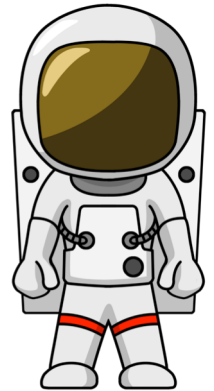


Spacer: Model Checking in the land of Theorem Provers



Arie Gurfinkel

Electrical and Computer Engineering
University of Waterloo

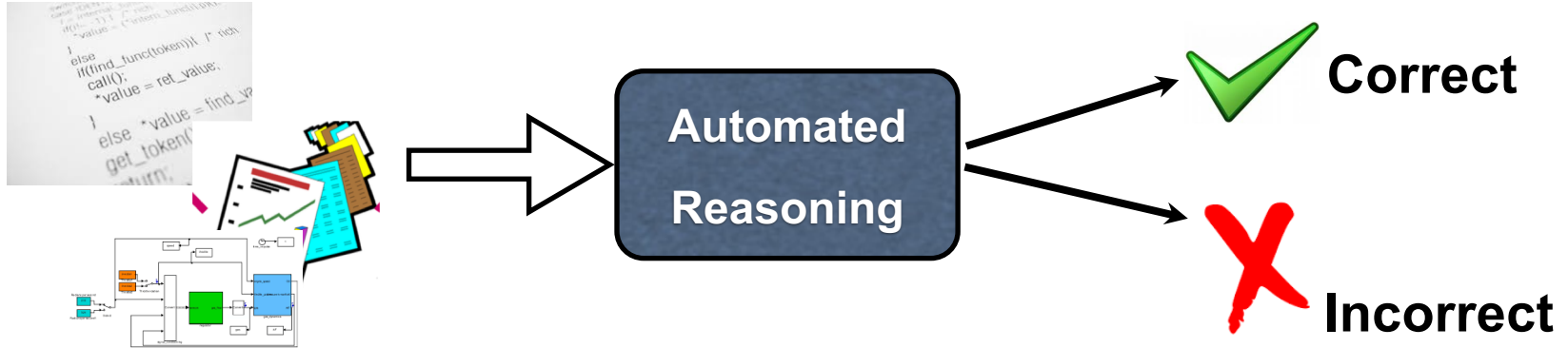
Coverif Workshop on Combining Abstract
Interpretation and Constraint Programming
CIRM, September 25 – 28, 2018



UNIVERSITY OF
WATERLOO

Automated (Software) Verification

Program and/or model





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.

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



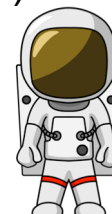
UFO

CPA✓

SMACK

SV-COMP

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



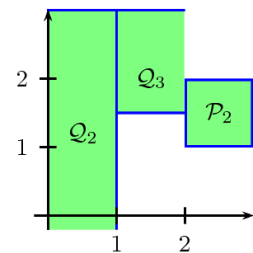
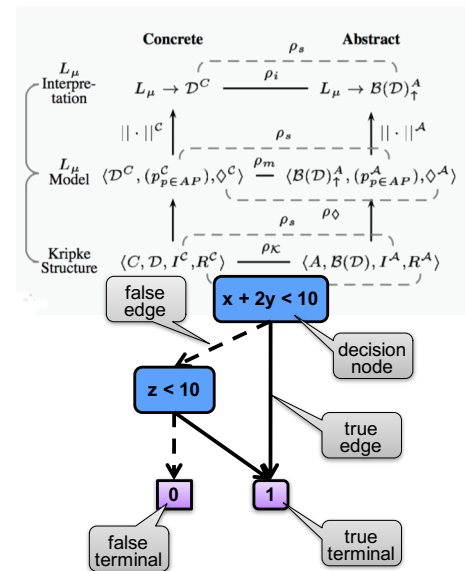
SLAM
 if (#node > 1) { i += 1; while (true) { node; } }

BLAST

VMCAI

CBMC

VMCAI'06



Symbolic Reachability Problem

$$P = (X, \textit{Init}, \textit{Tr}, \textit{Bad})$$

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

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

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

$$\left. \begin{array}{l} \textit{Init} \Rightarrow \textit{Inv} \\ \textit{Inv}(X) \wedge \textit{Tr}(X, X') \Rightarrow \textit{Inv}(X') \\ \textit{Inv} \Rightarrow \neg \textit{Bad} \end{array} \right\} \begin{array}{l} \text{Inductive} \\ \text{Safe} \end{array}$$

Constrained Horn Clauses (CHC)

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

$$\forall V \cdot (\varphi \wedge p_1[X_1] \wedge \cdots \wedge p_n[X_n]) \rightarrow h[X]$$

where

- \mathcal{T} is a background theory (e.g., Linear Arithmetic, Arrays, Bit-Vectors, or combinations of the above)
- V are variables, and X_i are terms over V
- φ is a constraint in the background theory \mathcal{T}
- p_1, \dots, p_n, h are n -ary predicates
- $p_i[X]$ is an application of a predicate to first-order terms

CHC Satisfiability

A \mathcal{T} -**model** of a set of CHCs Π is an extension of the model M of \mathcal{T} with a first-order interpretation of each predicate p_i that makes all clauses in Π true in M

A set of clauses is **satisfiable** if and only if it has a model

- This is the usual FOL satisfiability

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

In the context of program verification

- a program satisfies a property iff corresponding CHCs are satisfiable
- solutions are inductive invariants
- refutation proofs are counterexample traces

Satisfiability Modulo Theory (SMT)

Satisfiability is the problem of determining whether a formula F has a model

- if F is **propositional**, a model is a truth assignment to Boolean variables
- if F is **first-order formula**, a model assigns values to variables and interpretation to all the function and predicate symbols

SAT Solvers

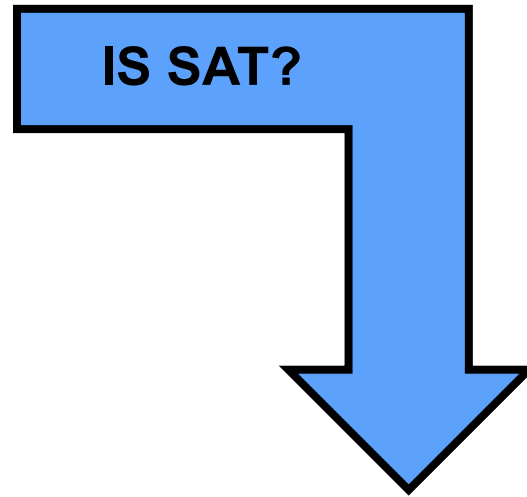
- check satisfiability of propositional formulas

SMT Solvers

- check satisfiability of formulas in a **decidable** first-order theory (e.g., linear arithmetic, uninterpreted functions, array theory, bit-vectors)

Program Verification with HORN(LIA)

```
z = x; i = 0;  
assume (y > 0);  
while (i < y) {  
    z = z + 1;  
    i = i + 1;  
}  
assert(z == x + y);
```



$z = x \ \& \ i = 0 \ \& \ y > 0$	\rightarrow	$\text{Inv}(x, y, z, i)$
$\text{Inv}(x, y, z, i) \ \& \ i < y \ \& \ z1=z+1 \ \& \ i1=i+1$	\rightarrow	$\text{Inv}(x, y, z1, i1)$
$\text{Inv}(x, y, z, i) \ \& \ i \geq y \ \& \ z \neq x+y$	\rightarrow	false

In SMT-LIB

```
(set-logic HORN)

;; Inv(x, y, z, i)
(declare-fun Inv ( Int Int Int Int) Bool)

(assert
  (forall ( (A Int) (B Int) (C Int) (D Int))
    (=> (and (> B 0) (= C A) (= D 0))
      (Inv A B C D)))
)
(assert
  (forall ( (A Int) (B Int) (C Int) (D Int) (C1 Int) (D1 Int) )
    (=>
      (and (Inv A B C D) (< D B) (= C1 (+ C 1)) (= D1 (+ D
1))))
    (Inv A B C1 D1)
  )
)
(assert
  (forall ( (A Int) (B Int) (C Int) (D Int))
    (=> (and (Inv A B C D) (>= D B) (not (= C (+ A B)))))
      false
    )
  )
)

(check-sat)
(get-model)
```

\$ z3 add-by-one.smt2

```
sat
(model
  (define-fun Inv ((x!0 Int) (x!1 Int) (x!2 Int) (x!3 Int)) Bool
    (and (<= (+ x!2 (* (- 1) x!0) (* (- 1) x!3)) 0)
      (<= (+ x!2 (* (- 1) x!0) (* (- 1) x!1)) 0)
      (<= (+ x!0 x!3 (* (- 1) x!2)) 0)))
  )
```

$\text{Inv}(x, y, z, i)$

$z = x + i$

$z \leq x + y$

Procedures for Solving CHC(T)

Predicate abstraction by lifting Model Checking to HORN

- QARMC, Eldarica, ...

Maximal Inductive Subset from a finite Candidate space (Houdini)

- TACAS'18: hoice, FreqHorn

Machine Learning

- PLDI'18: sample, ML to guess predicates, DT to guess combinations

Abstract Interpretation (Poly, intervals, boxes, arrays...)

- Approximate least model by an abstract domain (SeaHorn, ...)

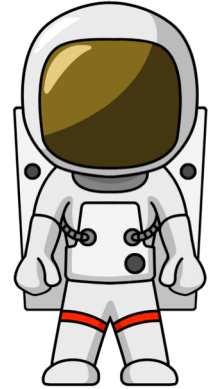
Interpolation-based Model Checking

- Duality, QARMC, ...

SMT-based Unbounded Model Checking (IC3/PDR)

- Spacer, Implicit Predicate Abstraction

Spacer: Solving SMT-constrained CHC



Spacer: a solver for SMT-constrained Horn Clauses

- now the default (and only) CHC solver in Z3
 - <https://github.com/Z3Prover/z3>
 - dev branch at <https://github.com/agurfinkel/z3>

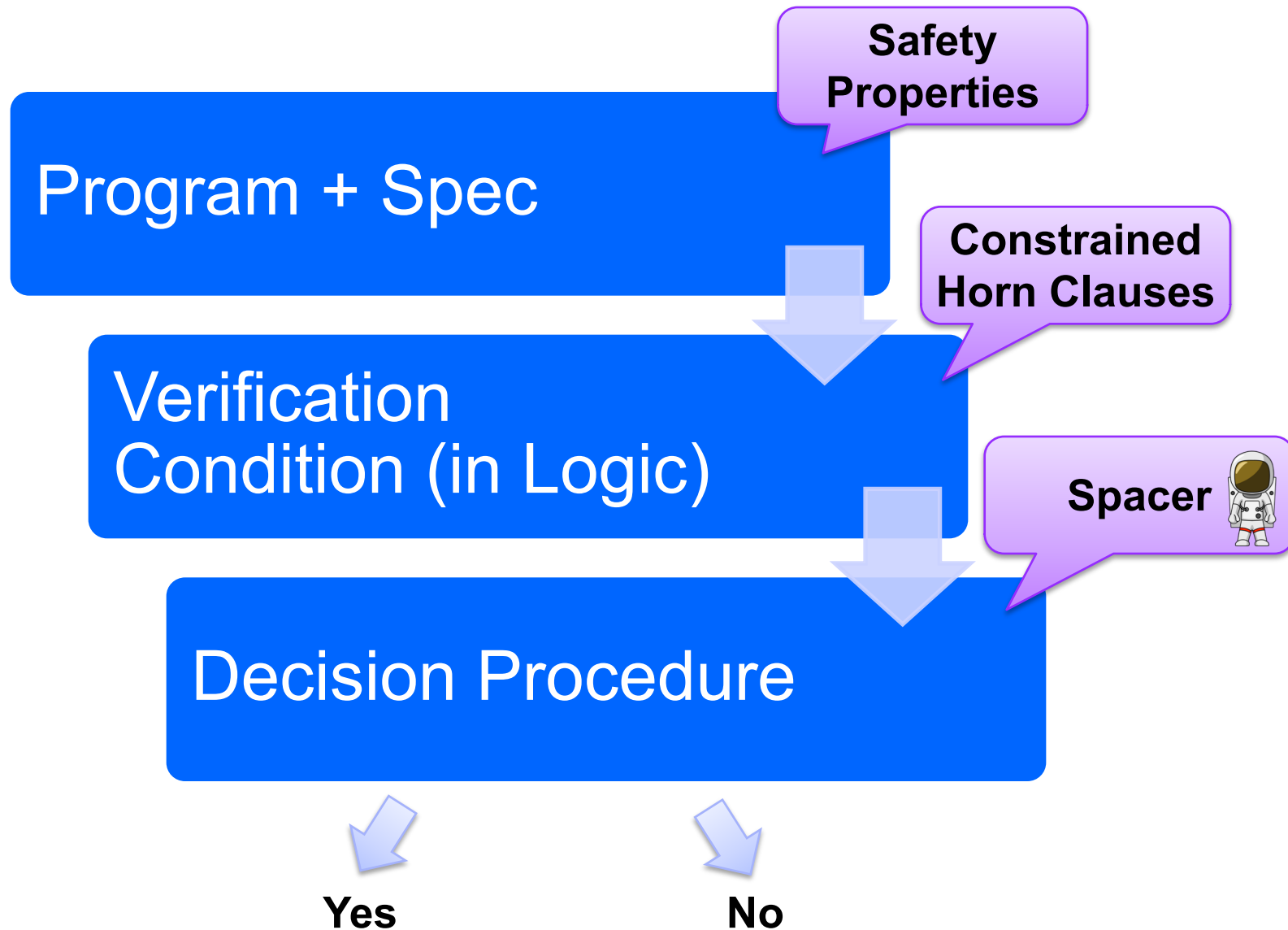
Supported SMT-Theories

- Linear Real and Integer Arithmetic
- Quantifier-free theory of arrays
- ***Universally quantified theory of arrays + arithmetic (NEW: ATVA18)***
- Best-effort support for many other SMT-theories
 - data-structures, bit-vectors, non-linear arithmetic

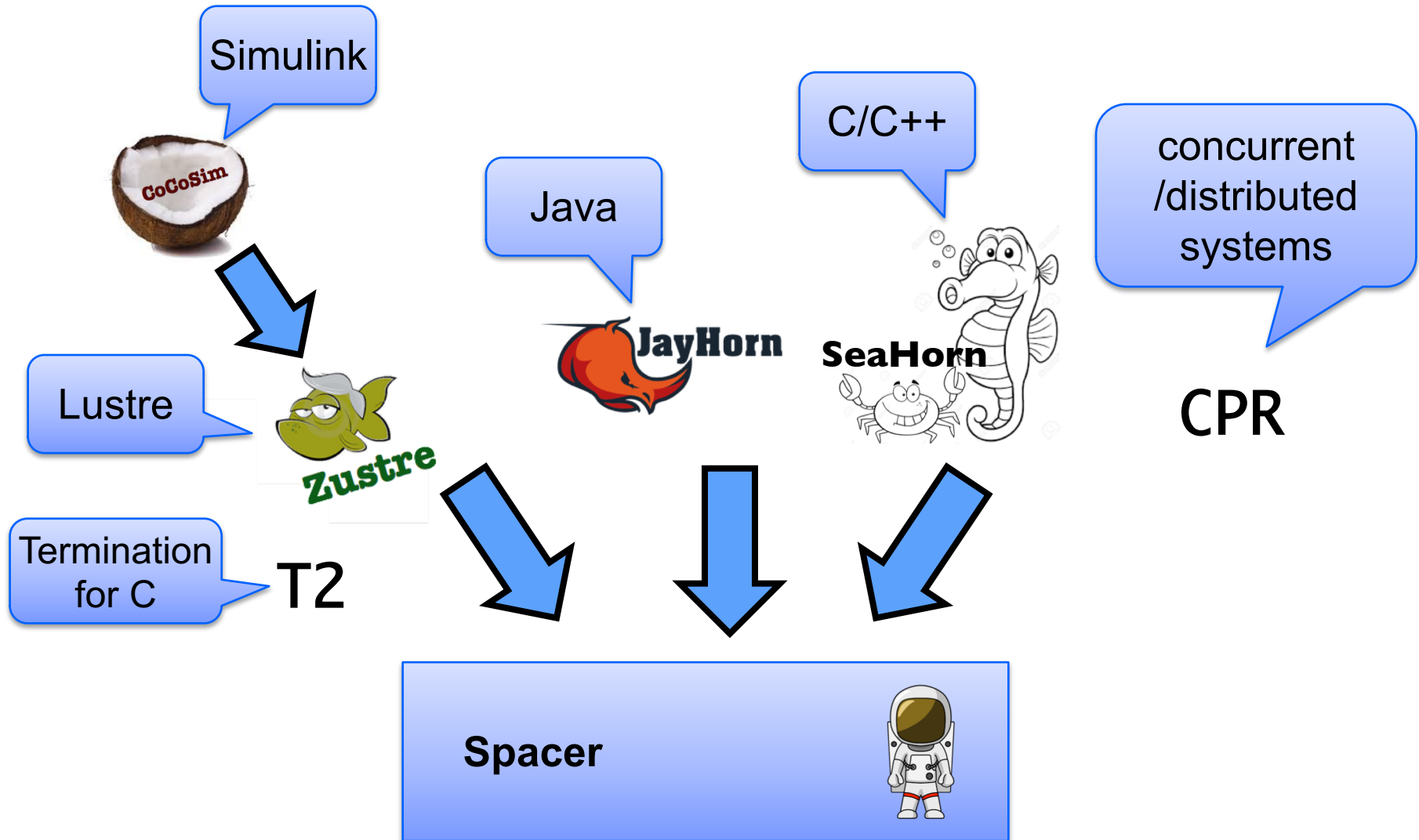
Support for Non-Linear CHC

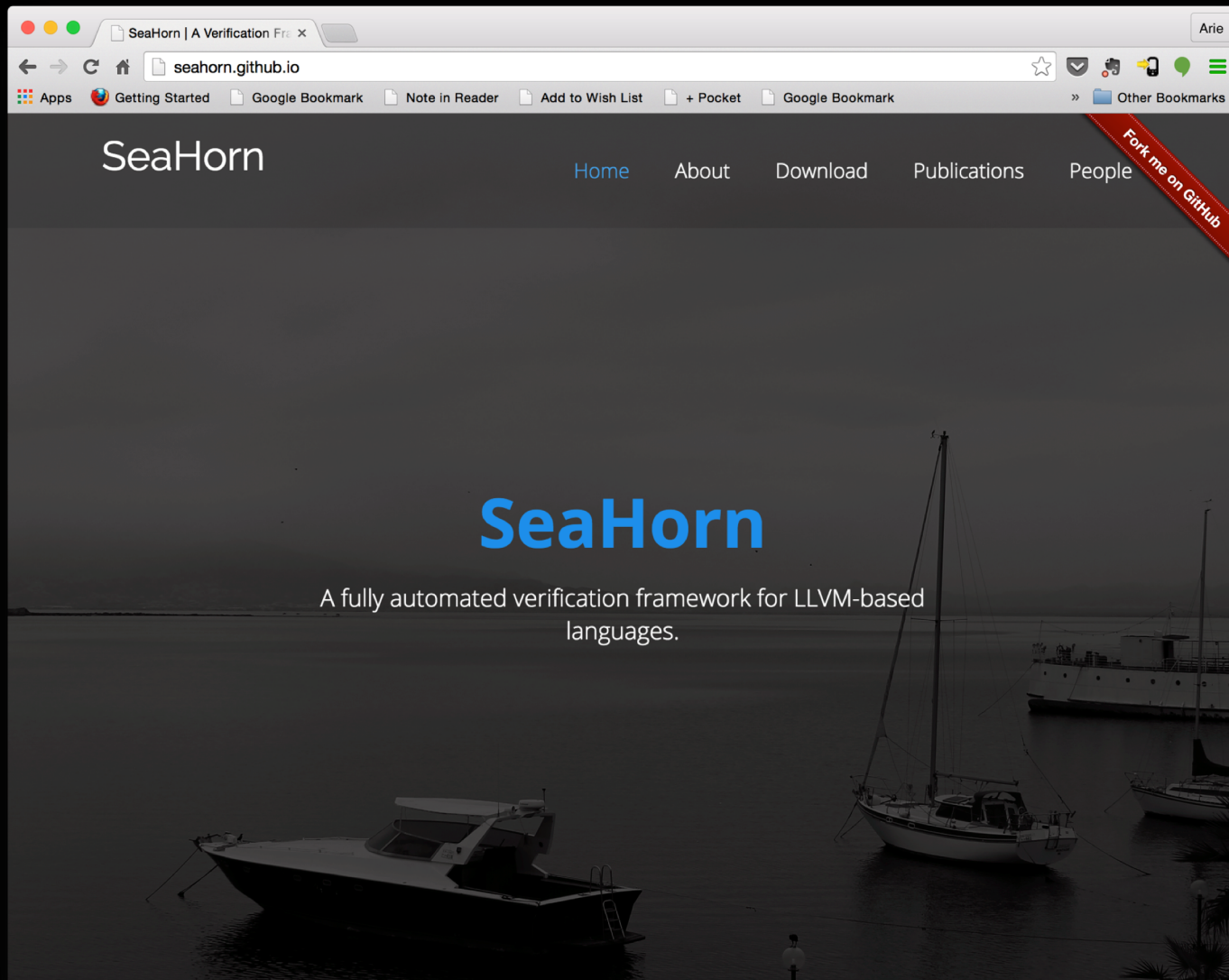
- for procedure summaries in inter-procedural verification conditions
- for compositional reasoning: abstraction, assume-guarantee, thread modular, etc.

Algorithmic Logic-Based Verification



Logic-based Algorithmic Verification





<http://seahorn.github.io>



SeaHorn at a glance

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

Industrial-strength front-end based on Clang and LLVM

Abstract Interpretation engine: **Crab**

SMT-based verification engine: **Spacer**

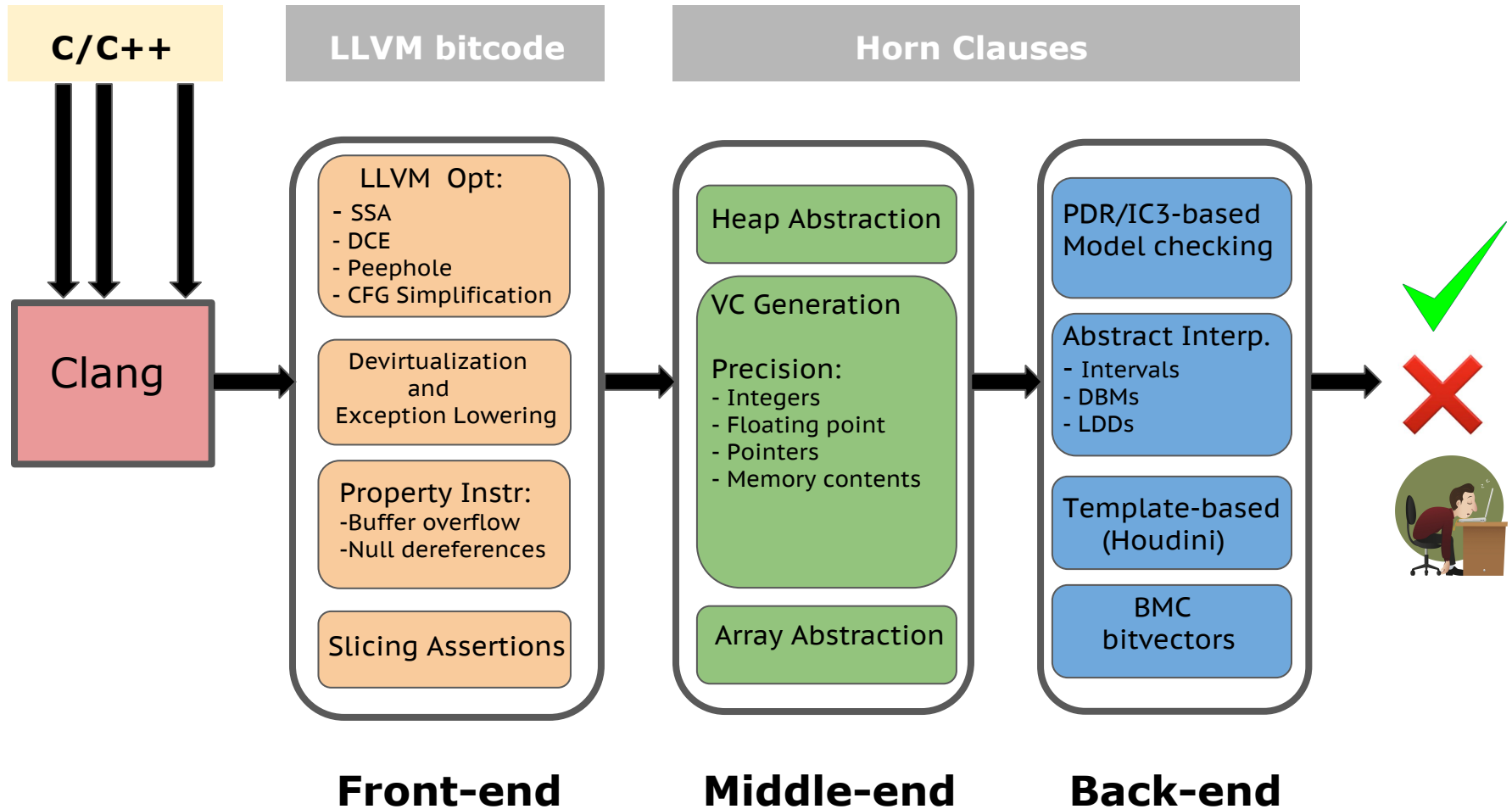
Bit-precise Bounded Model Checker and Symbolic Execution

Executable Counter-Examples

A framework for research and application of logic-based verification



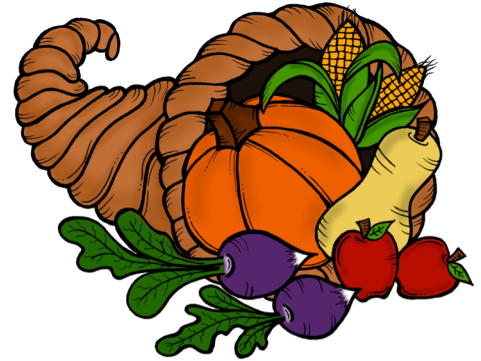
Architecture of Seahorn



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

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

Support for inter-procedural analysis

- pre-, post-conditions, function summaries

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

Horn Clauses for Program Verification

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

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

5. incorrect :- Z=W+1, W ≥ 0, W+1 < read(A, W, U), read(A, 2
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]:

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

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

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

Bjørner, Gurfinkel, McMillan, and Rybalchenko:
Horn Clause Solvers for Program Verification

Horn Clauses for Concurrent / Distributed / Parameterized Systems

For assertions R_1, \dots, R_N over V and E_1, \dots, E_N over V, V' ,

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

multi-threaded program P is safe

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

- (initial) $init(g, x_1) \wedge \dots \wedge init(g, x_n) \rightarrow Inv(g, \ell_{init}, x_1, \dots, \ell_{init}, x_k)$
 (inductive) $Inv(g, \ell_1, x_1, \dots, \ell_i, x_i, \dots, \ell_k, x_k) \wedge s(g, x_i, g', x'_i) \rightarrow Inv(g', \ell_1, x_1, \dots, \ell'_i, x'_i, \dots, \ell_k, x_k)$
 (non-interference) $Inv(g, \ell_1, x_1, \dots, \ell_k, x_k) \wedge$
 $Inv(g, \ell^\dagger, x^\dagger, \ell_2, x_2, \dots, \ell_k, x_k) \wedge$
 \vdots
 $Inv(g, \ell_1, x_1, \dots, \ell_{k-1}, x_{k-1}, \ell^\dagger, x^\dagger) \wedge s(g, x^\dagger, g', \cdot) \rightarrow Inv(g', \ell_1, x_1, \dots, \ell_k, x_k)$
 (safe) $Inv(g, \ell_1, x_1, \dots, \ell_k, x_k) \wedge err(g, \ell_1, x_1, \dots, \ell_m, x_m) \rightarrow false$

Figure 6. Horn clause encoding for thread modularity at level k (where (ℓ_i, s, ℓ'_i) and (ℓ^\dagger, s, \cdot) refer to statement s on a thread from ℓ_i to ℓ'_i and, respectively, from ℓ^\dagger to some other location in the control flow graph)

Hoenicke et al. Thread Modularity at Many Levels. POPL'17

$$\left\{ R(g, p_{\sigma(1)}, l_{\sigma(1)}, \dots, p_{\sigma(k)}, l_{\sigma(k)}) \leftarrow dist(p_1, \dots, p_k) \wedge R(g, p_1, l_1, \dots, p_k, l_k) \right\}_{\sigma \in S_k} \quad (6)$$

$$R(g, p_1, l_1, \dots, p_k, l_k) \leftarrow dist(p_1, \dots, p_k) \wedge Init(g, l_1) \wedge \dots \wedge Init(g, l_k) \quad (7)$$

$$R(g', p_1, l'_1, \dots, p_k, l_k) \leftarrow dist(p_1, \dots, p_k) \wedge ((g, l_1) \xrightarrow{p_1} (g', l'_1)) \wedge R(g, p_1, l_1, \dots, p_k, l_k) \quad (8)$$

$$R(g', p_1, l_1, \dots, p_k, l_k) \leftarrow dist(p_0, p_1, \dots, p_k) \wedge ((g, l_0) \xrightarrow{p_0} (g', l'_0)) \wedge RConj(0, \dots, k) \quad (9)$$

$$false \leftarrow dist(p_1, \dots, p_r) \wedge \left(\bigwedge_{j=1, \dots, m} (p_j = p_j \wedge (g, l_j) \in E_j) \right) \wedge RConj(1, \dots, r) \quad (10)$$

Figure 4: Horn constraints encoding a homogeneous infinite system with the help of a k -indexed invariant. S_k is the symmetric group on $\{1, \dots, k\}$, i.e., the group of all permutations of k numbers; as an optimisation, any generating subset of S_k , for instance transpositions, can be used instead of S_k . In (10), we define $r = \max\{m, k\}$.

Hojjat et al. Horn Clauses for Communicating Timed Systems. HCVS'14

$$Init(i, j, \bar{v}) \wedge Init(j, i, \bar{v}) \wedge$$

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

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

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

$$I_2(i, j, \bar{v}) \wedge I_2(i, k, \bar{v}) \wedge I_2(j, k, \bar{v}) \wedge$$

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

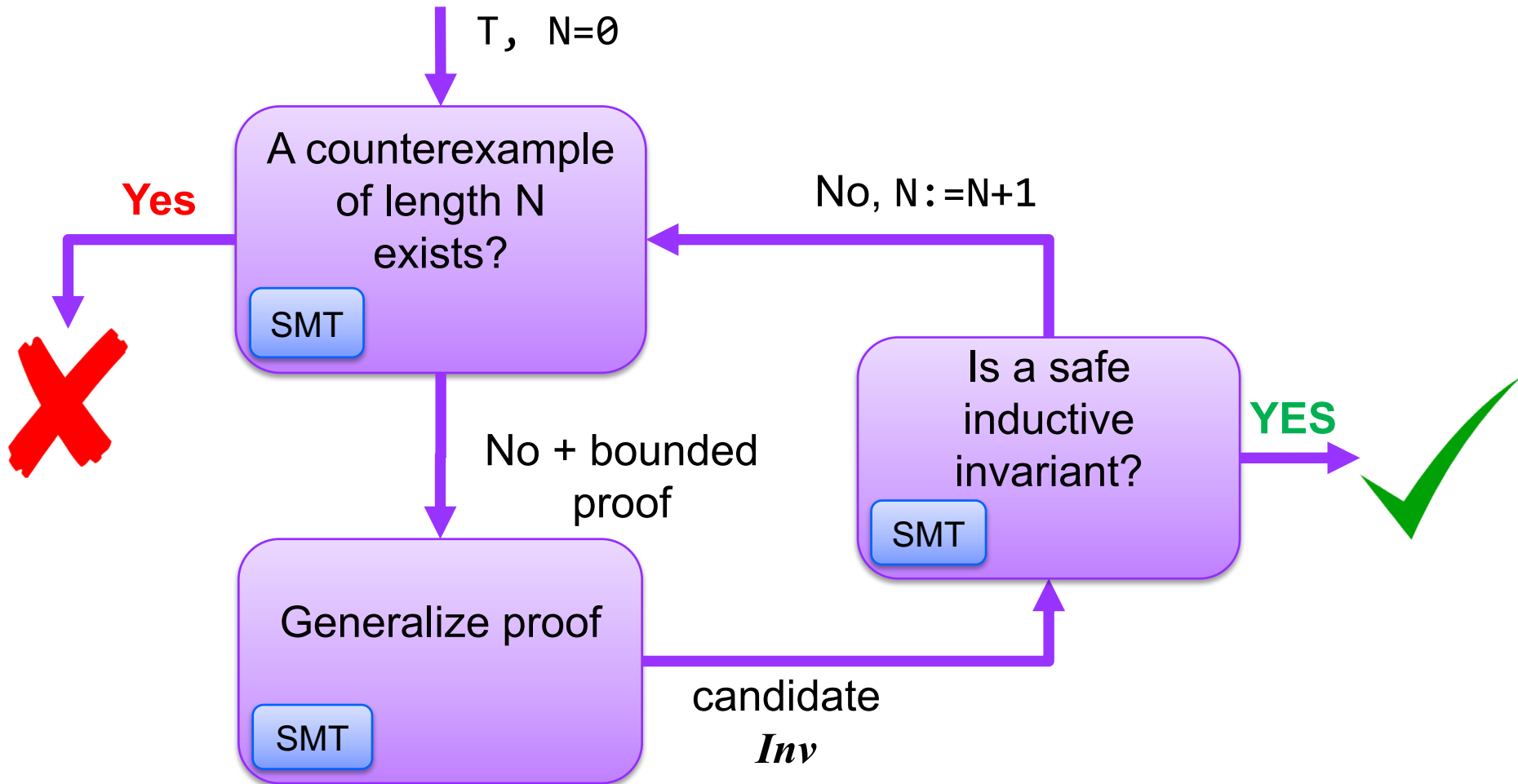
$$I_2(i, j, \bar{v}) \Rightarrow \neg Bad(i, j, \bar{v})$$

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

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

SMT-based Model Checking

Generalizing from bounded proofs



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 Arithmetic + Arrays

- Required to model heap manipulating programs
- A. Komuravelli, N. Bjørner, A. Gurfinkel, K. L. McMillan: Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays. FMCAD 2015

IC3, PDR, and Friends (3)

Quip: Forward Reachable States + Conjectures

- Use both forward and backward reachability information
- A. Gurfinkel and A. Ivrii: Pushing to the Top. FMCAD 2015

Avy: Interpolation with IC3

- Use SAT-solver for blocking, IC3 for pushing
- Y. Vizel, A. Gurfinkel: Interpolating Property Directed Reachability. CAV 2014

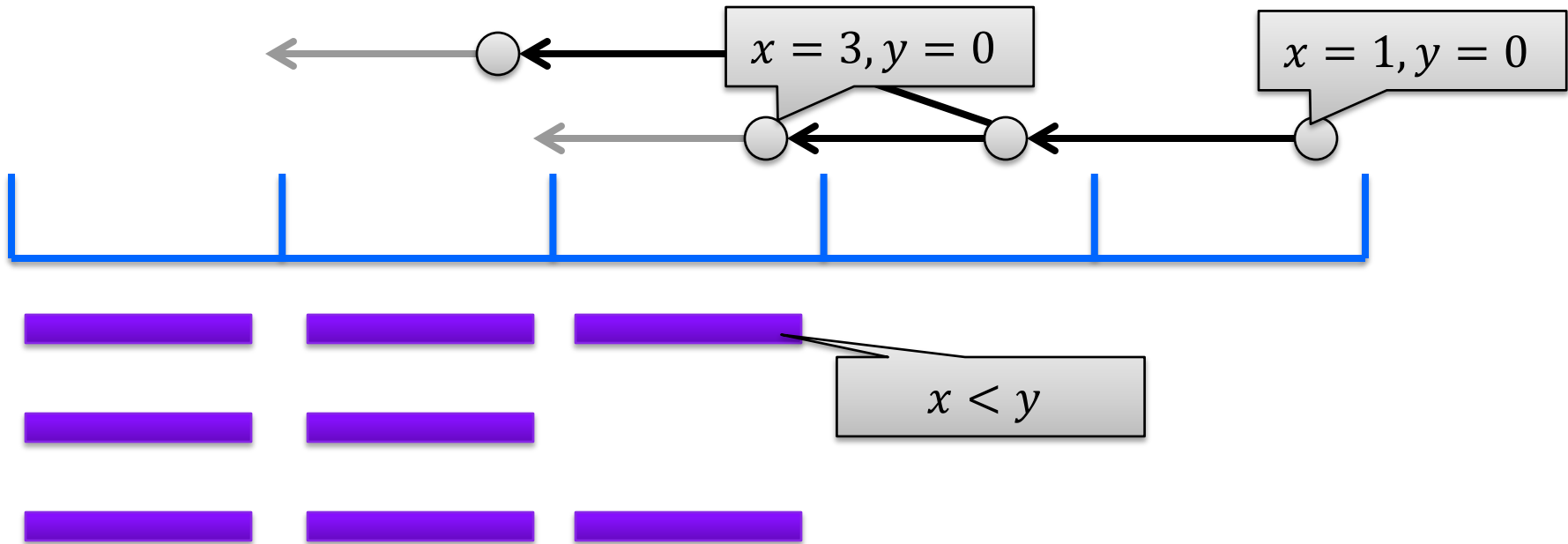
uPDR: Constraints in EPR fragment of FOL

- Universally quantified inductive invariants (or their absence)
- A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, S. Shoham: Property-Directed Inference of Universal Invariants or Proving Their Absence. CAV 2015

Quic3: Universally quantified invariants for LIA + Arrays

- Extending Spacer with quantified reasoning
- A. Gurfinkel, S. Shoham, Y. Vizel: Quantifiers on Demand. ATVA 2018

Spacer In Pictures: MkSafe



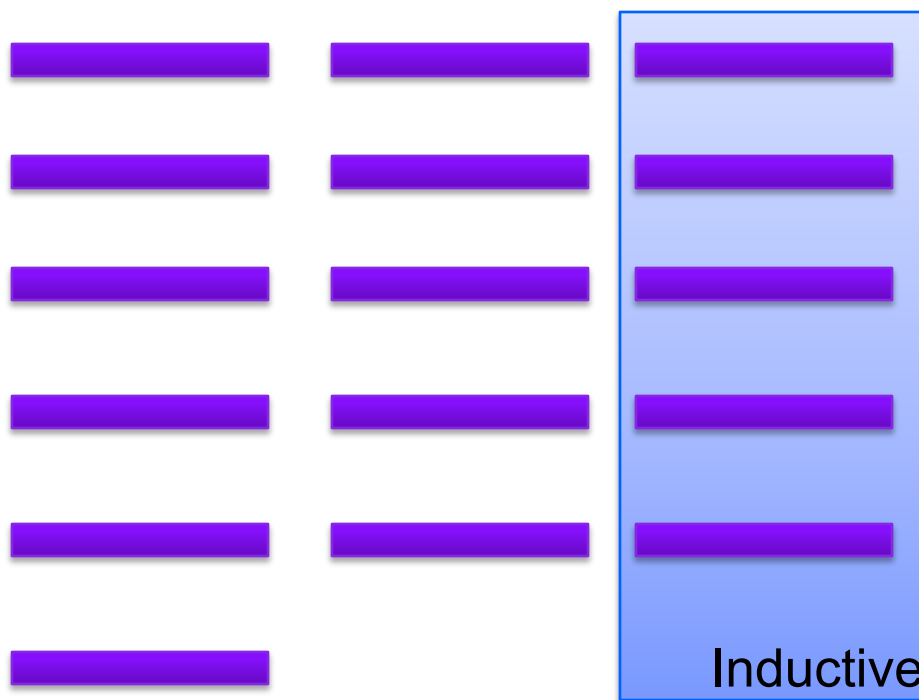
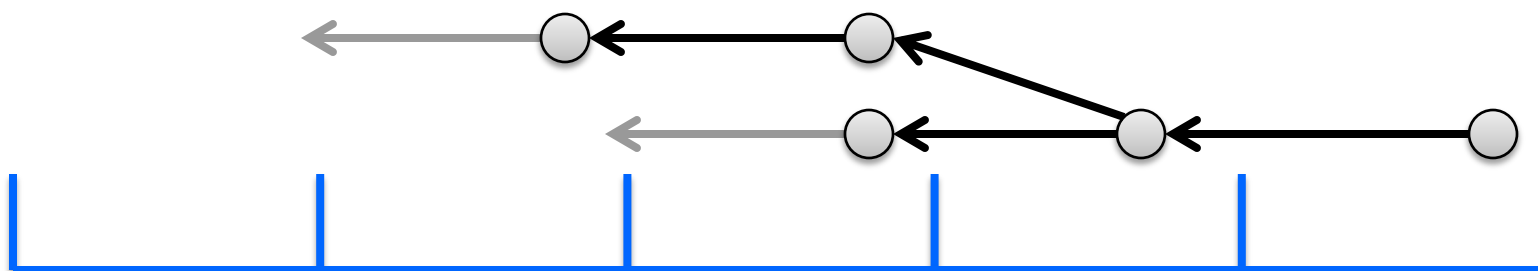
Predecessor

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

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

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

Spacer in Pictures: Push



Algorithm Invariants

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

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

Predecessor and NewLemma rules in Spacer

Predecessor – generate a new predecessor POB of a given POB m

- Use SMT to check satisfiability of a transition relation with given pre- and post-conditions
- Use Model-based Projection to construct new POB over pre-variables only

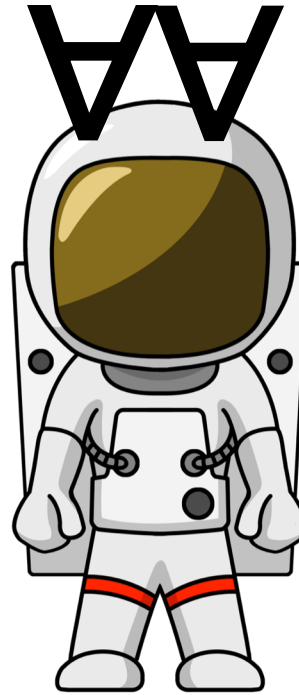
find M s.t. $M \models F_i \wedge Tr \wedge m'$

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

NewLemma – create a new lemma that blocks a given POB m

- Use SMT to check unsatisfiability of a transition relation with a given pre- and post-conditions
- Use Interpolation/UNSAT core/IUC to construct a new lemma

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



Extends Spacer with reasoning about quantified solutions

QUIC3: QUANTIFIED IC3

HORN(ALIA): Arrays + LIA

```
int A[N];  
for (int i = 0; i < N; ++i)  
    A[i] = 0;  
int j = nd();  
assume(0 <= j < N);  
assert(A[j] == 0);
```

IS SAT?



$\text{Inv}(A, N, 0)$

$\text{Inv}(A, N, i) \ \& \ i < N \rightarrow \text{Inv}(A[i := 0], N, i+1)$

$\text{Inv}(A, N, i) \ \& \ i \geq N \ \& \ 0 \leq j < N \ \& \ A[j] \neq 0 \rightarrow \text{false}$

In SMT-LIB

```
(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv ( (Array Int Int) Int Int ) Bool)

(assert
  (forall ( (A (Array Int Int)) (N Int) (C Int)) (Inv A N 0)))

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) )
    (=>
      (and (Inv A N i) (< i N) )
      (Inv (store A i 0) N (+ i 1))
    )
  )
)

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) (j Int))
    (=> (and (Inv A N i )
      (>= i N) (<= 0 j) (< j N) (not (= (select A
j) 0)))
      false
    )
  )
)

(check-sat)
(get-model)
```

```
$ z3 -t:100 array-zero.smt2
canceled
unknown
```

$\text{Inv}(A, N, i)$

$$\forall \ 0 \leq j < i < N \Rightarrow A[j] = 0$$

Predecessor in array-zero example

$\text{Inv}(A, N, i) \ \& \ i \geq N \ \& \ 0 \leq j < N \ \& \ A[j] \neq 0 \rightarrow \text{false}$

Tr: $i < N \ \& \ 0 \leq j < N \ \& \ A[j] \neq 0$

POB: true

$$\exists j \cdot i \geq N \wedge 0 \leq j < N \wedge A[j] \neq 0$$

$$= i \geq N \wedge \exists j \cdot (0 \leq j < N \wedge A[j] \neq 0)$$

$$= ???$$

No way to eliminate the existential quantifier!

- can use the value of j in the current model
- but this only works when $A[j]$ is not important

Quantified POBs and Lemmas

Must deal with existentially quantified POBs

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

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

Learning universally quantified lemmas is easy!

- if POB m is existentially quantified, then it's negation is universally quantified
- checking that Tr implies a universally quantified lemma is easy

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

But universal quantifiers make even basic SMT queries undecidable!

- cannot assume that SMT-solver will magically handle this for us

QUIC3: Quantified IC3

[kwik-ee]

Spacer extends IC3/PDR from Propositional logic to LIA + Arrays

Quic3 extends Spacer to discovering Universally Quantified solutions

- Extend proof obligations with free (implicitly existentially quantified) variables
- Allow universal quantifiers in lemmas
- Explicitly manage quantifier instantiations to guarantee progress
 - **without** syntactic restriction of formulas (e.g., MBQI, Local Theory, APF)
 - **without** user-specified patterns
- Quantified generalization to heuristically infer new quantifiers

Implemented in spacer in Z3 master branch

- `z3 fp.spacer.ground_pobs=false fp.spacer.q3.use_qgen=true`
`NAME.smt2`

QUIC3: Trace and Proof Obligations

A **quantified trace** $Q = Q_0, \dots, Q_N$ is a sequence of **frames**.

- A frame Q_i is a set of (ℓ, σ) , where ℓ is a **lemma** and σ a **substitution**.
- $qi(Q) = \{\ell\sigma \mid (\ell, \sigma) \in Q\}$ $\forall Q = \{\forall\ell \mid (\ell, \sigma) \in Q\}$
- Invariants:
 - **Bounded Safety**: $\forall i < N . \forall Q_i \rightarrow \neg \text{Bad}$
 - **Monotonicity**: $\forall i < N . \forall Q_{i+1} \subseteq \forall Q_i$
 - **Inductiveness**: $\forall i < N . \forall Q_i \wedge \text{Tr} \rightarrow \forall Q'_{i+1}$

A priority queue \mathcal{Q} of **quantified proof obligations (POBs)**

- $(m, \xi, i) \in \mathcal{Q}$ where m is a cube, ξ is a ground substitution for all free variables of m , and i is a numeric level
- if $(m, \xi, i) \in \mathcal{Q}$ then there exists a path of length $(N-i)$ from a state in $m\xi$ to a state in **Bad**



QUIC3: Rules

Input: A safety problem $\langle \text{Init}(X), \text{Tr}(X, X'), \text{Bad}(X) \rangle$.

Assumptions: Init , Tr and Bad are quantifier free.

Data: A POB queue \mathcal{Q} , where a POB $c \in \mathcal{Q}$ is a triple $\langle m, \sigma, i \rangle$, m is a conjunction of literals over X and free variables, σ is a substitution s.t. $m\sigma$ is ground, and $i \in \mathbb{N}$. A level N . A quantified trace $\mathcal{T} = Q_0, Q_1, \dots$, where for every pair $(\ell, \sigma) \in Q_i$, ℓ is a quantifier-free formula over X and free variables and σ a substitution s.t. $\ell\sigma$ is ground.

Notation: $\mathcal{F}(A) = (A(X) \wedge \text{Tr}(X, X')) \vee \text{Init}(X')$; $qi(Q) = \{\ell\sigma \mid (\ell, \sigma) \in Q\}$; $\forall Q = \{\forall \ell \mid (\ell, \sigma) \in Q\}$.

Output: *Safe* or *Cex*

Initially: $\mathcal{Q} = \emptyset$, $N = 0$, $Q_0 = \{(\text{Init}, \emptyset)\}$, $\forall i > 0 \cdot Q_i = \emptyset$.

repeat

Safe If there is an $i < N$ s.t. $\forall Q_i \subseteq \forall Q_{i+1}$ **return** *Safe*.

Cex If there is an m, σ s.t. $\langle m, \sigma, 0 \rangle \in \mathcal{Q}$ **return** *Cex*.

Unfold If $qi(Q_N) \rightarrow \neg \text{Bad}$, then set $N \leftarrow N + 1$.

Candidate If for some m , $m \rightarrow qi(Q_N) \wedge \text{Bad}$, then add $\langle m, \emptyset, N \rangle$ to \mathcal{Q} .

Predecessor If $\langle m, \xi, i + 1 \rangle \in \mathcal{Q}$ and there is a model M s.t.

$M \models qi(Q_i) \wedge \text{Tr} \wedge (m'_{sk})$, add $\langle \psi, \sigma, i \rangle$ to \mathcal{Q} , where $(\psi, \sigma) = \text{abs}(U, \varphi)$ and $(\varphi, U) = \text{PMBP}(X' \cup SK, \text{Tr} \wedge m'_{sk}, M)$.

NewLemma For $0 \leq i < N$, given a POB $\langle m, \sigma, i + 1 \rangle \in \mathcal{Q}$ s.t. $\mathcal{F}(qi(Q_i)) \wedge m'_{sk}$ is unsatisfiable, and $L' = \text{ITP}(\mathcal{F}(qi(Q_i)), m'_{sk})$, add (ℓ, σ) to Q_j for $j \leq i + 1$, where $(\ell, _) = \text{abs}(SK, L)$.

Push For $0 \leq i < N$ and $((\varphi \vee \psi), \sigma) \in Q_i$, if $(\varphi, \sigma) \notin Q_{i+1}$, $\text{Init} \rightarrow \forall \varphi$ and $(\forall \varphi) \wedge \forall Q_i \wedge qi(Q_i) \wedge \text{Tr} \rightarrow \forall \varphi'$, then add (φ, σ) to Q_j , for all $j \leq i + 1$.

until ∞ ;

QUIC3: Predecessor, NewLemma, and Push

repeat

⋮

Predecessor If $\langle m, \xi, i+1 \rangle \in \mathcal{Q}$ and there is a model M s.t.

$M \models qi(Q_i) \wedge Tr \wedge (m'_{sk})$, add $\langle \psi, \sigma, i \rangle$ to \mathcal{Q} , where $(\psi, \sigma) = abs(U, \varphi)$ and $(\varphi, U) = PMBP(X' \cup SK, Tr \wedge m'_{sk}, M)$.

NewLemma For $0 \leq i < N$, given a POB $\langle m, \sigma, i+1 \rangle \in \mathcal{Q}$ s.t. $qi(Q_i) \wedge Tr \wedge m'_{sk}$ is unsatisfiable, and $L' = ITP(\mathcal{F}(qi(Q_i)), m'_{sk})$, add (ℓ, σ) to Q_j for $j \leq i+1$, where $(\ell, -) = abs(SK, L)$.

Push For $0 \leq i < N$ and $((\varphi \vee \psi), \sigma) \in Q_i$, if $(\varphi, \sigma) \notin Q_{i+1}$, $Init \rightarrow \forall \varphi$ and $(\forall \varphi) \wedge \forall Q_i \wedge qi(Q_i) \wedge Tr \rightarrow \forall \varphi'$, then add (φ, σ) to Q_j , for all $j \leq i+1$.

until ∞ ;

In **Predecessor** and **NewLemma** only use current instantiations of quantified lemmas. All SMT queries are quantifier free

In **Push**, quantified lemmas are required for relative completeness

- in practice, we use incomplete pattern-based instantiation and hope that it is sufficient together with $qi(Q_i)$

Progress and Counterexamples

The **Predecessor** rule is only finitely applicable to any POB

- follows from how quantified terms are abstracted by free variables and how quantified lemmas are instantiated
- assumes that Skolemization is deterministic
- uses finiteness of Model Based Projection

MkSafe in Quic3 is terminating for any given bound N

- W.l.o.g, assume Bad is a single POB
- Follows by induction on the bound N

MkSafe in Quic3 computes a quantified interpolation sequence

If there is a counterexample, Quic3 will terminate with the shortest counterexample

In SMT-LIB

```
(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv ( (Array Int Int) Int Int ) Bool)

(assert
  (forall ( (A (Array Int Int)) (N Int) (C Int)) (Inv A N 0)))

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) )
    (=>
      (and (Inv A N i) (< i N) )
      (Inv (store A i 0) N (+ i 1))
    )
  )

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) (j Int))
    (=> (and (Inv A N i )
      (>= i N) (<= 0 j) (< j N) (not (= (select A
j) 0)))
      false
    )
  )

(check-sat)
(get-model)
```

```
$ z3 array-zero.smt2
```

```
sat
```

```
(model
  (define-fun Inv ((x!0 (Array Int Int)) (x!1 Int) (x!2 Int)) Bool
    (let ((a!1 (forall ((sk!0 Int))
      (! (or (not (>= sk!0 0))
        (>= (select x!0 sk!0) 0)
        (<= (+ x!2 (* (- 1) sk!0)) 0))
      :weight 15)))
      (a!2 (forall ((sk!0 Int))
        (! (or (not (>= sk!0 0))
          (<= (select x!0 sk!0) 0)
          (<= (+ x!2 (* (- 1) sk!0)) 0))
          :weight 15))))
    (and a!1 a!2)))
)
```

almost ...
THE END

HORN(ALIA): Arrays + LIA

```
int A[N];  
for (int i = 0; i < N; ++i)  
    A[i] = 0;  
for (i = 0; i < N; ++i)  
    assert(A[i] == 0);
```

IS SAT?



Inv1(A, N, 0)

Inv1(A, N, i) & i < N \rightarrow Inv1(A[i := 0], N, i+1)

Inv1(A, N, i) & i \geq N \rightarrow Inv2(A, N, 0)

Inv2(A, N, i) & i < N & A[i] = 0 \rightarrow Inv2(A, N, i+1)

Inv2(A, N, i) & i < N & A[i] \neq 0 \rightarrow false

In SMT-LIB

```
(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv1 ( (Array Int Int) Int Int ) Bool)
(declare-fun Inv2 ( (Array Int Int) Int Int ) Bool)

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int)) (Inv1 A N 0)))

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) )
    (=>
      (and (Inv1 A N i) (< i N) )
      (Inv1 (store A i 0) N (+ i 1))
    )
  )
)

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) )
    (=>
      (and (Inv1 A N i) (>= i N) ) (Inv2 A N 0)
    )
  )
)

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) )
    (=>
      (and (Inv2 A N i) (< i N) (= (select A i) 0) ) (Inv2 A N (+ i 1))
    )
  )
)

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) )
    (=>
      (and (Inv2 A N i) (< i N) (not (= (select A i) 0)) ) false
    )
  )
)

(check-sat)
(get-model)
```

```
$ z3 -t:100 array-zero2.smt2
canceled
unknown
```

Why this example diverges?

$\text{Inv2}(A, N, i) \ \& \ i < N \ \& \ A[i] \neq 0 \rightarrow \text{false}$

$i < N \wedge A[i] \neq 0 \longleftarrow \text{true}$

$\text{Inv1}(A, N, i) \ \& \ i \geq N \rightarrow \text{Inv2}(A, N, 0)$

$0 < N \leq i \wedge A[0] \neq 0 \longleftarrow i < N \wedge A[i] \neq 0$

$\text{Inv2}(A, N, i) \ \& \ i < N \ \& \ A[i] = 0 \rightarrow \text{Inv2}(A, N, i+1)$

$i + 1 < N \wedge A[i] = 0 \wedge A[i + 1] \neq 0 \longleftarrow i < N \wedge A[i] \neq 0$

$\text{Inv1}(A, N, i) \ \& \ i \geq N \rightarrow \text{Inv2}(A, N, 0)$

$1 < N \leq i \wedge A[0] = 0 \wedge A[1] \neq 0 \longleftarrow i + 1 < N \wedge A[i] = 0 \wedge A[i + 1] \neq 0$

Quantified Generalizer

“... to boldly go where no one has gone before” (but many have been)

$$1 < N \leq i \wedge A[0] = 0 \wedge A[1] \neq 0$$

Quantified generalizer is a heuristic to generalize POBs using existential quantifiers

- e.g., in our example, we want to generalize the pob into

$$\exists j \cdot 1 < N \leq i \wedge 0 \leq j < N \wedge A[j] \neq 0$$

We look for a pattern in the formula (anti-unification)

Use convex closure (i.e., abstract join) to capture the pattern by a conjunction

Apply **after** pob is blocked and generalized

- As any generalization, it is a *dark magic*

In SMT-LIB

```
(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv1 ( (Array Int Int) Int Int ) Bool)
(declare-fun Inv2 ( (Array Int Int) Int Int ) Bool)

(assert
  (forall ( (A (Array Int Int)) (N Int) (C Int)) (Inv1 A N 0)))

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) )
    (=>
      (and (Inv1 A N i) (< i N) )
      (Inv1 (store A i 0) N (+ i 1))
    )
  )
)
(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) )
    (=>
      (and (Inv1 A N i) (>= i N) ) (Inv2 A N 0)
    )
  )
)

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) )
    (=>
      (and (Inv2 A N i) (< i N) (= (select A i 0) ) (Inv2 A N (+ i 1))
    )
  )
)

(assert
  (forall ( (A (Array Int Int)) (N Int) (i Int) )
    (=>
      (and (Inv2 A N i) (< i N) (not (= (select A i 0) ) ) false
    )
  )
)

(check-sat)
(get-model)
```

\$ z3 array-zero2.smt2

sat

```
(model
  (define-fun Inv2 ((x!0 (Array Int Int)) (x!1 Int) (x!2 Int)) Bool
    (let ((a!1 (forall ((sk!0 Int))
      (! (or (<= (+ x!1 (* (- 1) sk!0)) 0)
        (<= (select x!0 sk!0) 0)
        (<= (+ sk!0 (* (- 1) x!2)) 0))
      :weight 15))))
      (a!2 (or (<= (+ x!1 (* (- 1) x!2)) 0) (<= (select x!0 x!2) 0)))
      (a!3 (or (>= (select x!0 x!2) 0) (<= (+ x!1 (* (- 1) x!2)) 0)))
      (a!4 (forall ((sk!0 Int))
        (! (or (<= (+ x!1 (* (- 1) sk!0)) 0)
          (>= (select x!0 sk!0) 0)
          (<= (+ sk!0 (* (- 1) x!2)) 0))
        :weight 15))))
      (and a!1 a!2 a!3 a!4)))
  (define-fun Inv1 ((x!0 (Array Int Int)) (x!1 Int) (x!2 Int)) Bool
    (let ((a!1 (forall ((sk!0 Int))
      (! (or (<= (select x!0 sk!0) 0)
        (<= (+ x!2 (* (- 1) sk!0)) 0)
        (<= sk!0 0))
      :weight 15))))
      (a!2 (forall ((sk!0 Int))
        (! (let ((a!1 (>= (+ sk!0 (* (- 1) (select x!0 sk!0))) 0)))
          (or (not (>= sk!0 0)) (<= (+ x!2 (* (- 1) sk!0)) 0) a!1))
        :weight 15))))
      (a!3 (forall ((sk!0 Int))
        (! (or (<= (+ x!2 (* (- 1) sk!0)) 0)
          (>= (select x!0 sk!0) 0)
          (<= sk!0 0))
        :weight 15))))
      (and a!1 a!2 (or (>= (select x!0 0) 0) (<= x!2 0)) a!3)))
  )
)
```

DEMO

Related Work

Predicate Abstraction

- extend predicates with fresh universally quantified variables
- relies on a decision procedure for quantified logic

Model-Checking Modulo Theories (MCMT)

- model checking of array manipulating programs
- supported by multiple tools: cubicle, mcmt, safari, ...
- quantifier elimination to compute predecessors
- requires checking satisfiability of quantified formulas for sub-sumption and convergence

Discovery of Universal Invariants with Abstract Interpretation

- compute universally quantified inductive invariants of a certain shape
- often specialized for reasoning about arrays in programming languages
- not property directed, no guarantees, but often very quick
- can be combined with Quic3 as pre-processing

Most Closely Related Work

Safari and Booster

- extends Lazy Abstraction with Interpolants (LAWI) to array manipulating programs
- solves mkSafe() using quantifier free theory of arrays and computes **quantifier free** sequence interpolant
- heuristically guesses quantified lemmas by abstracting terms
- see Avy for in-depth comparison between interpolation and IC3

Transformation into non-linear CHC

- guess number of quantifiers and instances statically
- use quantifier-free **non-linear** CHC solver to find template invariant
- generalizes most Abstract Interpretation / Template-based approaches
- cannot discover counterexamples
- can be simulated in Quic3 by restricting instantiations used

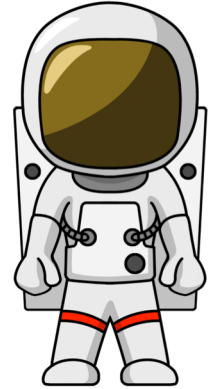
UPDR

- existential pobs and universal lemmas over decidable theories

Conclusion

Quic3 brings reasoning about quantified invariants to CHC

- Implemented in spacer
- can discover non-trivial quantified invariants of complex code



Guarantee progress and counterexamples

- don't get stuck with a quantified SMT query
- find shortest counterexample

Many open questions remain

- strides – memory is traversed in a stride (e.g., $x=x+4$)
- additional quantified generalizers (speed vs precision)

Enumerating invariants in a decidable fragment (EssenUF, APF, etc.)

Full paper to appear in ATVA 2018

CHC-COMP: CHC Solving Competition

First edition on July 13, 2018 at HVCS@FLOC

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

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

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

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

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



?

?

?



?

?

?



IC3/PDR: Solving Linear (Propositional) CHC

Unreachable and Reachable

- terminate the algorithm when a solution is found

Unfold

- increase search bound by 1

Candidate

- choose a bad state in the last frame

Decide

- extend a cex (backward) consistent with the current frame
- choose an assignment \mathbf{s} s.t. $(\mathbf{s} \wedge F_i \wedge \text{Tr} \wedge \text{cex}')$ is SAT

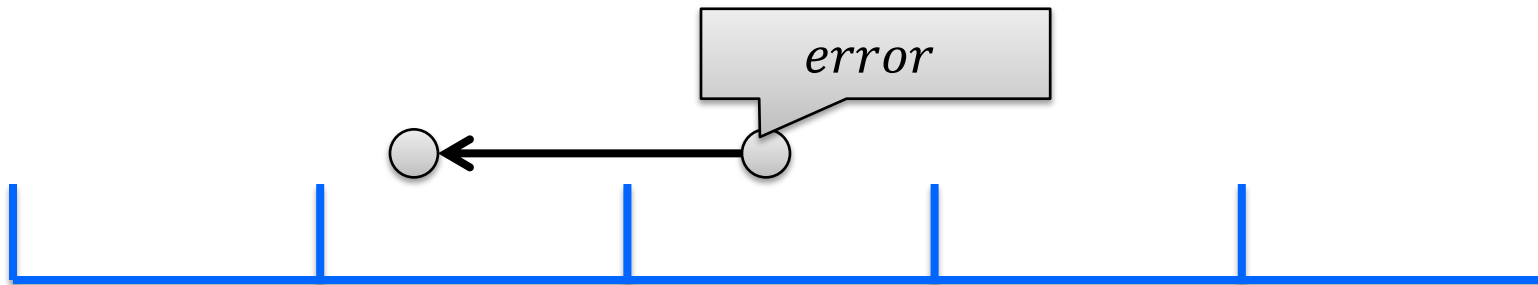
Conflict

- construct a lemma to explain why cex cannot be extended
- Find a clause L s.t. $L \Rightarrow \neg \text{cex}$, $\text{Init} \Rightarrow L$, and $L \wedge F_i \wedge \text{Tr} \Rightarrow L'$

Induction

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

Lemma Generation Example



Transition Relation

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

Pob

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

Farkas explanation for unsat

$$\begin{array}{c}
 \frac{x_0 + y_0 \leq z_0, \quad x \leq x_0, \quad z_0 < z, \quad i \leq i_0 + 1}{x + i \leq z} \qquad \frac{i \geq y, \quad x + y > z}{x + i > z} \\
 \hline
 \text{false}
 \end{array}$$

Learn lemma:

$$x + i \leq z$$