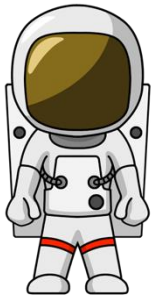
Constrained Horn Clauses for Program Verification and Synthesis

Prof. Arie Gurfinkel
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

July 25, 2026
CI-BD-SOQE Workshop @ FLOC

joint work with **A. Komuravelli**, S. Chaki, G. Fedyukovich, S. Shoham, N. Bjørner, **Hari Govind V. K.**, Y. (Jeff) Chen, I. Garcia-Contreras, **S. Wesley**, J.A. Navas, V. Wüstholz, M. Christakis, R. Trefler

Software Model Checking of
Programs / Transitions Systems /
Push-down Systems



=

Satisfiability of Constrained
Horn Logic (CHC) fragment of
First Order Logic

Reduce Model Checking to
FOL Satisfiability

Constrained Horn Clauses (CHC)

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

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

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

CHC Satisfiability

Π - set of CHCs

M - $\mathcal{T}$-**model** of a set of Π

- M satisfies $\mathcal{T}$
- M satisfies Π – through first-order interpretation of each predicate pi

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

- this is the usual FOL satisfiability for a set of FOL formulas

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

For program verification, the mapping is:

Program $\models \varphi$	**iff**	$CHC_{Program} \rightarrow \varphi$
Inductive Invariant	=	Solution to CHC
Counter Example Trace	=	Resolution proof of CHC

Example CHC: Is this SAT?

$$\forall x \cdot x \leq 0 \implies P(x)$$
$$\forall x, x' \cdot P(x) \wedge x < 5 \wedge x' = x + 1 \implies P(x')$$
$$\forall x \cdot P(x) \wedge x \geq 10 \implies \mathit{false}$$

Yes! This set of clauses is satisfiable
The **model** is an extension of the standard model of arithmetic with:

$$P(x) \equiv \{x \mid x \leq 5\}$$
$$\equiv \{5, 4, 3, 2, \ldots\}$$

Note that P(x) is definable by LIA predicate x <= 5

Validating the solution

Original CHC

$$\forall x \cdot x \leq 0 \implies P(x)$$
$$\forall x, x' \cdot P(x) \wedge x < 5 \wedge x' = x + 1 \implies P(x')$$
$$\forall x \cdot P(x) \wedge x \geq 10 \implies \mathit{false}$$

Validation of P(x) = {x | x <= 5}

$$\vdash \forall x \cdot x \leq 0 \implies x \leq 5$$
$$\vdash \forall x, x' \cdot x \leq 5 \wedge x < 5 \wedge x' = x + 1 \implies x' \leq 5$$
$$\vdash \forall x \cdot x \leq 5 \wedge x \geq 10 \implies \mathit{false}$$

Example CHC: is this SAT?

$$\forall x \cdot x \leq 0 \implies Q(x)$$
$$\forall x, x' \cdot Q(x) \wedge x < 5 \wedge x' = x + 1 \implies Q(x')$$
$$\forall x \cdot Q(x) \wedge x \geq 2 \implies \mathit{false}$$

No! This set of clauses is unsatisfiable

Justification is a refutation by **resolution** and **instantiation**

Example CHC: is this SAT?

$$\forall x \cdot x \leq 0 \implies Q(x)$$
$$\forall x, x' \cdot Q(x) \wedge x < 5 \wedge x' = x + 1 \implies Q(x')$$
$$\forall x \cdot Q(x) \wedge x \geq 2 \implies \mathit{false}$$

Refutation

$$\dfrac{\dfrac{\dfrac{(x = 0)\ \dfrac{\forall x \cdot x \leq 0 \implies Q(x)}{Q(0)} \qquad \forall x \cdot Q(x) \wedge x < 5 \implies Q(x+1)}{Q(1)} \qquad \forall x \cdot Q(x) \wedge x < 5 \implies Q(x+1)}{Q(2)} \qquad \forall x \cdot Q(x) \wedge x \geq 2 \implies \mathit{false}}{\mathit{false}}$$

A Brief History of Modern CHC in MC

PLDI 2012 S. Grebenshchikov, N. P. Lopes, C. Popeea, A. Rybalchenko , "**Synthesizing software verifiers from proof rules**"
- Constrained Horn Clauses as input format for Software Model Checkers

SAT 2012 K. Hoder, N. Bjørner , "**Generalized Property Directed Reachability**"
- IC3/PDR for SMT == Solving CHCs

SMT 2012 N. Bjørner, K. L. McMillan, A. Rybalchenko, "**Program Verification as Satisfiability Modulo Theories**"
- CHC format extension for SMT-LIB

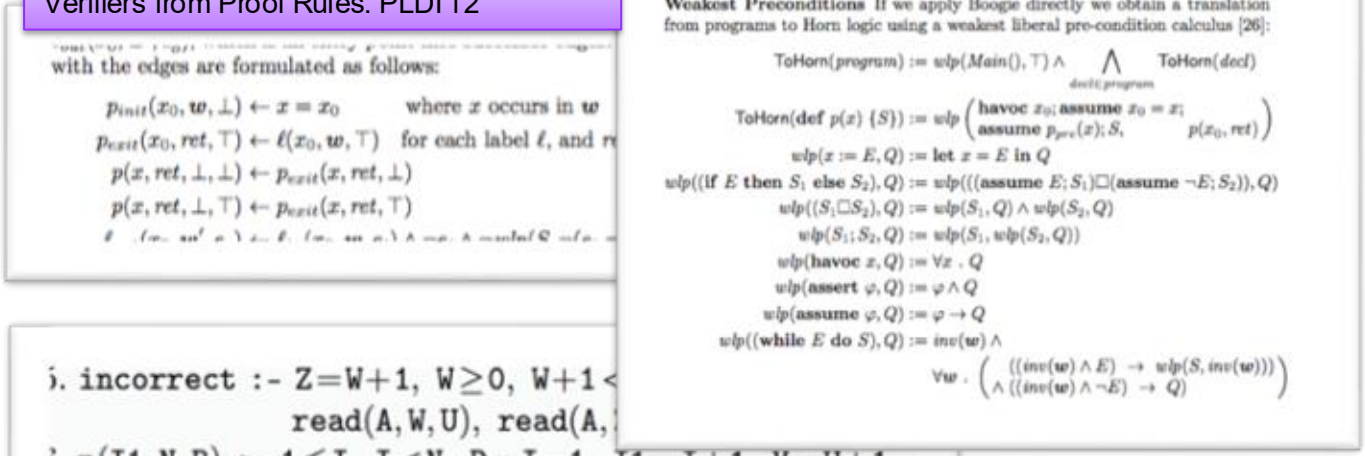
CAV 2014 A. Komuravelli, G., S. Chaki, "**SMT-Based Model Checking of Recursive Programs**"
- First version of SPACER as an extension of GPDR in Z3

CAV 2015 G, T. Kahsai, A. Komuravelli, J. Navas , "**The SeaHorn Verification Framework**"
- First robust and efficient automated verification tool based on CHC solving

2018 1st CHC-COMP, SPACER merged into Z3 master
- https://chc-comp.github.io/2018/

Horn Clauses for Program Verification

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



Weakest Preconditions If we apply Boogie directly we obtain a translation from programs to Horn logic using a weakest liberal pre-condition calculus [26]:

$$\text{ToHorn}(program) := wlp(Main(), \top) \wedge \bigwedge_{decl \in program} \text{ToHorn}(decl)$$

$$\text{ToHorn}(\textbf{def}\ p(x)\ \{S\}) := wlp\left(\begin{array}{l}\textbf{havoc}\ x_0; \textbf{assume}\ x_0 = x;\\ \textbf{assume}\ p_{pre}(x); S, \qquad p(x_0, ret)\end{array}\right)$$

$$wlp(x := E, Q) := \textbf{let}\ x = E\ \textbf{in}\ Q$$

$$wlp((\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2), Q) := wlp(((\textbf{assume}\ E; S_1) \Box (\textbf{assume}\ \neg E; S_2)), Q)$$

$$wlp((S_1 \Box S_2), Q) := wlp(S_1, Q) \wedge wlp(S_2, Q)$$

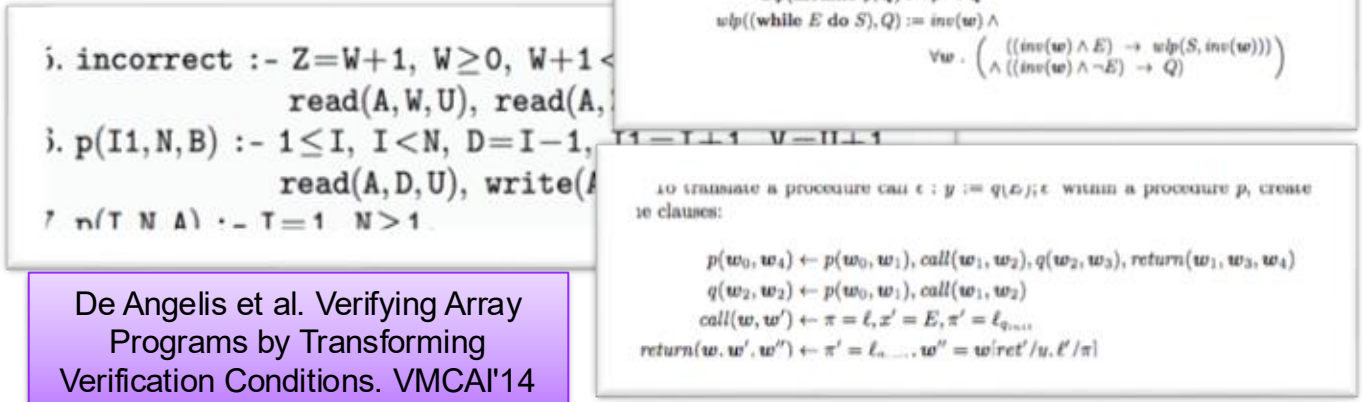
$$wlp(S_1; S_2, Q) := wlp(S_1, wlp(S_2, Q))$$

$$wlp(\textbf{havoc}\ x, Q) := \forall x\ .\ Q$$

$$wlp(\textbf{assert}\ \varphi, Q) := \varphi \wedge Q$$

$$wlp(\textbf{assume}\ \varphi, Q) := \varphi \rightarrow Q$$

$$wlp((\textbf{while}\ E\ \textbf{do}\ S), Q) := inv(\mathbf{w}) \wedge \forall \mathbf{w}\ .\ \left(\begin{array}{l}((inv(\mathbf{w}) \wedge E) \rightarrow wlp(S, inv(\mathbf{w})))\\ \wedge\ ((inv(\mathbf{w}) \wedge \neg E) \rightarrow Q)\end{array}\right)$$



```
). incorrect :- Z=W+1, W≥0, W+1<
                read(A,W,U), read(A,
). p(I1,N,B) :- 1≤I, I<N, D=I−1,
                read(A,D,U), write(A
```

To translate a procedure call $c : y := q(x_1, \ldots, x_n)$ within a procedure p, create the clauses:

$$p(\mathbf{w}_0, \mathbf{w}_4) \leftarrow p(\mathbf{w}_0, \mathbf{w}_1), call(\mathbf{w}_1, \mathbf{w}_2), q(\mathbf{w}_2, \mathbf{w}_3), return(\mathbf{w}_1, \mathbf{w}_3, \mathbf{w}_4)$$

$$q(\mathbf{w}_2, \mathbf{w}_2) \leftarrow p(\mathbf{w}_0, \mathbf{w}_1), call(\mathbf{w}_1, \mathbf{w}_2)$$

$$call(\mathbf{w}, \mathbf{w}') \leftarrow \pi = \ell, x' = E, \pi' = \ell_{q_{init}}$$

$$return(\mathbf{w}, \mathbf{w}', \mathbf{w}'') \leftarrow \pi' = \ell_{\ldots}, \mathbf{w}'' = \mathbf{w}[ret'/u, \ell'/\pi]$$

De Angelis et al. Verifying Array Programs by Transforming Verification Conditions. VMCAI'14

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

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

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

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

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

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

Figure 4: Horn constraints encoding a homogeneous infinite system with the help of a k-indexed invariant. S_k is the symmetric group on $\{1, \ldots, k\}$, 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

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

(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

$$
\begin{aligned}
& Init(i, j, \overline{v}) \wedge Init(j, i, \overline{v}) \wedge \\
& \quad Init(i, i, \overline{v}) \wedge Init(j, j, \overline{v}) \Rightarrow I_2(i, j, \overline{v}) \\
& \quad I_2(i, j, \overline{v}) \wedge Tr(i, \overline{v}, \overline{v}') \Rightarrow I_2(i, j, \overline{v}') \quad (3) \\
& \quad I_2(i, j, \overline{v}) \wedge Tr(j, \overline{v}, \overline{v}') \Rightarrow I_2(i, j, \overline{v}') \quad (4) \\
& I_2(i, j, \overline{v}) \wedge I_2(i, k, \overline{v}) \wedge I_2(j, k, \overline{v}) \wedge \\
& \quad Tr(k, \overline{v}, \overline{v}') \wedge k \neq i \wedge k \neq j \Rightarrow I_2(i, j, \overline{v}') \quad (5) \\
& \quad I_2(i, j, \overline{v}) \Rightarrow \neg Bad(i, j, \overline{v})
\end{aligned}
$$

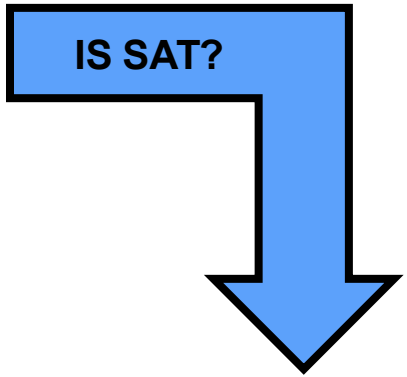
Figure 3: $VC_2(T)$ **for two-quantifier invariants.**

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

UNIVERSITY OF WATERLOO

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                          ➔ Inv(x, y, z, i)
Inv(x, y, z, i) & i < y & z1=z+1 & i1=i+1      ➔ Inv(x, y, z1, i1)
Inv(x, y, z, i) & i >= y & z != x+y            ➔ 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)))
)
```

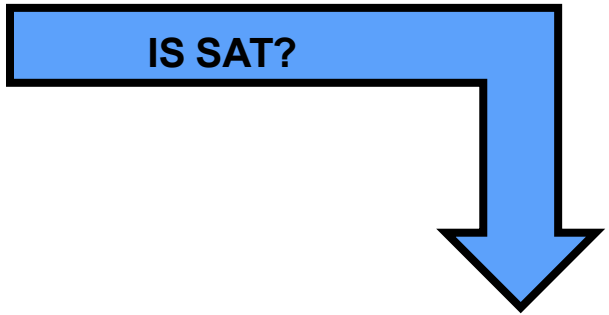
$$Inv(x, y, z, i)$$

$$z = x + i$$

$$z <= x + y$$

Program Verification with HORN(LIA)

```
int inc(int z) { return z + 1; }
assume(x <= 0);
while (x < 5) {
  x = inc(x);
}
assert(x < 10);
```




```
r = z + 1                          ➔   Inc(z, r)
x <= 0                             ➔   Inv(x)
Inv(x) & x < 5 & Inc(x, y)         ➔     Inv(y)
Inv(x) & x >= 5 & x >= 10          ➔     false
```

In SMT-LIB

```
(set-logic HORN)
(set-option :fp.xform.inline_linear false)
(set-option :fp.xform.inline_eager false)
(declare-fun Inv ( Int ) Bool)
(declare-fun Inc ( Int Int ) Bool)

(assert (forall ((z Int)) (Inc z (+ z 1))))

(assert (forall ((x Int)) (=> (<= x 0) (Inv x))))

(assert (forall ((x Int) (y Int)) (=> (and (< x 5) (Inc x y))
(Inv y))))

(assert (forall ((x Int)) (=> (and (Inv x) (>= x 5) (>= x 10))
false)))

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

```
$ z3 add-by-one-fn.smt2
sat
(
  (define-fun Inc ((x!0 Int) (x!1 Int)) Bool
    (not (>= (+ x!1 (* (- 1) x!0)) 2)))
  (define-fun Inv ((x!0 Int)) Bool
    (not (>= x!0 6)))
)
```

$$Inc(x0,x1) := x1 <= x0 + 1$$

$$Inv(x0) := x0 <= 5$$

Applications of CHCs

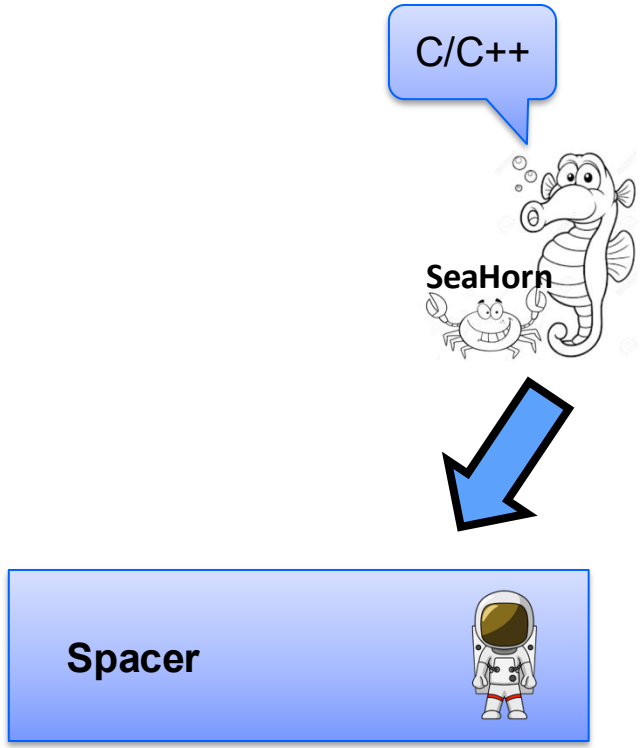
Prototyping different strategies and proof rules for verification

- verification by inductive invariants
- modular invariants
- predicate abstraction
- modular proof rules for concurrent systems
- verification of parameterized systems
- type inference for refinement type systems
- synthesis
- …
- create new verification tools by reducing to CHCs

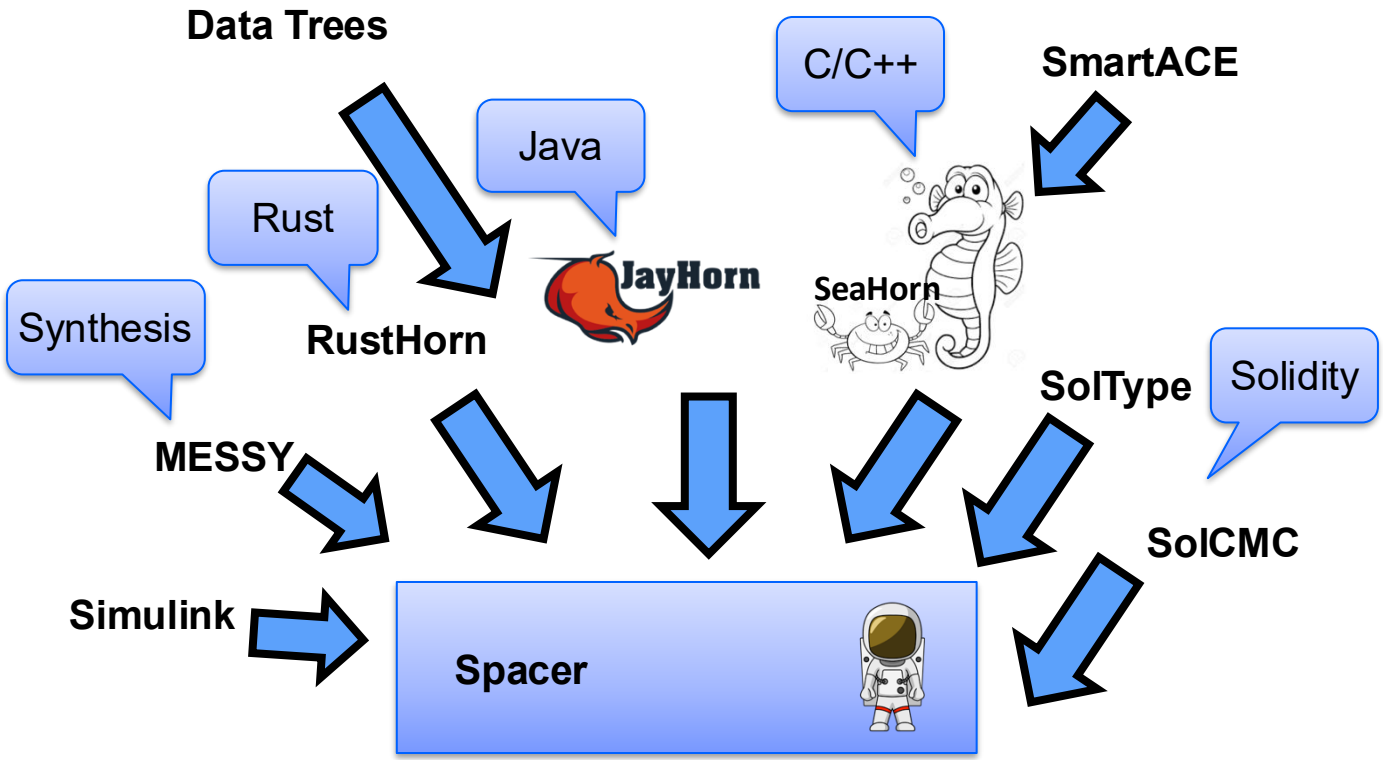
Building automated verification tools

- SeaHorn, JayHorn, RustHorn, …
- SmartACE, SolCMC, …

Logic-based Algorithmic Verification



Logic-based Algorithmic Verification



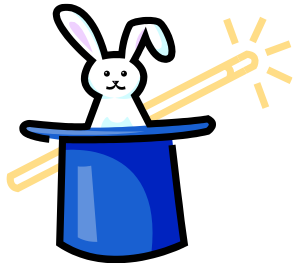
Current State of CHC Solving

Multiple mature solvers using competing techniques and algorithms
- Spacer (in Z3), Eldarica, FreqHorn, Golem, …

Annual competition
- CHC-COMP: https://chc-comp.github.io/
- in 2022, 7 tracks with 5+1 solvers
- In 2024, 6 tracks with 7 solvers (no Spacer this year)

Growing collection of benchmarks
- maintained by CHC-COMP
- established (simplified) format
- organized in separate repos under https://github.com/chc-comp

Growing number of academic and industrial users
- SeaHorn, JayHorn, RustHorn, MESSY, SolType, SolC SMTChecker, …



SOLVING CONSTRAINED HORN CLAUSES

A little bit of complexity

Satisfiability of CHC over most interesting theories is **undecidable**
- e.g., CHC(Linear Real Arithmetic), CHC(Linear Integer Arithmetic)
- proof: many easy reductions, for example, counter automata

Satisfiability of Linear CHC over Propositional logic is **decidable**
- **Finite state model checking** of transition systems
- Complexity: linear in the size of the graph induced by the transition system

Satisfiability of Non-Linear CHC over Propositional logic is **decidable**
- **Finite state** model checking of **pushdown systems**
- Complexity: cubic in the size of the pushdown system

Decidability of some classes of CHC: Difference arithmetic (= timed automata)

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

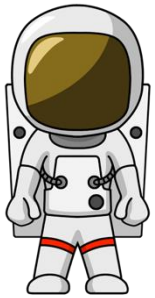
Machine Learning
- PLDI'18: sample, ML to guess predicates, DT to guess combinations

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 (extending IC3/PDR to SMT)
- SPACER, Implicit Predicate Abstraction

Spacer: Solving SMT-constrained CHC



Spacer: SAT procedure for SMT-constrained Horn Clauses

- now the default CHC solver in Z3
  - https://github.com/Z3Prover/z3

Supported SMT-Theories

- Linear Real and Integer Arithmetic
- Quantifier-free theory of arrays
- Universally quantified theory of arrays + arithmetic
- Good support for many other SMT-theories
  - bit-vectors, ADT, recursive functions, …

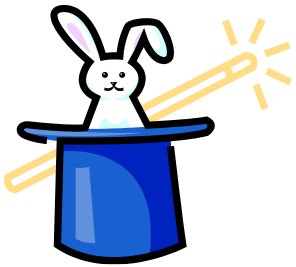
Supports Non-Linear CHC

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

A Magician's Guide to Solving Undecidable Problems

Develop a procedure ***P*** for a decidable problem

Show that ***P*** is a decision procedure for the problem
- e.g., model checking of finite-state systems



Choose one of
- Always terminate with some answer (over-approximation)
- Always make useful progress (under-approximation)

Extend procedure ***P*** to procedure ***Q*** that "solves" the undecidable problem
- Ensure that ***Q*** is still a decision procedure whenever ***P*** is
- Ensure that ***Q*** either always terminates or makes progress

SPACER's guiding principles for solving CHCs

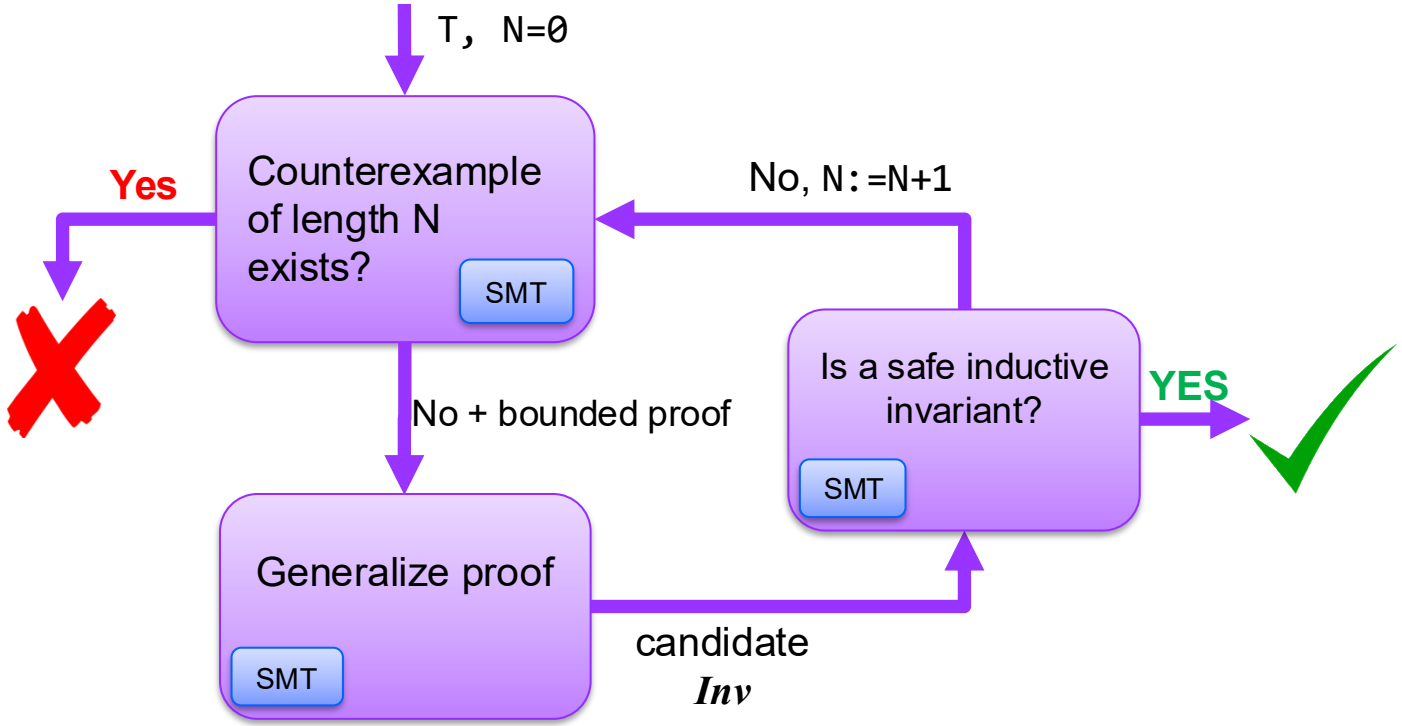
Make Progress

- always make progress
- if input CHC is unsatisfiable, after enough time, the solving procedure must terminate with UNSAT
- e.g., examine longer and longer resolution proofs (i.e., unfoldings)

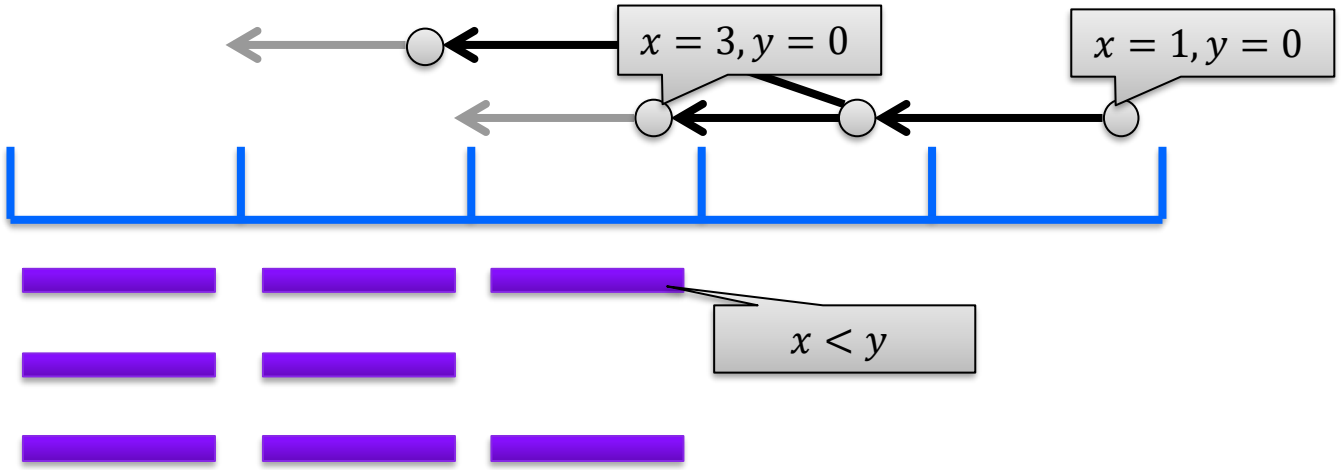
Keep Decidability

- decision procedure for decidable fragments
- usually, we ensure that solving procedures are decision procedures for CHC over Propositional logic (i.e., finite state model checking)
- "sharpen" decidability result based on specific domain (i.e., LIA, ADT, etc.)
- many open decidability questions remain
  - e.g., is Spacer a decision procedure for (encoding) of timed automata?

Verification by Incremental Generalization



IC3/PDR In Pictures: Search for Finite Cexs



Predecessor

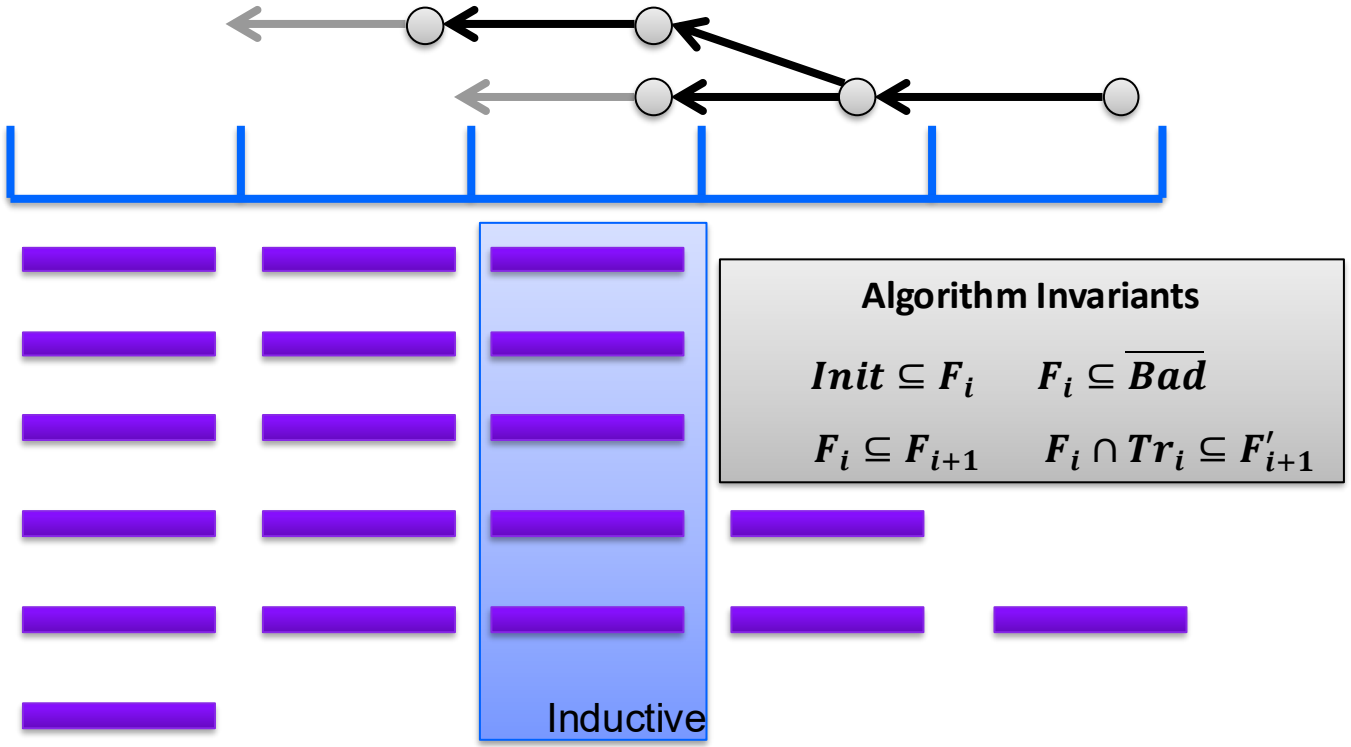
$$\text{find } M \text{ s.t. } M \models F_i \wedge Tr \wedge m'$$

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

NewLemma

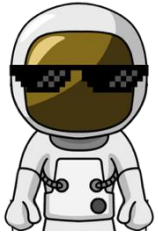
$$\text{find } \ell \text{ s.t. } (F_i \wedge Tr \implies \ell') \wedge (\ell \implies \neg m)$$

IC3/PDR in Pictures: Is Inductive



SMT-query: $\vdash \ell \wedge F_i \wedge Tr \implies \ell'$

The Curse of Interpolation



Hari Govind V. K., J. Chen, S. Shoham, G.; Global Guidance for Local Generalization in Model Checking. CAV 2020

The Curse of Interpolation

Interpolation is capable of generating many interesting terms

- (almost) any inductive invariant is an interpolant of something under the right conditions!

Interpolation often works in practice

- creates false sense of security
- predicate / term generation is a solved problem

However, interpolation is very hard to control!

- Small changes to input result in big change in interpolants
- Small changes to solver parameter result in big change in interpolants
- Works well overall (i.e., large benchmark set), but poorly for any given user problem!

Spacer Tom ONLY knows how to do

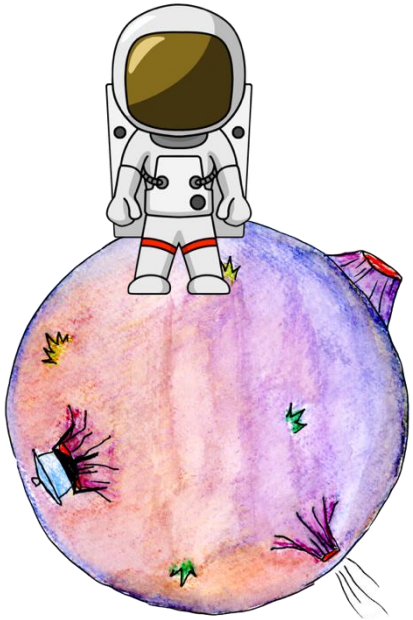
Local **reasoning**

Generalizing from single predecessors results in **limited** exploration **horizon**

Generalization typically relies on **interpolation**

Interpolation can work wonders!

e.g., generate breakthrough terms like equality: a = b



Ground Control to Spacer Tom:

We've got a PROBLEM!

Interpolation heuristics are not aware of the overall structure of the inductive proof so far

Interpolant computation is very much dependent on the heuristics in the underlying SMT:

$a + b < 4$ is just as likely as $a = b$

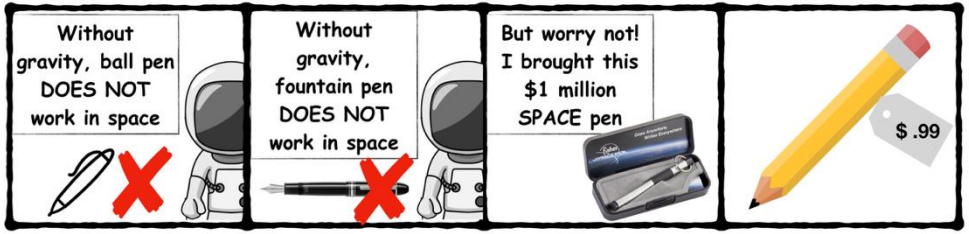
The problem is more pronounced in infinite-state systems since there are infinitely many generalization possible

Spacer Tom can be **MISSGUIDED!**

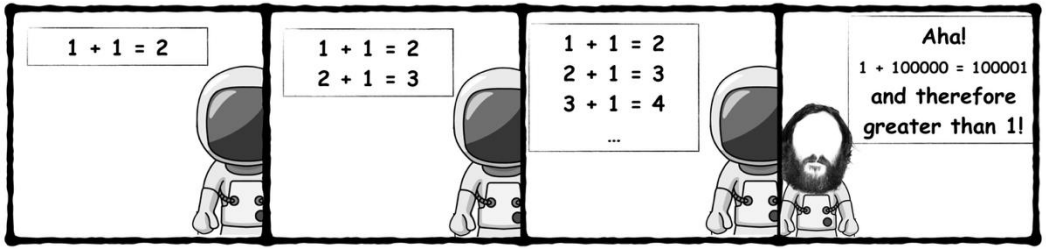
As illustrated by

Myopic generalization

Excessive generalization



Getting stuck in a rut



Data Driven Generalization & Lemma Discovery

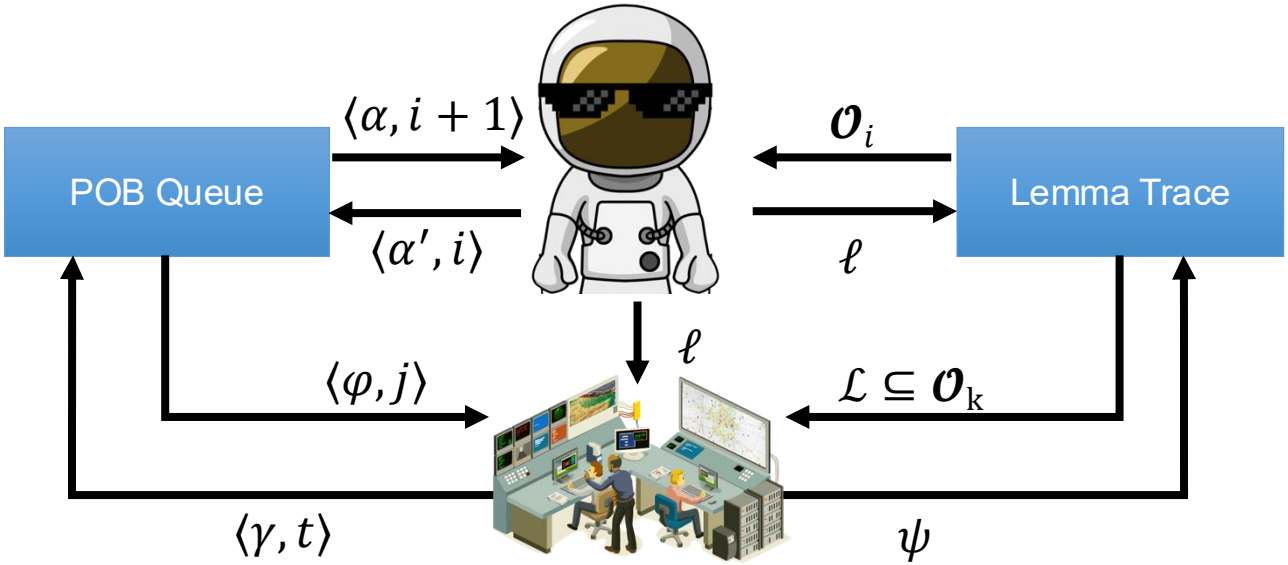
Global view of the current solver state
- **group** lemmas (and pobs) based on syntactic/semantic similarity
  - we currently use anti-unification on interpreted constants
- detect whenever global proof is diverging and mitigate

One lemma to rule them all
- **merge** lemmas in group to form a single *universal* lemma
- interpolation and inductive generalization can be applied to generalize further
- new lemma reduces the global proof by blocking all POBs in its group

Reduce, reuse, recycle
- under-approximate groups that cannot be merged in current theory
- learn multiple (simple) lemmas to block a (complex) proof obligation

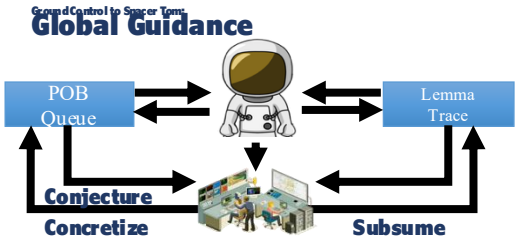
Ground Control to Spacer Tom:

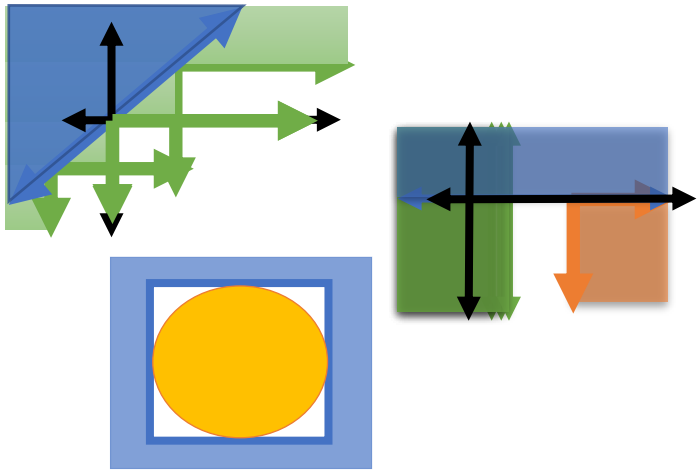
Global Guidance



Hari Govind V. K., J. Chen, S. Shoham, G.; Global Guidance for Local Generalization in Model Checking. CAV 2020

Conclusion

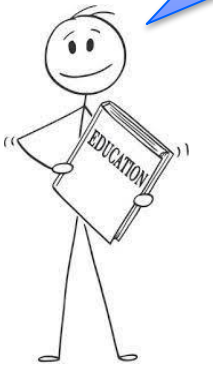




Global guidance technique to mitigate limitations of local reasoning

Stable under different interpolation strategies

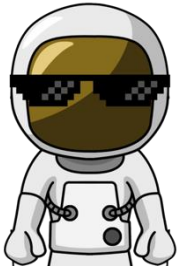
Data driven guidance for MC is better than both invariant inference and local reasoning



Lets build a model checker to analyze smart contracts!

A. Scott Wesley*

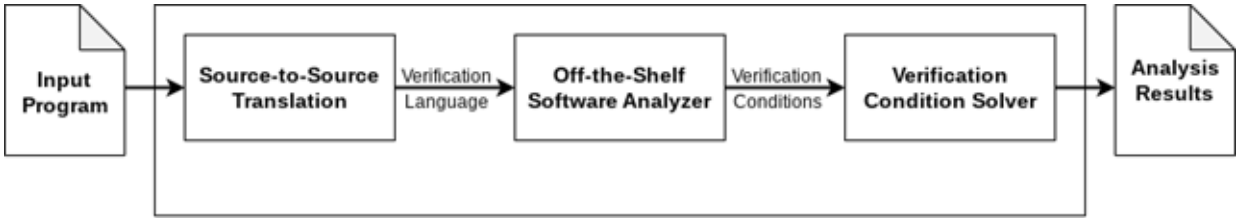
SeaHorn



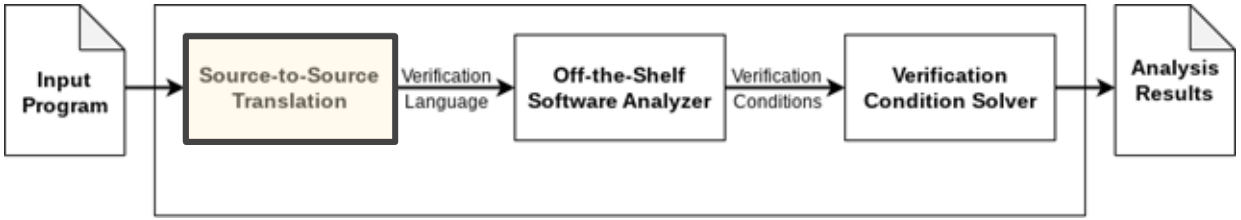
Spacer

* Not an accurate representation

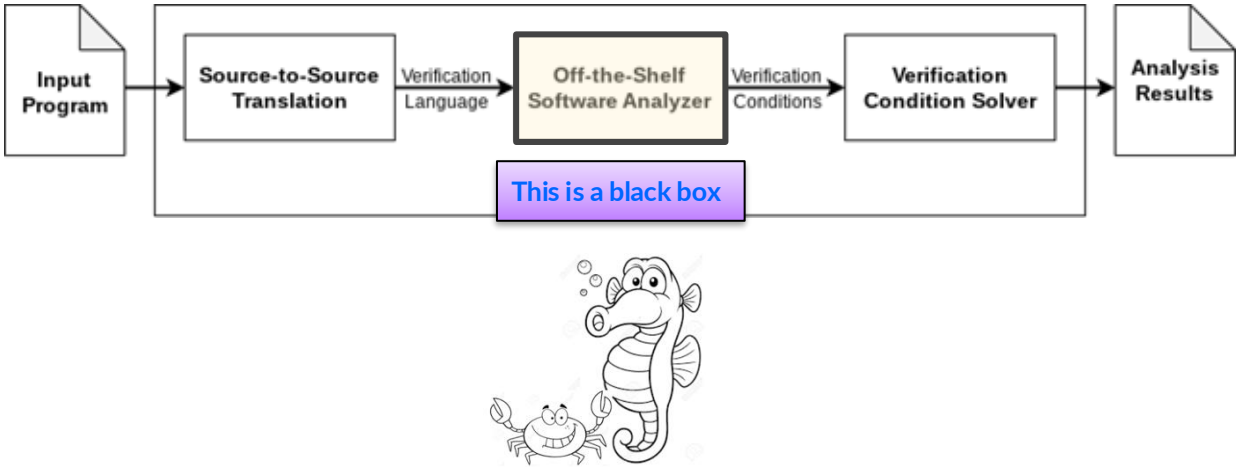
Building Tools from Off-the-Shelf Analyzers



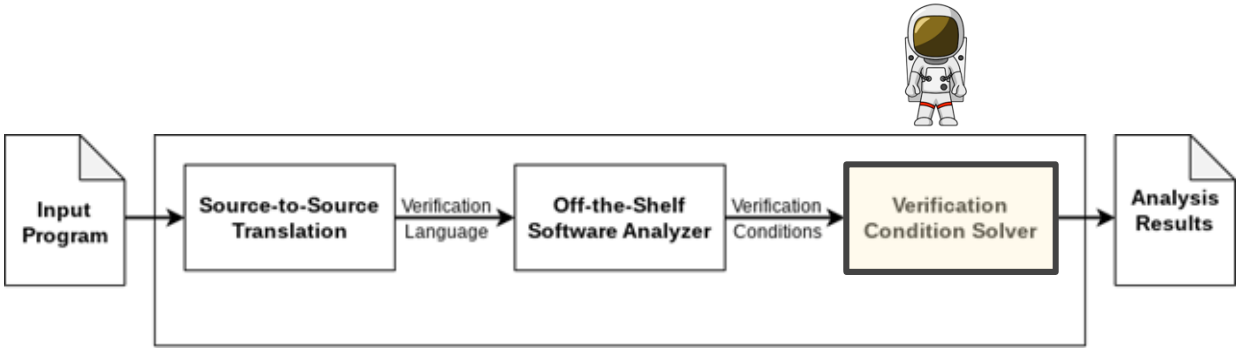
Building Tools from Off-the-Shelf Analyzers



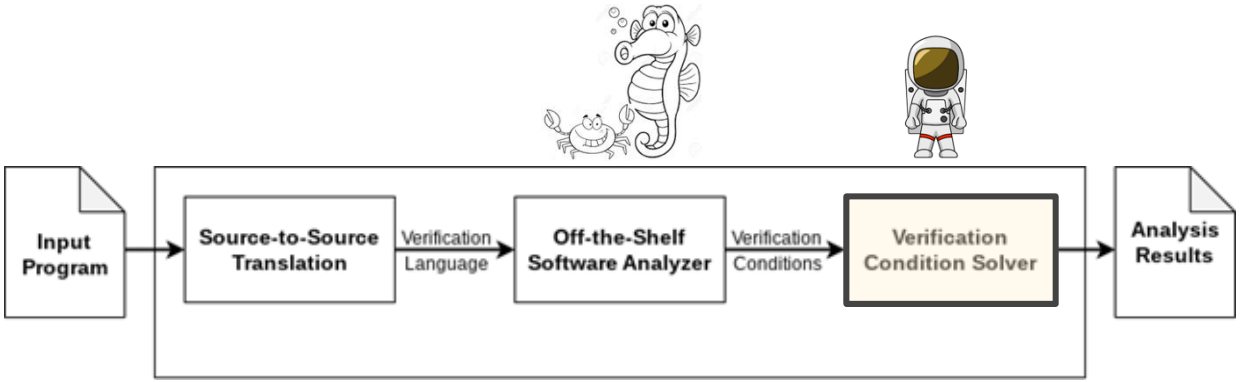
Building Tools from Off-the-Shelf Analyzers



Building Tools from Off-the-Shelf Analyzers

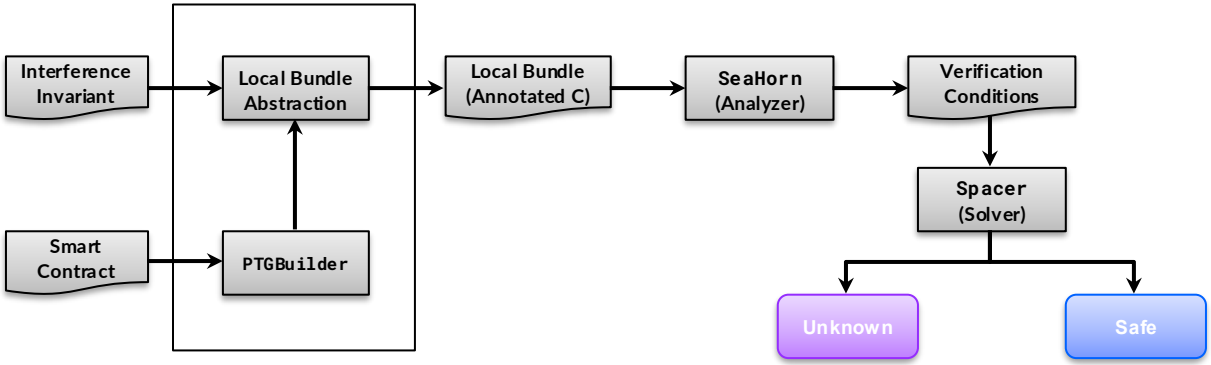


Building Tools from Off-the-Shelf Analyzers



There are many well-known examples of this design pattern (i.e., using off-the-shelf analyzers to build new software verification tools). In this talk we focus on SmartACE, which is built upon SeaHorn.

SmartACE: An Overview

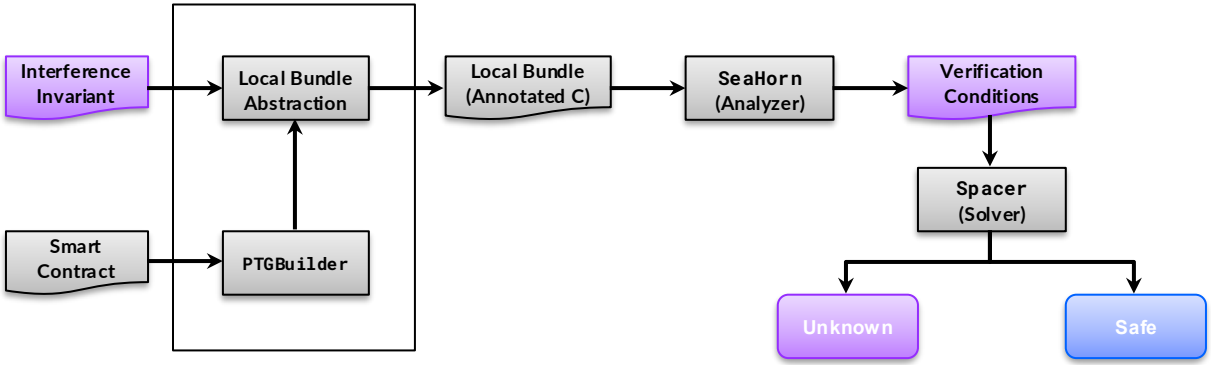


Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE (2022). S. Wesley, M. Christakis, J. A. Navas, R. J. Trefler, V. Wüstholz, A. Gurfinkel. In VMCAI 2022.

SmartACE: An Overview

Parameterized Compositional Model Checking (PCMC)

- specialized to Smart Contracts (addresses are participants)
- verification via a combination of
  - Interference Invariants and Inductive Loop Invariants

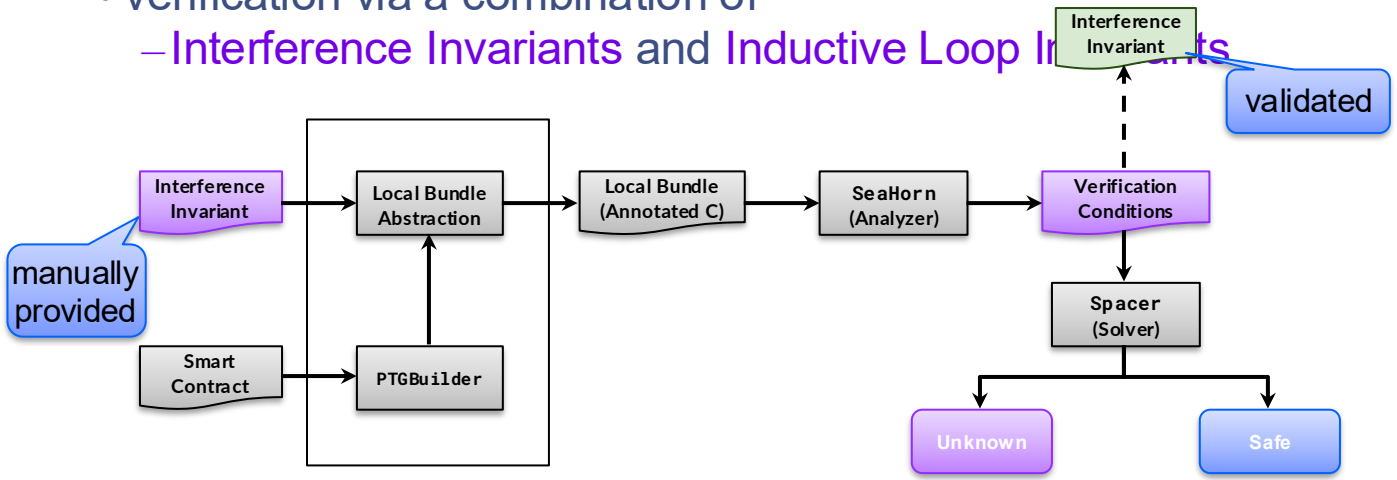


Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE (2022). S. Wesley, M. Christakis, J. A. Navas, R. J. Trefler, V. Wüstholz, A. Gurfinkel. In VMCAI 2022.

Interference Invariants Manual but Validated

Parameterized Compositional Model Checking (PCMC)

- specialized to Smart Contracts (addresses are participants)
- verification via a combination of
  - Interference Invariants and Inductive Loop In[illegible]ts

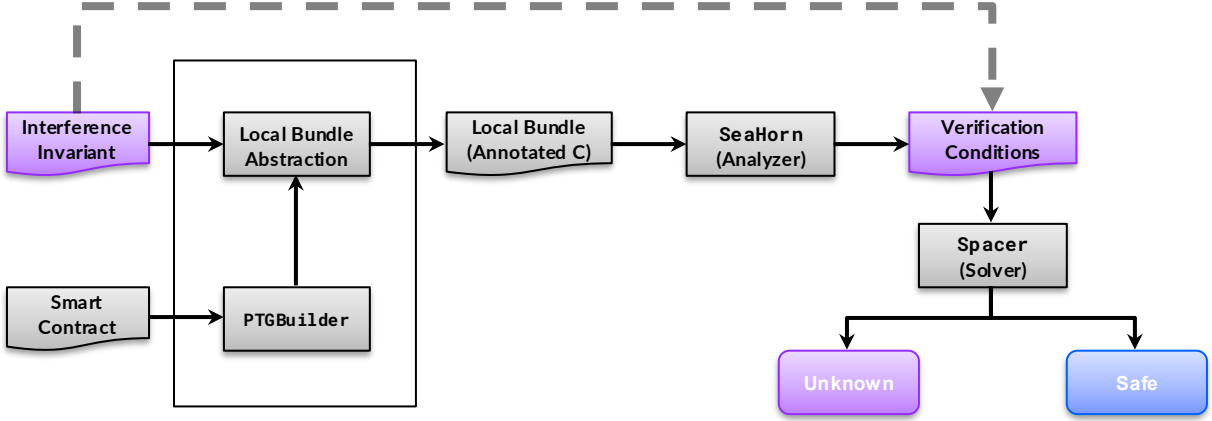


Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE (2022). S. Wesley, M. Christakis, J. A. Navas, R. J. Trefler, V. Wüstholz, A. Gurfinkel. In VMCAI 2022.

SmartACE: Automation?

Can discovery of interference invariants be automated?

- should be easy, CHCs can model multiple invariants!



Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE (2022). S. Wesley, M. Christakis, J. A. Navas, R. J. Trefler, V. Wüstholz, A. Gurfinkel. In VMCAI 2022.

All Ingredients for automation are known…

Parameterized Compositional Model Checking

Kedar S. Namjoshi[1(✉)] and Richard J. Trefler[2(✉)]

[1] Bell Laboratories, Nokia, Murray Hill, NJ, USA
kedar@research.bell-labs.com
[2] University of Waterloo, Waterloo, ON, Canada
trefler@cs.uwaterloo.ca

SMT-Based Model Checking for Recursive Programs

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki

Carnegie Mellon University, Pittsburgh, PA, USA

SMT-Based Verification of Parameterized Systems

Arie Gurfinkel
SEI/CMU, USA
University of Waterloo, Canada
arie.gurfinkel@uwaterloo.ca

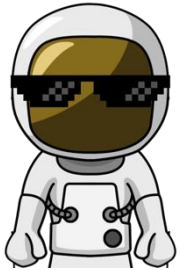
Sharon Shoham
Tel Aviv University, Israel
sharon.shoham@gmail.com

Yuri Meshman
Technion, Israel
yurim@gmail.com

The SeaHorn Verification Framework

Arie Gurfinkel[1(✉)], Temesghen Kahsai[2], Anvesh Komuravelli[3], and Jorge A. Navas[4]

[1] Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA
arie@sei.cmu.edu
[2] NASA Ames, Carnegie Mellon University, Pittsburgh, USA
temesghen.kahsai-1@nasa.gov
[3] Computer Science Department, Carnegie Mellon University, Pittsburgh, USA
anvesh@cs.cmu.edu
[4] NASA Ames, SGT, Pittsburgh, USA
jorge.a.navaslaserna@nasa.gov



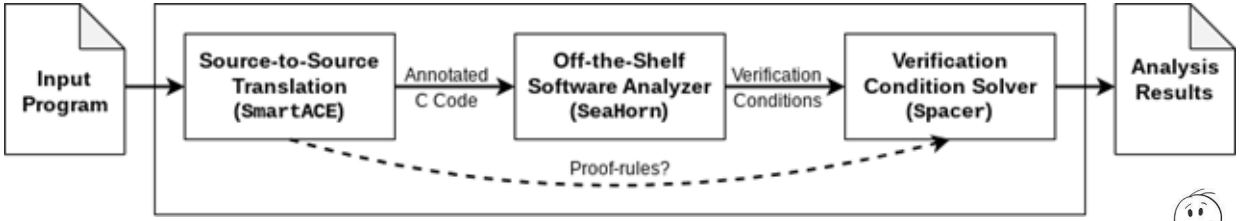
UNIVERSITY OF WATERLOO

Automation is Hard due to an Impedance Mismatch

SmartACE reduces SC verification to sequential verification w/ SeaHorn

SeaHorn reduces verification of sequential programs to CHC

SmartACE has no way to communicate with Spacer/CHC directly!



Specifying Proof Rules in Source Code

As a first step, let's see what it would look like to specify proof rules at the level of source code. As a **toy example,** consider the case of encoding **adequate loop invariant inference**.

Original Code

```
1 void main(int x) {
2   assume(x >= 0);
3   int i = 0; int y = 0;
4   while (i < x) {
5     i += 1; y += 2;
6   }
7   assert(y == 2 * x);
8 }
```

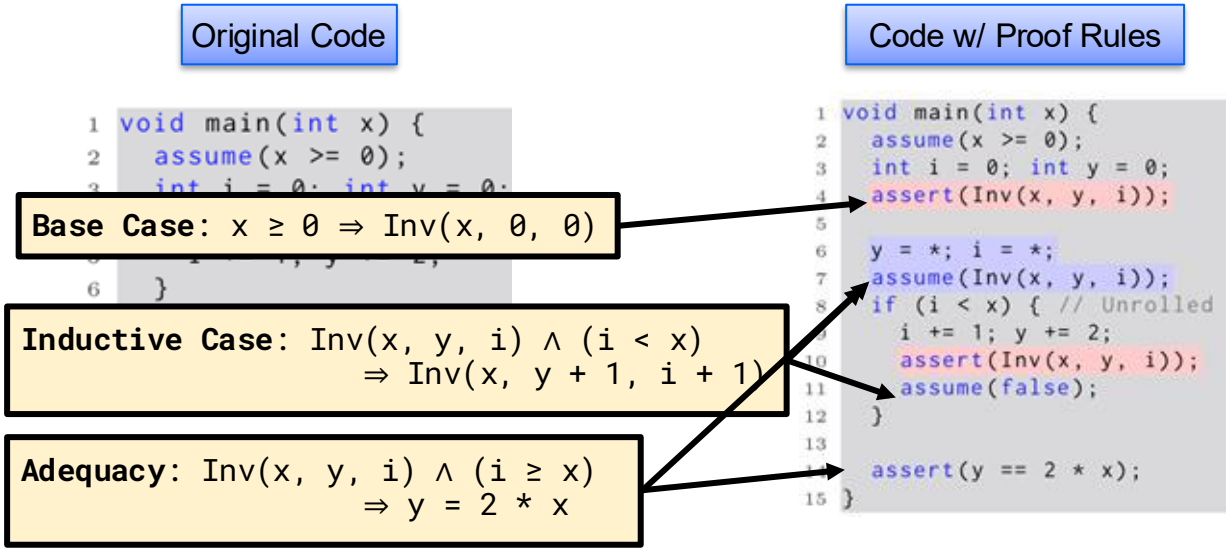
Code w/ Proof Rules

```
1  void main(int x) {
2    assume(x >= 0);
3    int i = 0; int y = 0;
4    assert(Inv(x, y, i));
5
6    y = *; i = *;
7    assume(Inv(x, y, i));
8    if (i < x) { // Unrolled
9      i += 1; y += 2;
10     assert(Inv(x, y, i));
11     assume(false);
12   }
13
14   assert(y == 2 * x);
15 }
```

Problem: find Inv so that all asserts are satisfied

Specifying Proof Rules in Source Code

As a first step, let's see what it would look like to specify proof rules at the level of source code. As a **toy example,** consider the case of encoding **adequate loop invariant inference**.



Problem: find Inv so that all asserts are satisfied

Proof Obligations as Invariant Synthesis

It is well known that invariant inference reduces to program synthesis. However, the problem described on the previous slide does not fit well within any of the existing frameworks for program synthesis. The goal of our project is to characterize and formally study this synthesis problem.

Synthesis solutions which do not fit, include…

Specification Synthesis

Looks for maximal solution

SyGuS

SemGuS

“Sketching”

These solve general purpose synthesis which is computationally harder than verification

ECOOP 2024

Inductive Predicate Synthesis Modulo Programs (IPS-MP)

Scott Wesley

Jorge A. Navas

Valentin Wüstholz

Maria Christakis

Arie Gurfinkel
Richard Trefler

An Overview of the IPS-MP Language

We define a programming language that is sufficient for specifying compositional proof rules. The features of the language build on the loop example.

Imperative Verification Language *(Standard Features)*	We start with a standard imperative verification language. This includes **loops**, **function calls**, sources of **non-determinism**, **assumptions**, and **assertions**.
Partial Predicates *(IPS-MP Features)*	We then introduce support for partial predicates. These are pure Boolean functions with **one or more holes** in the implementation. These functions **may only appear** in assume or assert statements.

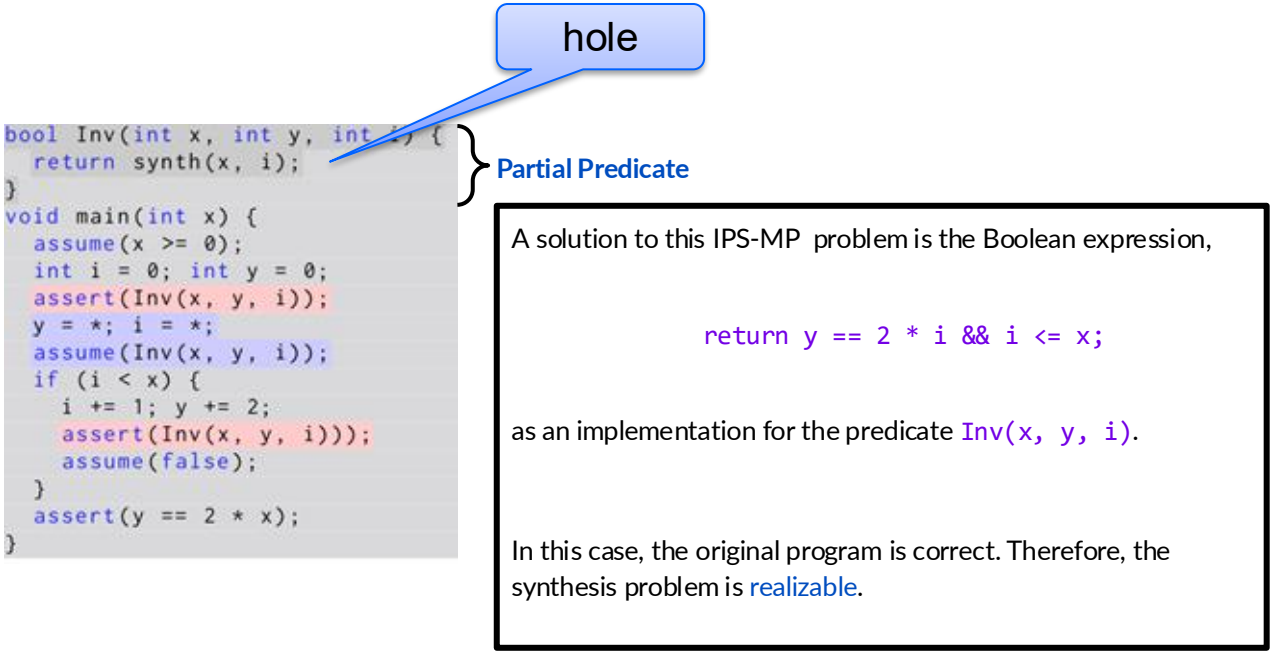
The IPS-MP Problem

The solution to an IPS-MP problem is an implementation for each predicate, that can be obtained by filling each hole with a Boolean expression, such that the resulting “closed” program is correct with respect to all of its assertions.

No Partial Predicates	Partial Predicates Without Assertions	General IPS-MP
Inter-procedural program verification	Specification synthesis without biases (trivial)	A form of adequate proof-rule synthesis

IPS-MP: Inductive Predicate Synthesis Modulo Programs

Formalizing Our Toy Example as an IPS-MP Problem



A solution to this IPS-MP problem is the Boolean expression,

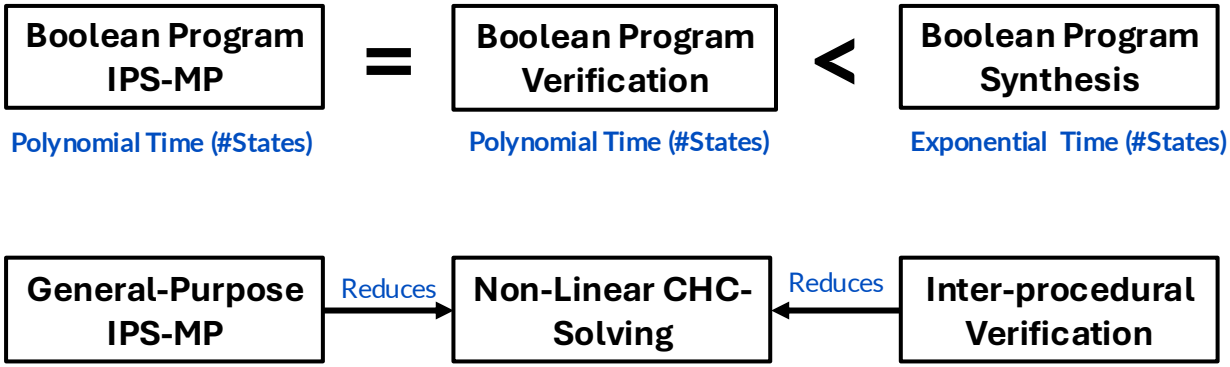
```
return y == 2 * i && i <= x;
```

as an implementation for the predicate `Inv(x, y, i)`.

In this case, the original program is correct. Therefore, the synthesis problem is realizable.

Do not be confused! IPS-MP is defined **modulo** programs. Therefore, it is not necessary to transform loops in this way. A more typical application of IPS-MP is to introduce predicates for other invariant types.

Worst Case Complexity of IPS-MP



IPS-MP: Summary of Contributions

We identify a new synthesis problem (IPS-MP) with many applications to the design of compositional program analysis tools

We establish that IPS-MP is strictly easier than general synthesis

We show that IPS-MP reduces to non-linear CHC-solving

- we use this to prototype an IPS-MP solver on top of SeaHorn

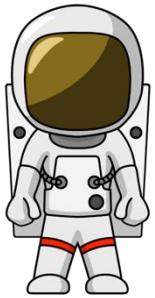
We establish a set of benchmarks to evaluate future IPS-MP solvers

We show that IPS-MP can be used to automate an existing tool

- specifically, we automate our prior tool SmartACE that used to require manually specified interference invariants

S. Wesley, J.A. Navas, V. Wüstholz, M. Christakis, R. Trefler. Inductive Predicate Synthesis Modulo Programs. ECOOP 2024

Art, Science, and Magic of CHCs



Model Checking of Safety Properties is CHC satisfiability

- Logic: Constrained Horn Clauses (CHC)
- “Decision” procedure: Spacer
- Constraints: arithmetic, bv, **arrays**, **quantifiers**, **adt + recfn**, …

Art: finding the right encoding from the problem domain to logic

- the difference between easy to impossible
- encodings can “simulate” specialized algorithms

Science: Progress, termination (when decidable)

- while the underlying problem is undecidable, many fragment or sub-problems are decidable

Magic: actually solving useful problems

- interpolation, heuristics, generalizations, …
- the list is endless

Compositional Array Invariants

```
void fn_arr1 (size_t sz) {
  assume(sz > 0);
  size_t data_sz = sizeof(int)*sz;
  int *data = malloc(data_sz);
  memset(data, 0, data_sz);

  int max = 0;
  size_t sid = nd();
  assume(0 <= sid && sid < sz);
  while (1) {

    // pick an index id
    size_t id = nd();
    assume(0 <= id && id < sz);
    int v = data[id];

    // p1: forall i :: i == sid ==> data[i] == 0
    if (id == sid) assert(v == 0);

    // p2: forall i :: data[i] <=  max
    assert(v <= max);

    // update v and max
    if (id != sid) { v += 1; }
    if (v > max) { max = v; }

    // update data[id] with new value
    data[id] = v;
  }
}
```

Compositional Array Invariants as IPS-MP

```
void fn_arr1_ips_mp(size_t sz) {
  assert(DataInv(nd(), 0, 0, nd()));

  int max = 0;
  size_t sid = nd();
  assume(0 <= sid && sid < sz);
  while (1) {

    // pick an index id
    size_t id = nd();
    assume(0 <= id && id < sz);

    // abstraction of int v = data[id];
    int v = nd();
    assume(DataInv(id, v, max, sid));

    // p1: forall i :: i == sid ==> data[i] == 0
    if (id == sid) { assert(v == 0); }

    // p2: forall i :: data[i] <=  max
    assert(v <= max);

    // assume data invariant of all other indexes
    size_t oid; int u;
    {
      // pick another array index
      oid = nd();
      assume(0 <= oid && oid != id && oid < sz);

      // DataInv is true of the other index as well
      int u = nd();
      assume(DataInv(oid, u, max, sid));
    }
```

```
int PRED_TEMPLATE DataInv(size_t idx, int v, int m, size_t sid) {

  // initially, max == 0 and (forall i :: data[i] == 0)
  if (m == 0 && v == 0) return 1;

  return synth(idx, v, m, sid);
}
```

```
int synth_sol(size_t idx, int v, int m, size_t sid) {
  if (idx == sid) return v == 0;
  return 0 <= v && v <= m;
}
```

```
    // update v and max
    if (id != sid) {
      v += 1;
    }
    if (v > max) {
      max = v;
    }

    // model: data[id] = v;
    assert(DataInv(id, v, max, sid));
    // check non-interference
    assert(DataInv(oid, u, max, sid));
  }
}
```

END