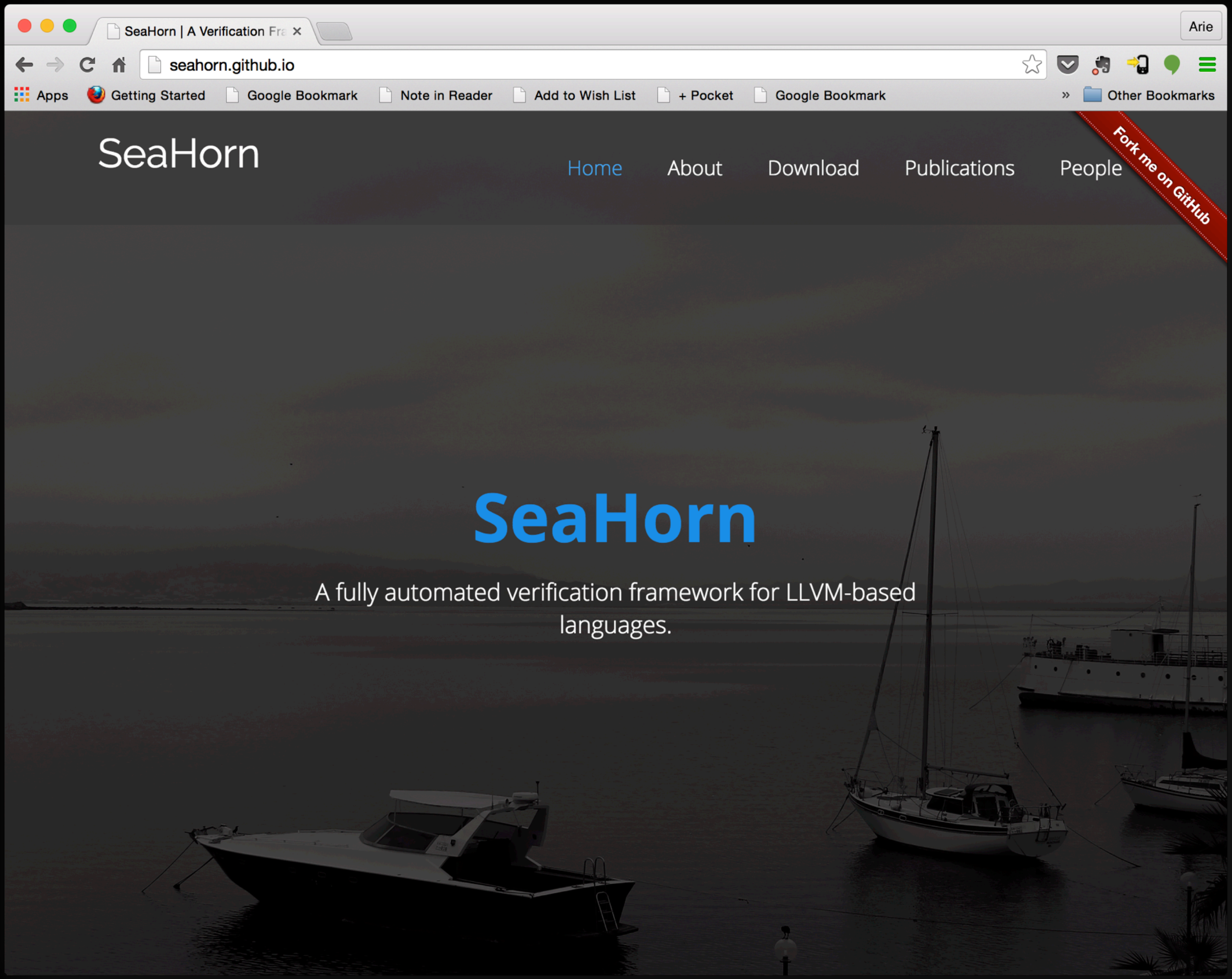
[Queille and Sifakis, 1982]

Abstract Interpretation


[Cousot and Cousot, 1977]

Symbolic Execution

[King, 1976]



http://seahorn.github.io

http://seahorn.github.io

UNIVERSITY OF WATERLOO

Automated Verification

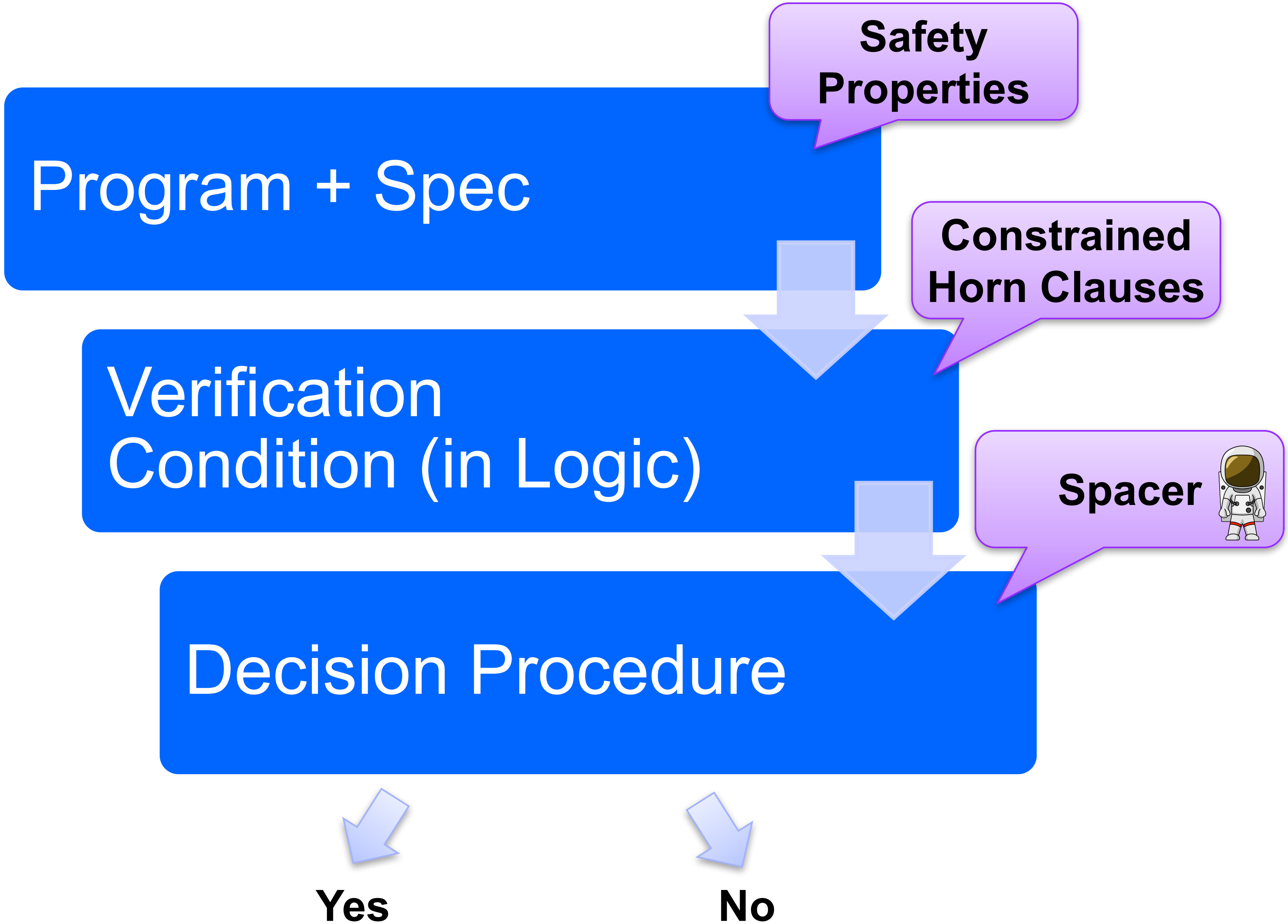
Deductive Verification

- 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



SeaHorn Usage

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

test.c

```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;
}
```

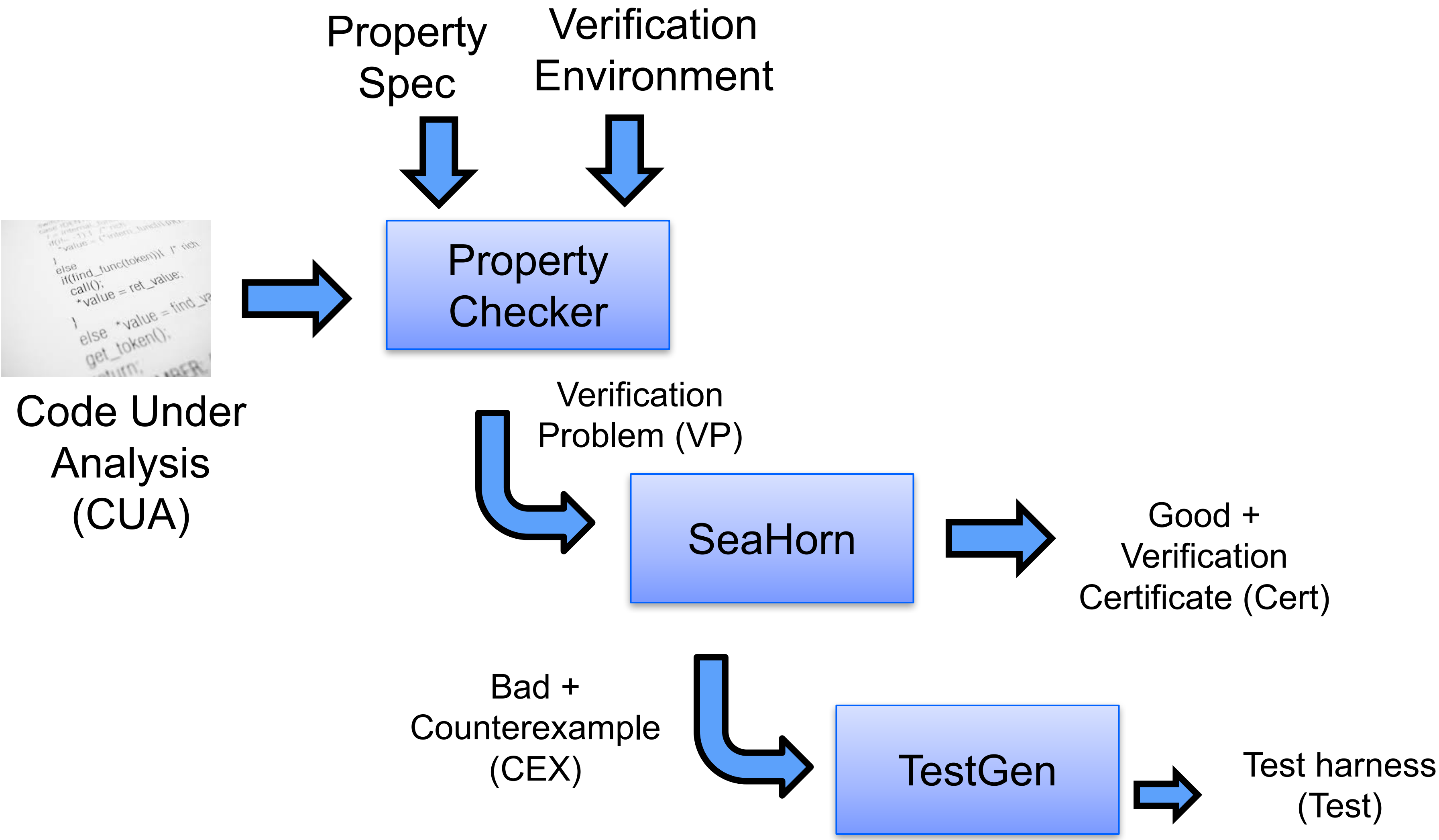
SeaHorn command:

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

SeaHorn result:

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

SeaHorn 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

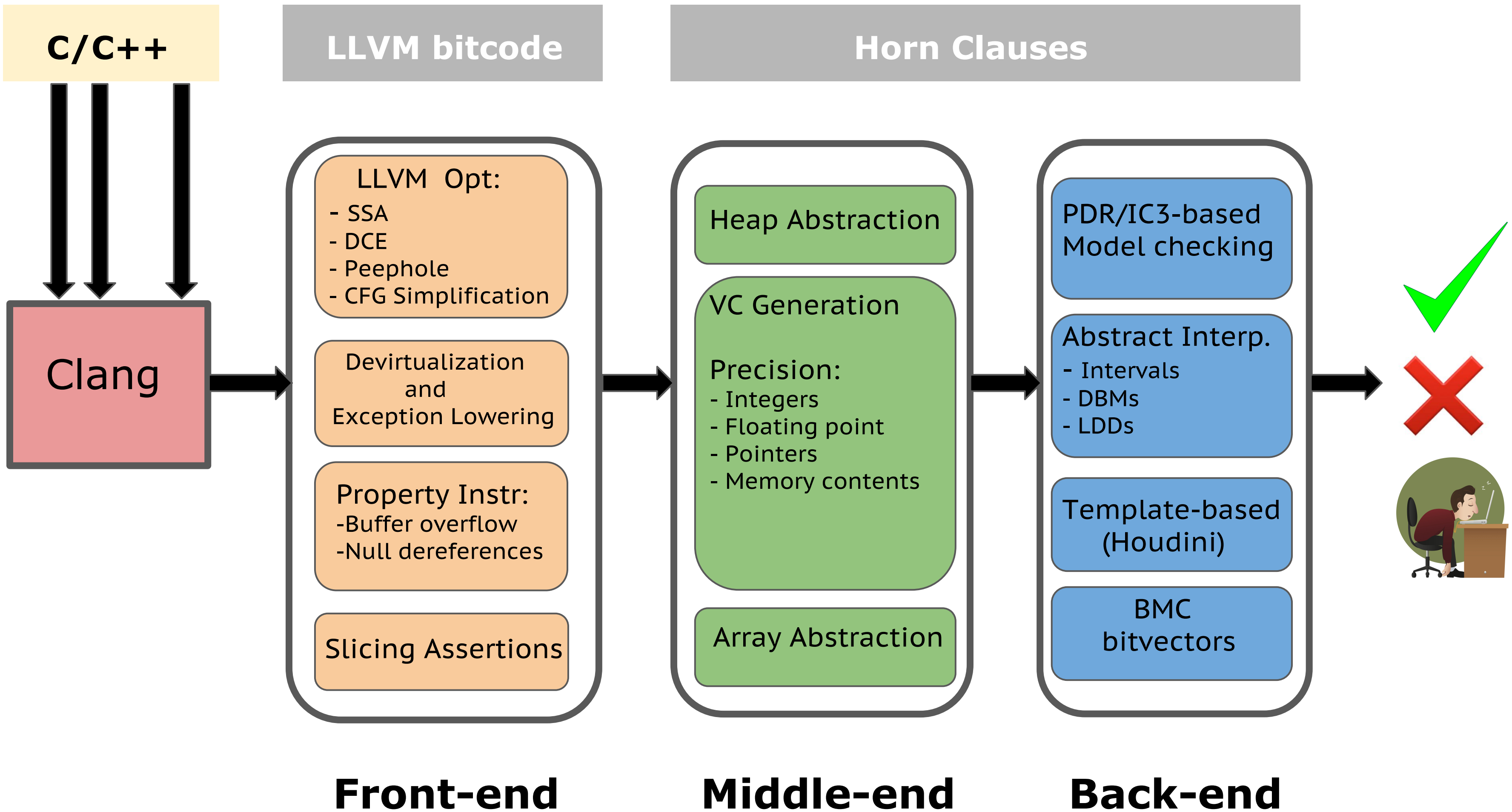
- understanding of verification techniques is useful (but not required)
- 3-6 month effort for a new property
  - but many things can be reused between similar properties
  - e.g., memory safety, null-dereference, taint checking, use-after-free, etc.

SeaHorn property checkers:

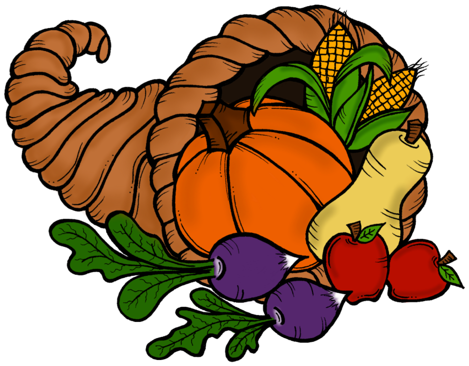
- memory safety (out of bound uses, null pointer)
  - ongoing work to improve scalability and usability
- taint analysis (being developed by Princeton)

DEMO

Architecture of Seahorn



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

- fixedpoint 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

Crab Abstract Domains

Numerical Domains

- interval with congruence: `0 <= x <= 10 && x mod 2 == 0`
- zone: `x – y <= k`
- non-convex
  - DisIntervals: `x <= -1 || x >= 1`
  - Boxes: Boolean combinations of intervals

Symbolic Domains

- numeric domains extended with uninterpreted functions
- `0 <= x <= 10 && y == f(…) && z == f(…)` ➔ `0 <= x <= 10 && y = z`

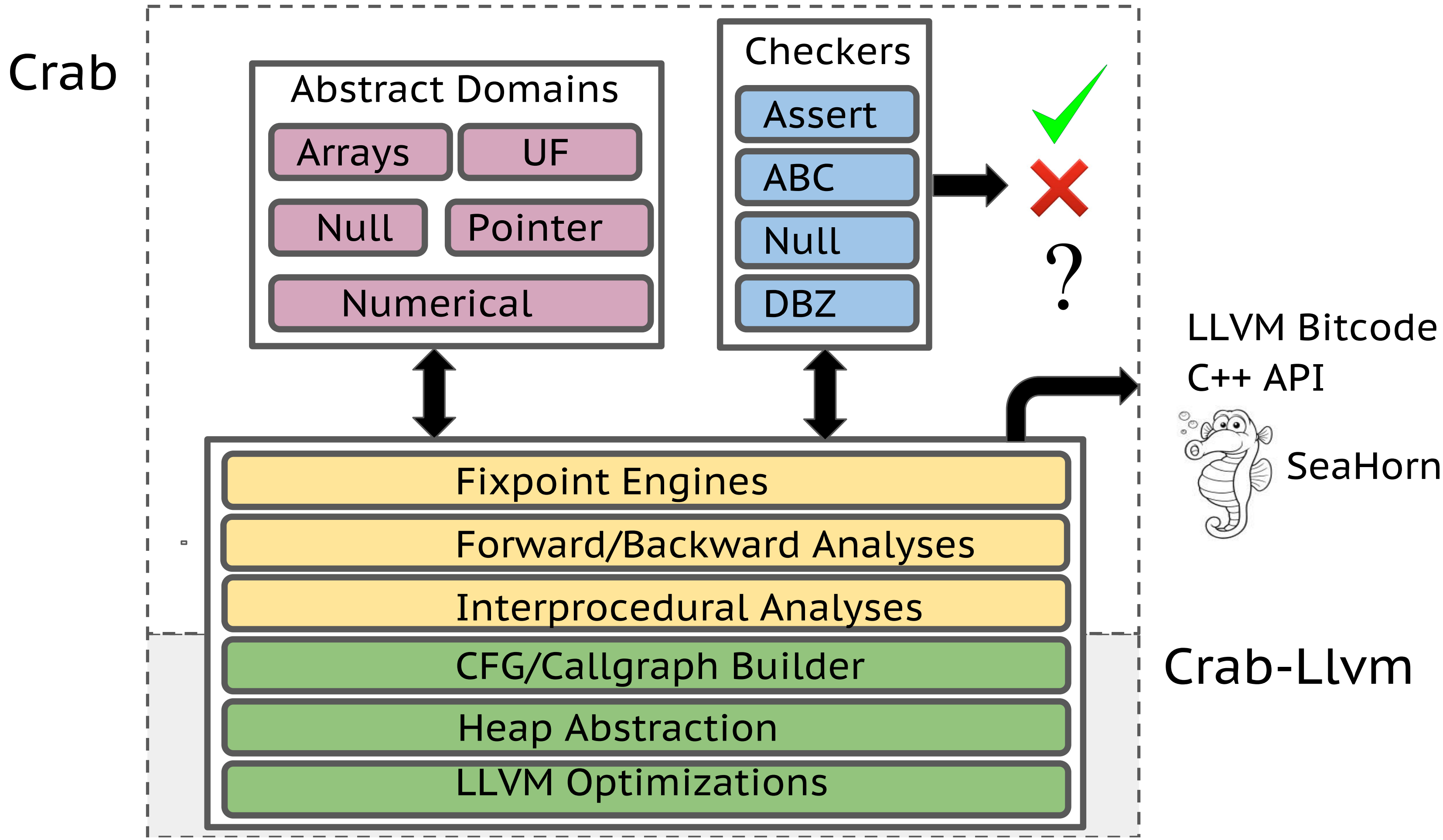
Array Domains

- array smashing: common properties of all array cells
- array graph domain:

Domains from Apron and Elina 3rd party libraries

- octagons, polyhedra, etc.

Architecture of Crab and Crab-Llvm



UNIVERSITY OF WATERLOO

https://github.com/seahorn/crab-llvm

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

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

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

Logic-based Program Verification

Low-Level Bounded Model Checking (BMC)

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

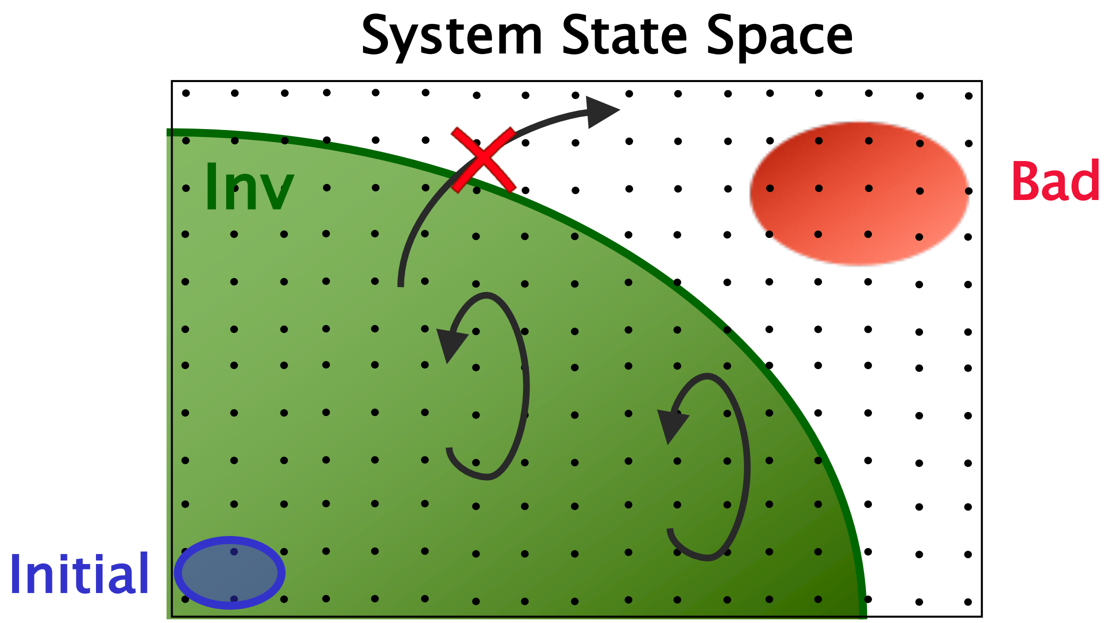
High-Level (Word-Level) Bounded Model Checking

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

What is an SMT-like equivalent for Safety Verification?

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

Inductive Invariants

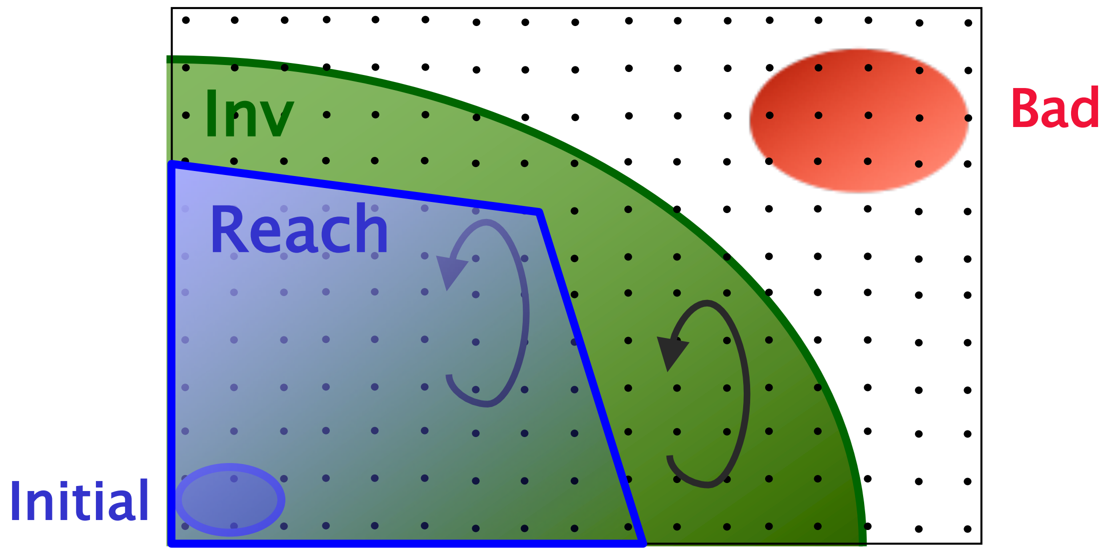


System S is safe iff there exists an inductive invariant Inv:

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

i.e., if s ∈ Inv and s⇝t then t ∈ Inv

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 = ∅

Symbolic Reachability Problem

P = (V, 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.*

$$Init \Rightarrow Inv$$

$$Inv(X) \wedge Tr(X, X') \Rightarrow Inv(X')$$

Inductive

$$Inv \Rightarrow \neg Bad$$

Safe

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

UNIVERSITY OF WATERLOO

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:

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

```
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]:

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

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 $R_1, \ldots, R_N$ over V and $E_1, \ldots, E_N$ over V, V',

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

multi-threaded program P is safe

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

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

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

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

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

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

Figure 4: Horn constraints encoding a homogeneous infinite system with the help of a k-indexed invariant. S_k is the symmetric group on $\{1, \ldots, k\}$, 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) $\mathsf{init}(g, x_1) \wedge \cdots \wedge \mathsf{init}(g, x_n) \rightarrow Inv(g, \ell_{\mathsf{init}}, x_1, \ldots, \ell_{\mathsf{init}}, x_k)$

(inductive) $Inv(g, \ell_1, x_1, \ldots, \ell_i, x_i, \ldots, \ell_k, x_k) \wedge s(g, x_i, g', x_i') \rightarrow Inv(g', \ell_1, x_1, \ldots, \ell_i', x_i', \ldots, \ell_k,$

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

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

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 an 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

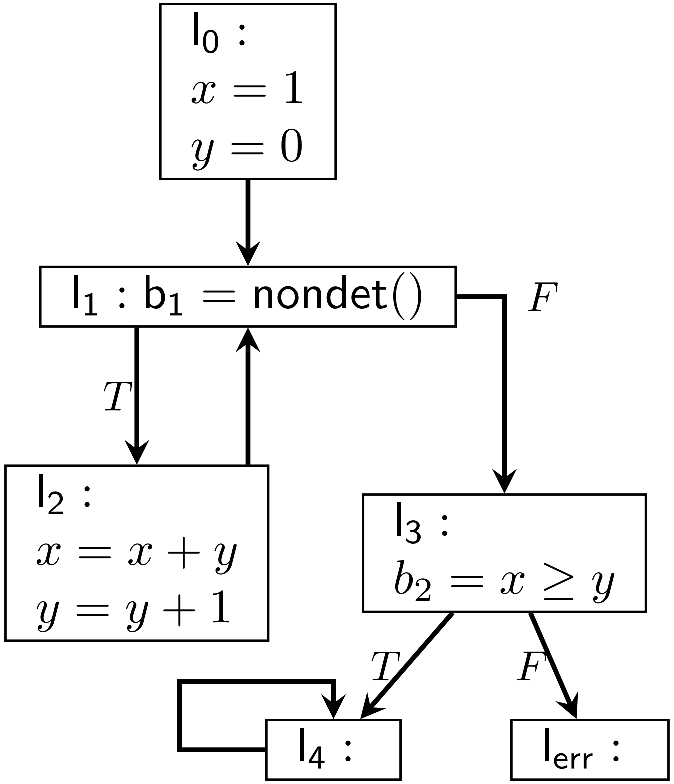
UNIVERSITY OF WATERLOO

From Programs to Logic

Program

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

CFG

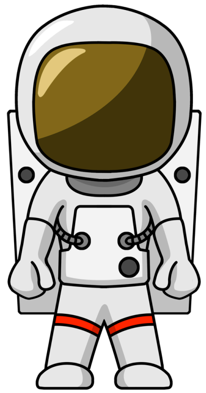


CHC

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

UNIVERSITY OF WATERLOO

Spacer: Solving SMT-constrained CHC



Spacer: a solver for SMT-constrained Horn Clauses

- now part of Z3
  - https://github.com/Z3Prover/z3 since commit `72c4780`
  - use option `fixedpoint.engine=spacer`
- development version at http://bitbucket.org/spacer/code

Supported SMT-Theories

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

Support for Non-Linear CHC

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

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

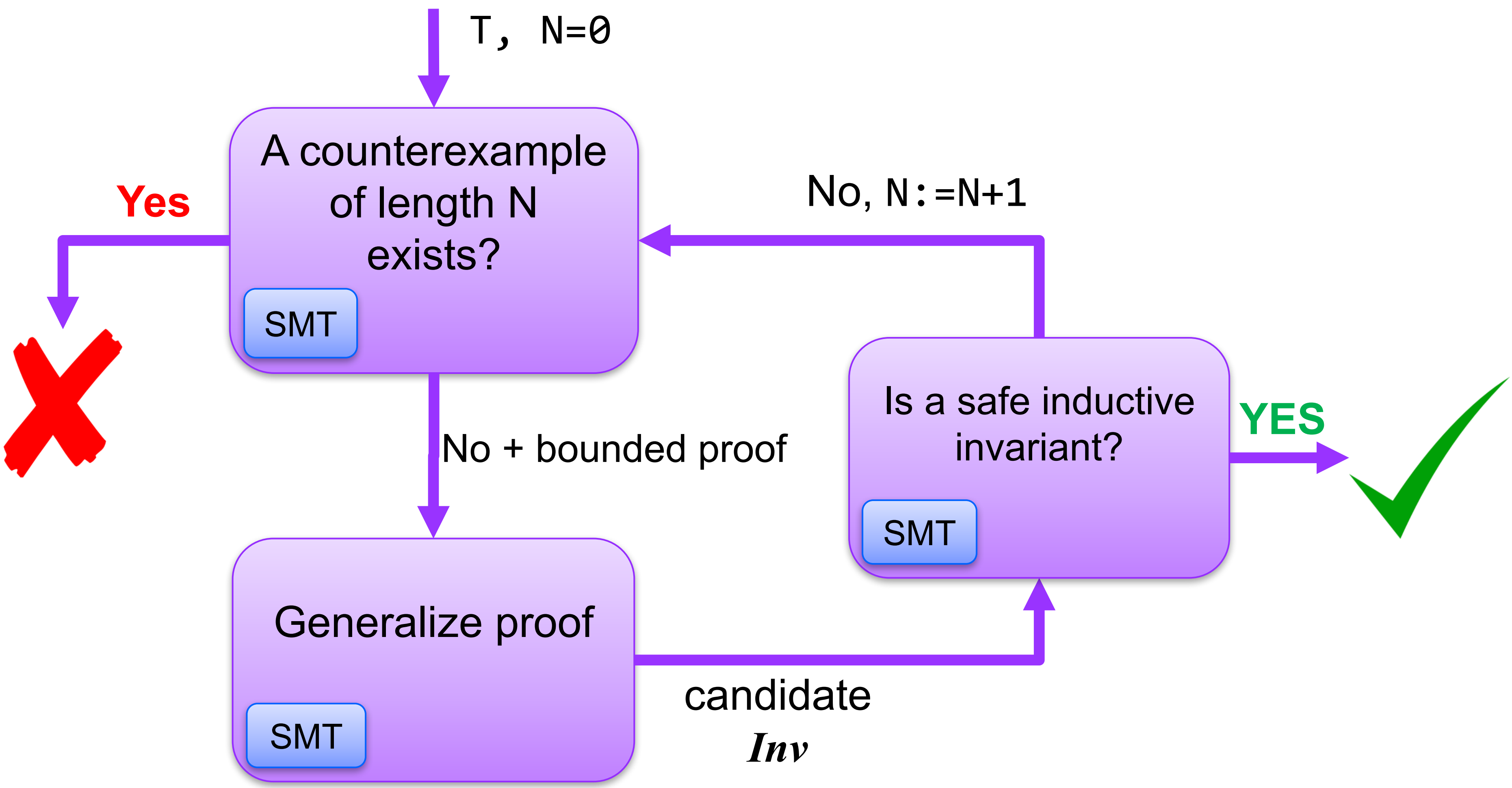
PolyPDR: Convex models for Linear CHC

- 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

Generalizing from Bounded Proofs



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*;

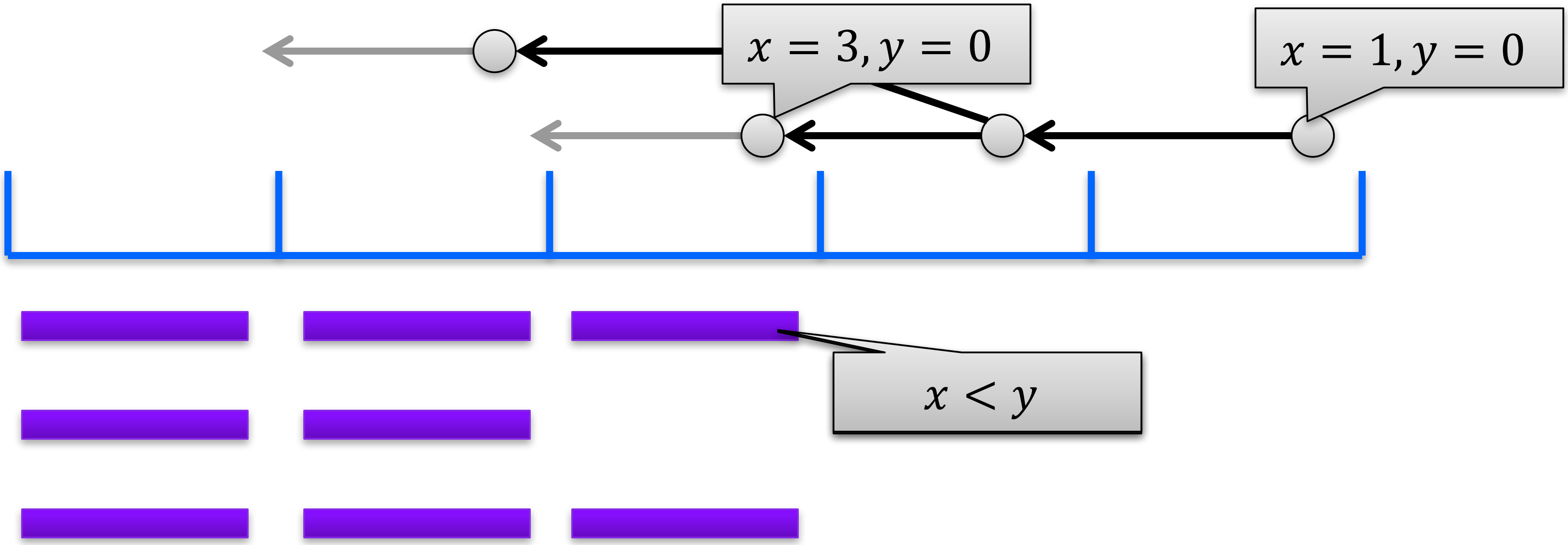
$\quad N \leftarrow N + 1 \,;\, F_N \leftarrow \emptyset$

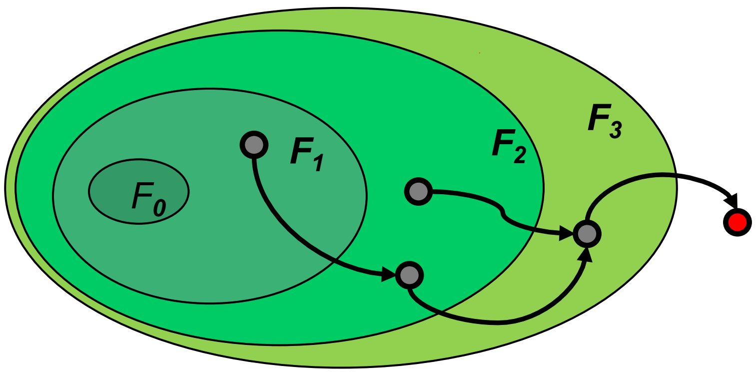
until ∞;

strengthen result

Spacer/IC3/PDR In Pictures: MkSafe

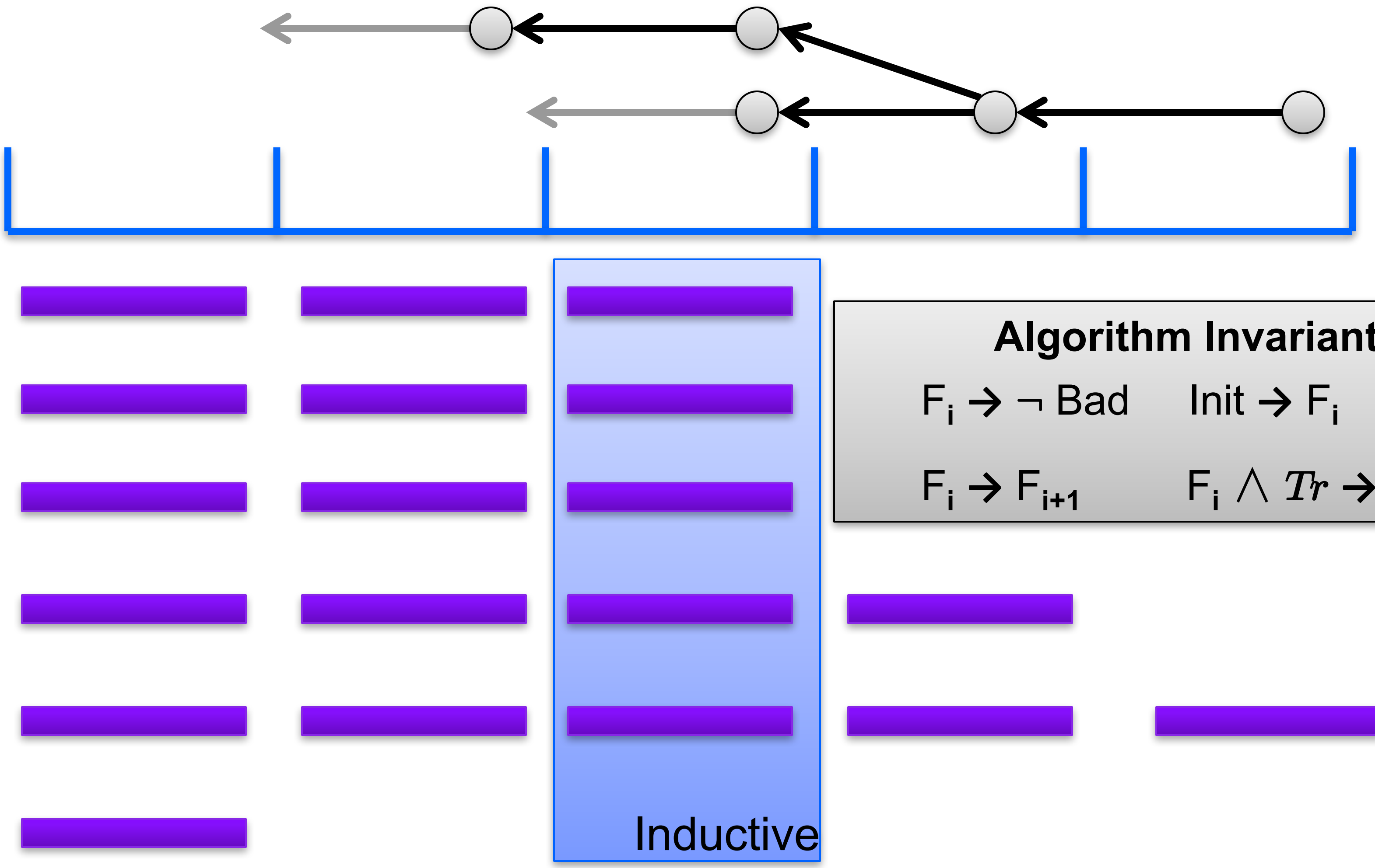
MkSafe





Spacer/IC3/PDR in Pictures: Push

Push

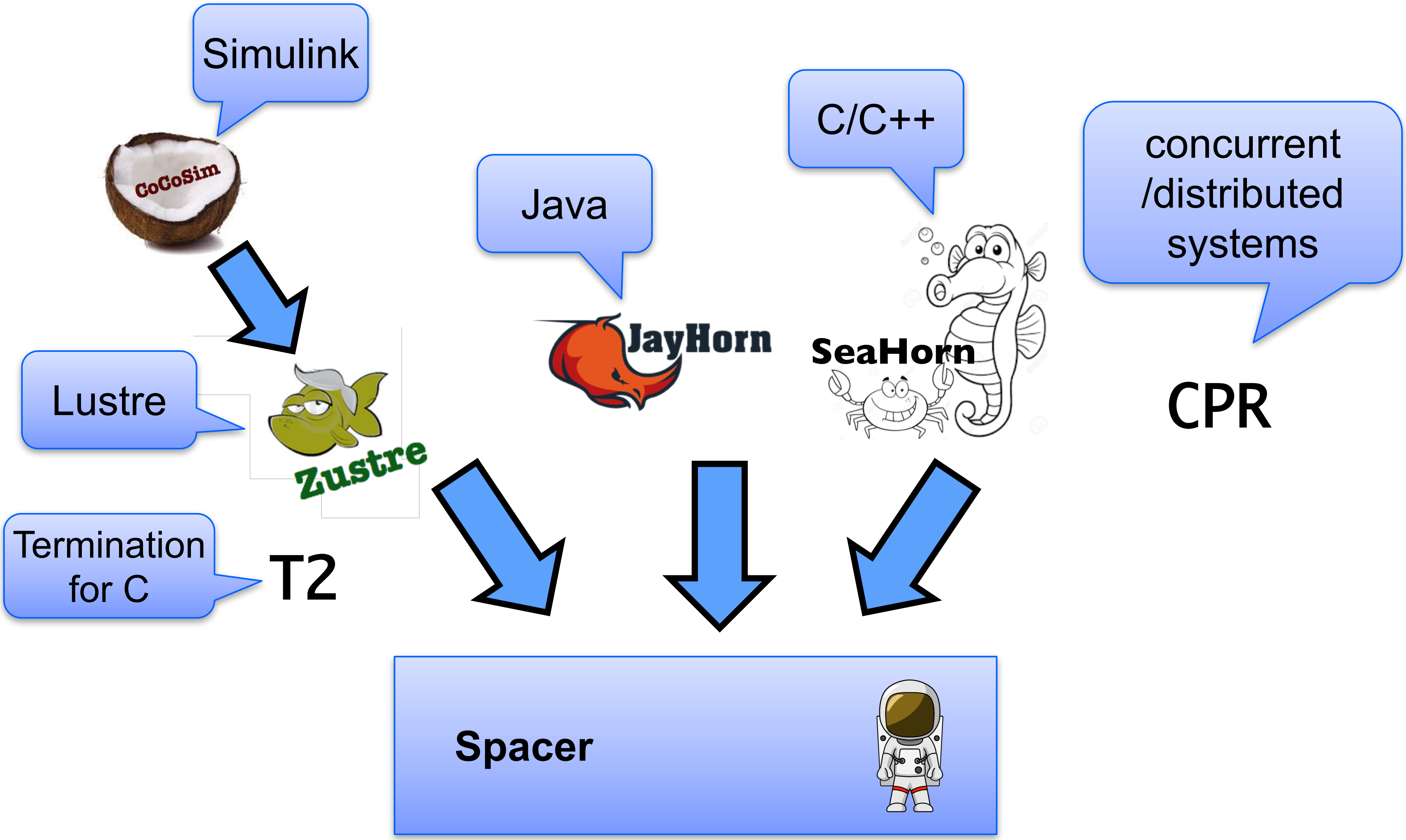


Algorithm Invariants

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

$F_i \rightarrow F_{i+1}$ $F_i \wedge \mathit{Tr} \rightarrow F'_{i+1}$

Logic-based Algorithmic Verification



SV-COMP 2015

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

4th Competition on Software Verification held 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.

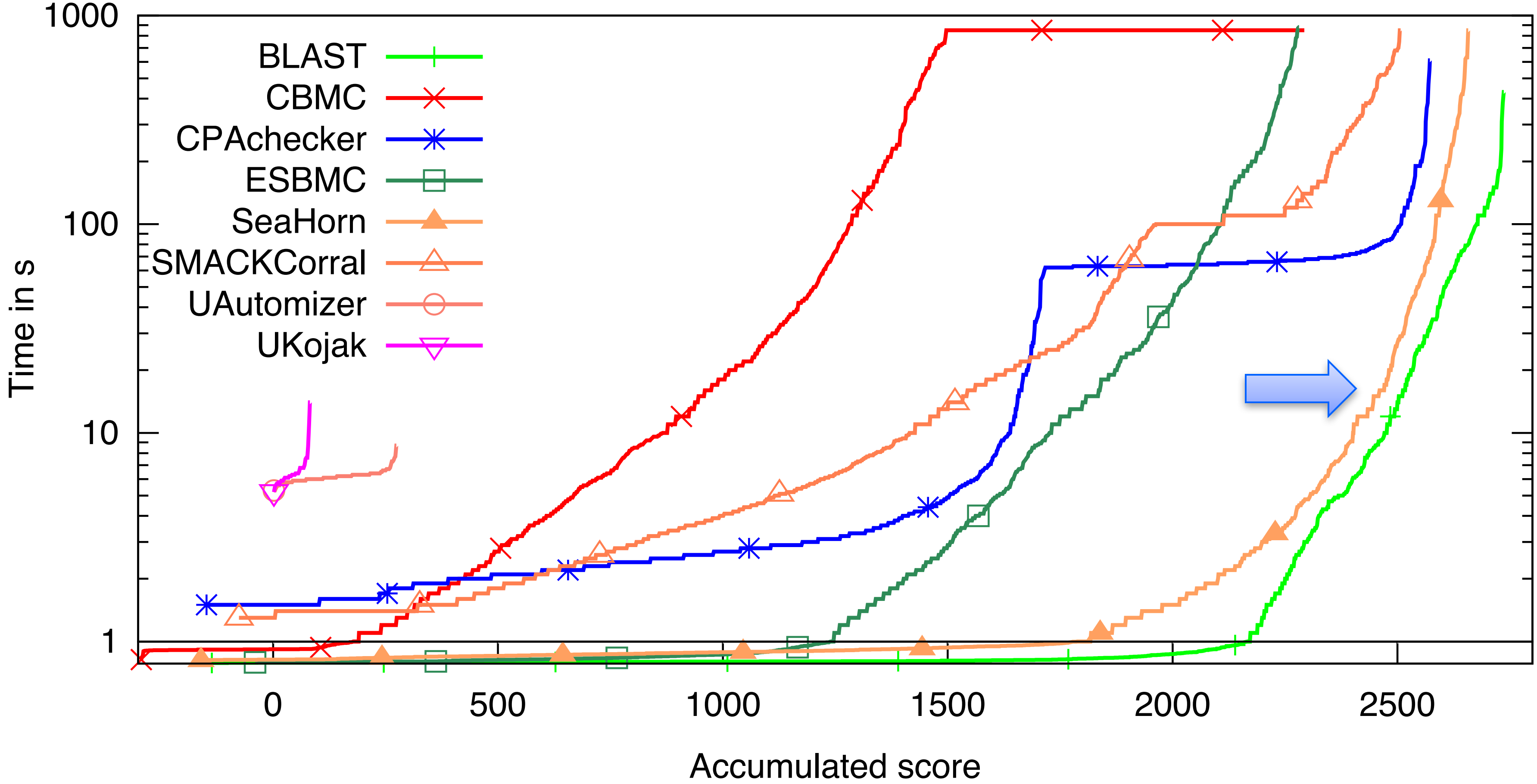
Participants:

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

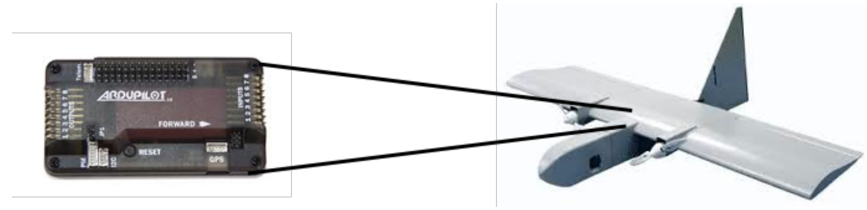
Benchmarks:

- 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



Applications of SeaHorn at NASA

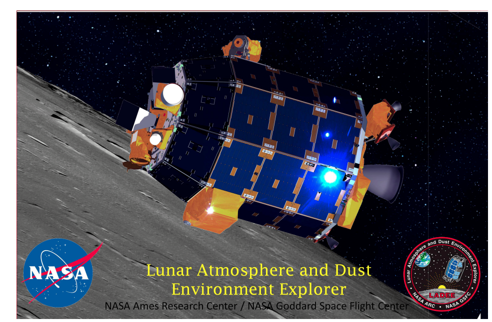


Absence of Buffer Overflows

- Open source auto-pilots
  - paparazzi and mnav autopilots
- Automatically instrument buffer accesses with runtime checks
- Use SeaHorn to validate that run-time checks never fail
  - slower than pure abstract interpretation
  - BUT, much more precise!

Verify Level 5 requirements of the LADEE software stack

- Manually encode requirements in Simulink model
- Verify that the requirements hold in auto-generated C



Memory safety of C++ controller code

- ongoing…

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

Advanced 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

Current and Future Work

Precise Memory Analysis

- pointer / alias analysis for LLVM
- bug discovery using symbolic execution
- verification of buffer overflows, null-deref, memory safety
- specialized checkers / proof rules / verification conditions

Verification of Concurrent / Distributed / Parametrized Systems

- modular verification (per thread, per task, per node)
- scale to systems with large / unbounded interacting components

Scalability and Precision

- develop and implement new algorithms to increase scalability and/or precision
- effective modular reasoning / slicing / lemma learning
- bit-precise verification

UNIVERSITY OF WATERLOO

References

Tools:

- SeaHorn: http://seahorn.github.io/

Papers:

- Blog: http://seahorn.github.io/blog/
- A. Gurfinkel, T. Kahsai, J.A. Navas: **Algorithmic logic-based verification**. SIGLOG News 2(2): 29-38 (2015)
- A. Gurfinkel, T. Kahsai, A. Komuravelli, J.A. Navas: **The SeaHorn Verification Framework**. CAV (1) 2015: 343-361
- A. Komuravelli, A. Gurfinkel, S. Chaki: **SMT-based model checking for recursive programs**. Formal Methods in System Design 48(3): 175-205 (2016)
- A. Gurfinkel, J.A. Navas: **A Context-Sensitive Memory Model for Verification of C/C++ Programs**. SAS 2017: 148-168
- C. Urban, A. Gurfinkel, T. Kahsai: **Synthesizing Ranking Functions from Bits and Pieces**. TACAS 2016: 54-70A.
- Gurfinkel, S. Chaki: **Boxes: A Symbolic Abstract Domain of Boxes**. SAS 2010: 287-303

?

?

?

?

?

?

2000 started PhD in MC at UofT

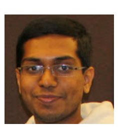
multi-valued model checking

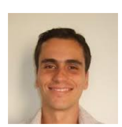
2006 SMC Yasm: safety, liveness, multi-valued abstraction for MC

2010 Boxes abstract domain (SAS'10)


2012 UFO: MC + AI: SAS'12

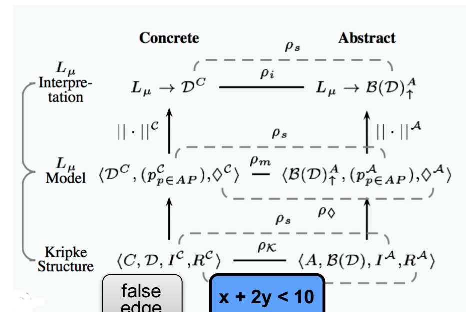

2015 SeaHorn: MC (Spacer) **and** AI (Crab)

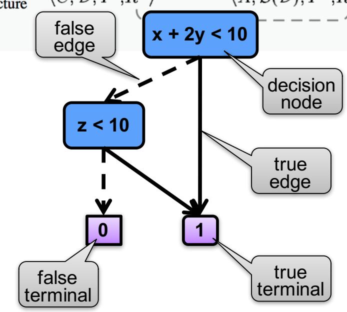


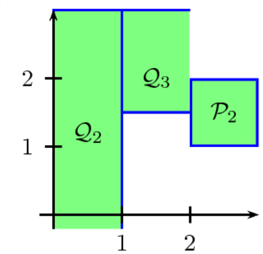



SLAM BLAST VMCAI CBMC

VMCAI'06

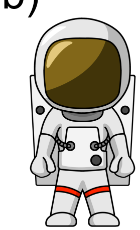






UFO CPA✓ SMACK

SV-COMP




UNIVERSITY OF WATERLOO