Program Verification with Constrained Horn Clauses

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

April 12th, 2023

joint work with **A. Komuravelli**, S. Chaki, G. Fedyukovich, S. Shoham, N. Bjørner, **Hari Govind V. K.**, Y. (Jeff) Chen

UNIVERSITY OF WATERLOO

Infamous Software Disasters

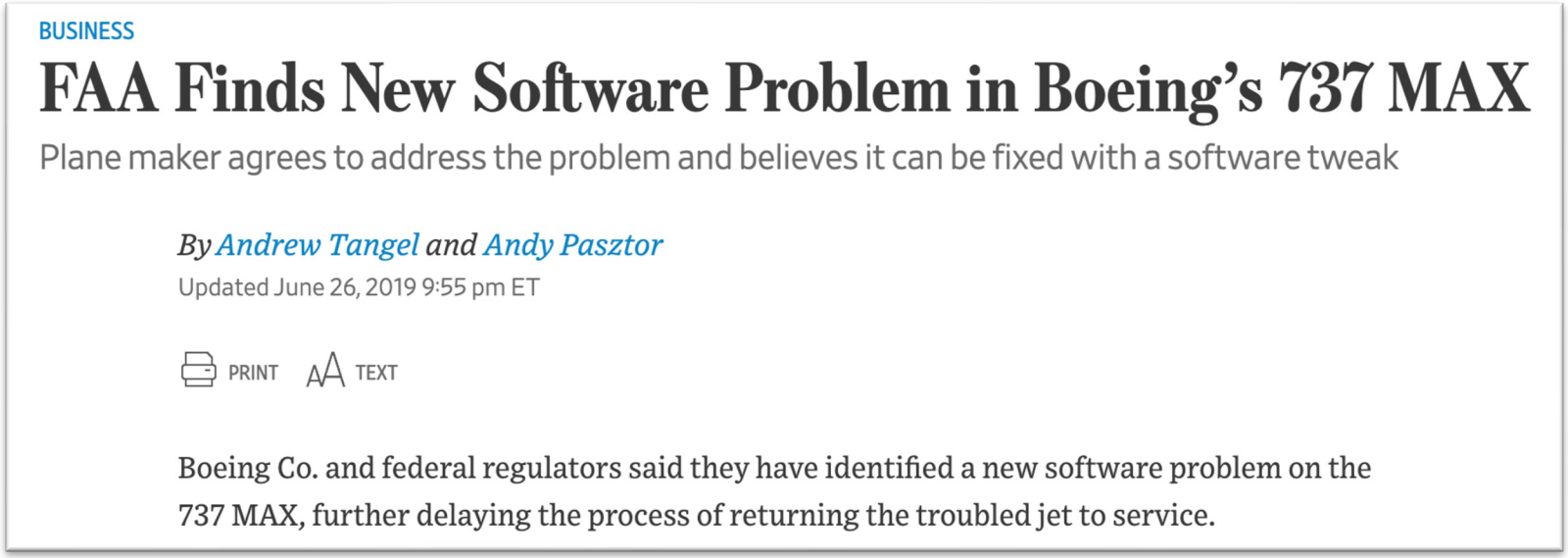
Between 1985 and 1987, **Therac-25** gave patients massive overdoses of radiation, approximately 100 times the intended dose. Three patients died as a direct consequence.

On February 25, 1991, during the Gulf War, an American **Patriot Missile** battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people.

On June 4, 1996 an unmanned **Ariane 5** rocket launched by the European Space Agency forty seconds after lift-off. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million.

http://www5.in.tum.de/~huckle/bugse.html

“Recent” Software “Disasters”



BUSINESS

FAA Finds New Software Problem in Boeing’s 737 MAX

Plane maker agrees to address the problem and believes it can be fixed with a software tweak

By *Andrew Tangel* and *Andy Pasztor*

Updated June 26, 2019 9:55 pm ET

PRINT TEXT

Boeing Co. and federal regulators said they have identified a new software problem on the 737 MAX, further delaying the process of returning the troubled jet to service.

Opinion
Technology

The millennium bug was real – and 20 years later we face the same threats

Martyn Thomas

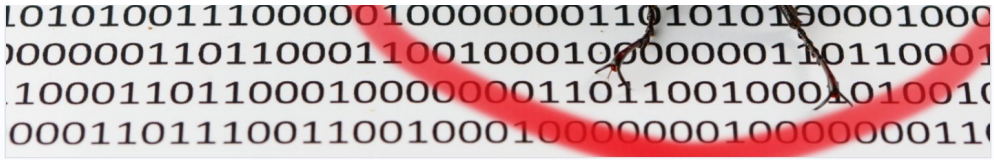
Tue 31 Dec 2019 09.00 GMT

The Y2K problem is now seen as a bit of a joke – but only a fool would be complacent about the vulnerability of IT systems

https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html#slide1

“Smart” Contracts Disasters

https://news.bitcoin.com/25-of-all-smart-contracts-contain-critical-bugs/



25% of All Smart Contracts Contain Critical Bugs

13,732 views | Jul 10, 2018, 11:38pm

Blockchain Smart Contracts: More Trouble Than They Are Worth?

Sherman Lee Contributor ⓘ
Asia
I write about deep tech, crypto, and artificial intelligence.

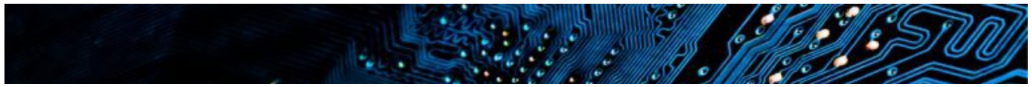
“Smart” Contracts Disasters

PYMNTS

PYMNTS TV | Today | B2B | Retail | Fintech | ConnectedEconomy™ | Crypto | EMEA

Aku’s Nightmare: $34M Locked Forever as Flaw Highlights Danger of Smart Contracts

BY **PYMNTS** | APRIL 25, 2022



In the case of the high-profile Aku Dreams project, created by former baseball player Micah Johnson, a series of coding errors turned into a $34 million disaster, locking 11,539 ether into a smart contract that cannot pay out.

But Aku Dreams’ problems cannot be totally attributed to hackers. The part of the smart contract that caused the problem was not exploited directly, and thus gives a better argument for proponents who say that smart contracts should not be made unchangeable.

https://www.pymnts.com/blockchain/2022/akus-nightmare-34m-locked-forever-as-flaw-highlights-danger-of-smart-contracts/

https://twitter.com/0xInuarashi/status/1517674505975394304

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

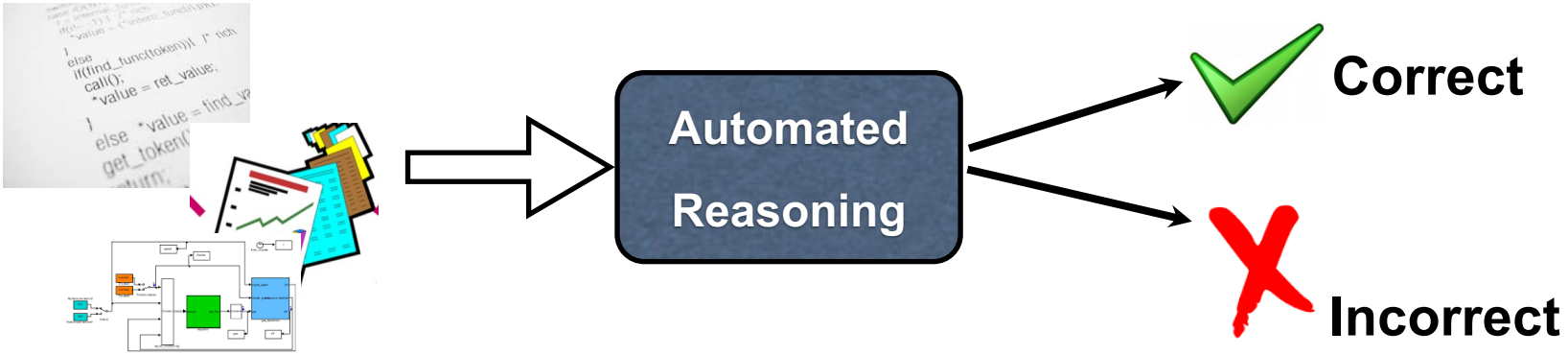
February 24, 2015 Envisage Written by Stijn de Gouw. $s

Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) by Joshua Bloch (the designer of Java Collections who also pointed out that most binary search algorithms were broken). TimSort is today used as the default sorting algorithm for Android SDK, Sun's JDK and OpenJDK. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

http://envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/

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.

Automated Software Analysis

Model Checking

[Clarke and Emerson, 1981]

[Queille and Sifakis, 1982]

Abstract Interpretation

[Cousot and Cousot, 1977]

Symbolic Execution

[King, 1976]

Automated Verification

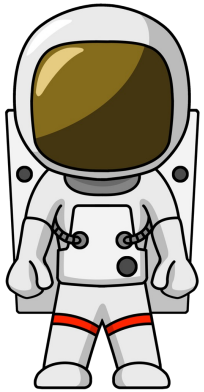
Deductive Verification

- 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”

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 p_i

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

- This is the usual FOL satisfiability

$\mathcal{T}$**-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

Program $\models \boldsymbol{\varphi}$	**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

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

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

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

Example CHC: is this SAT?

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

No! This set of clauses is unsatisfiable

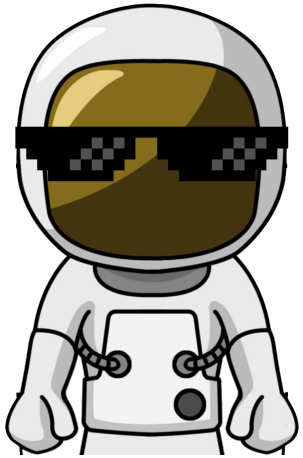
Justification is a refutation by **resolution** and **instantiation**

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

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



Spacer

INTERACTIVE TUTORIAL

https://github.com/agurfinkel/spacer-on-jupyter/blob/master/FSU_2023.ipynb

try it on Google Colab

Applications of CHCs

Prototyping different strategies and proof rules for verification

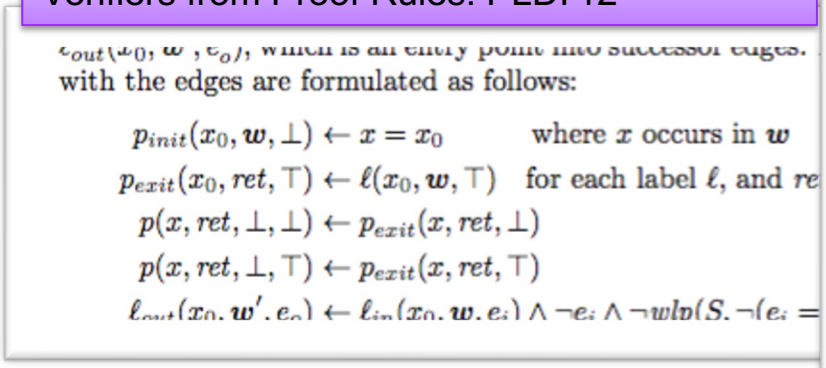
- verification by inductive invariants
- modular invariants
- predicate abstraction
- modular proof rules for concurrent systems
- verification of parameterized systems
- type inference for refinement type systems
- synthesis
- …
- create new verification tools by reducing to CHCs

Building automated verification tools

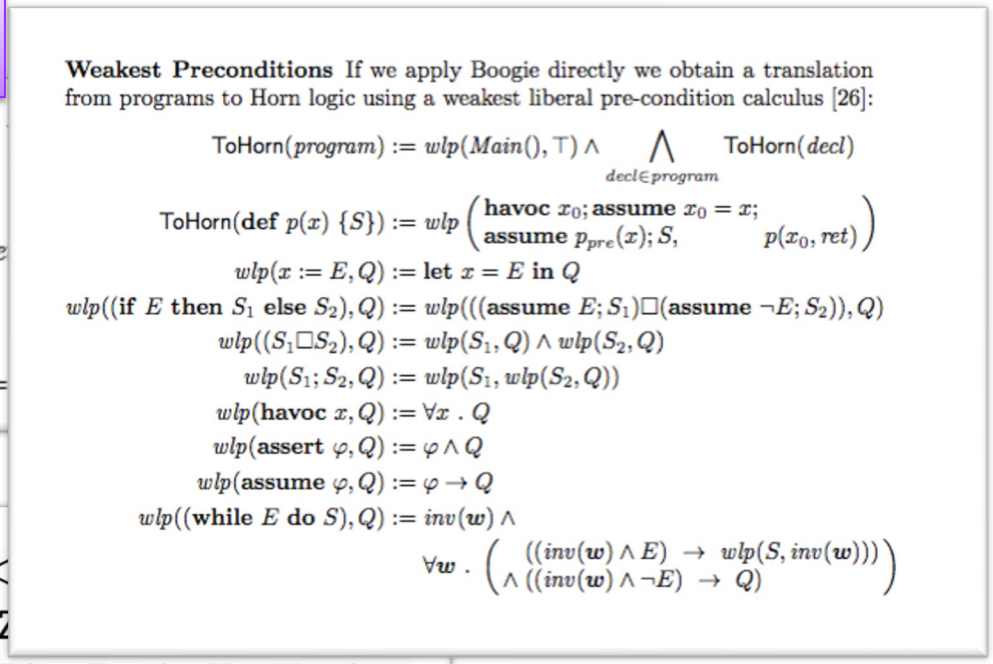
- SeaHorn, JayHorn, RustHorn, …
- SmartACE, SolCMC, …

Horn Clauses for Program Verification

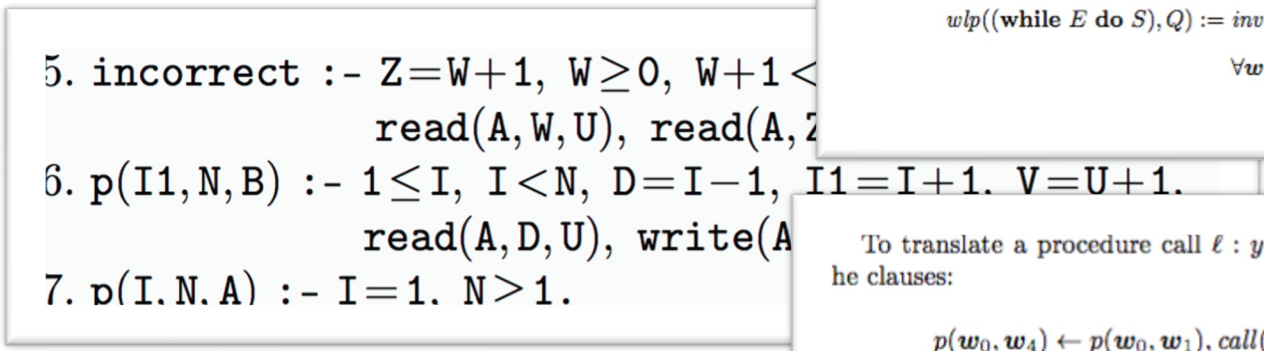
Rybalchenko et al. Synthesizing Software Verifiers from Proof Rules. PLDI'12


with the edges are formulated as follows:

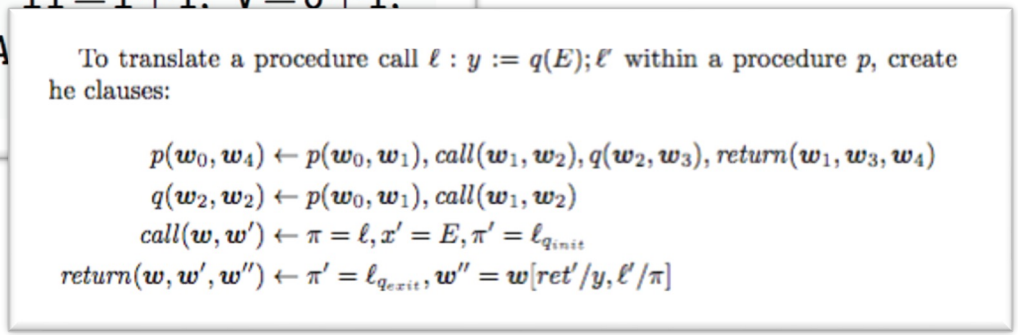
$$p_{init}(x_0, \boldsymbol{w}, \bot) \leftarrow x = x_0 \qquad \text{where } x \text{ occurs in } \boldsymbol{w}$$
$$p_{exit}(x_0, ret, \top) \leftarrow \ell(x_0, \boldsymbol{w}, \top) \qquad \text{for each label } \ell, \text{ and } re$$
$$p(x, ret, \bot, \bot) \leftarrow p_{exit}(x, ret, \bot)$$
$$p(x, ret, \bot, \top) \leftarrow p_{exit}(x, ret, \top)$$


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

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


```
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


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

$$p(\boldsymbol{w}_0, \boldsymbol{w}_4) \leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}_1), call(\boldsymbol{w}_1, \boldsymbol{w}_2), q(\boldsymbol{w}_2, \boldsymbol{w}_3), return(\boldsymbol{w}_1, \boldsymbol{w}_3, \boldsymbol{w}_4)$$
$$q(\boldsymbol{w}_2, \boldsymbol{w}_2) \leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}_1), call(\boldsymbol{w}_1, \boldsymbol{w}_2)$$
$$call(\boldsymbol{w}, \boldsymbol{w}') \leftarrow \pi = \ell, x' = E, \pi' = \ell_{q_{init}}$$
$$return(\boldsymbol{w}, \boldsymbol{w}', \boldsymbol{w}'') \leftarrow \pi' = \ell_{q_{exit}}, \boldsymbol{w}'' = \boldsymbol{w}[ret'/y, \ell'/\pi]$$

Bjørner, Gurfinkel, McMillan, and Rybalchenko:

Horn Clause Solvers for Program Verification

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',

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

multi-threaded program P is safe

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

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

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

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

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

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

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) $\quad \mathsf{init}(g, x_1) \wedge \cdots \wedge \mathsf{init}(g, x_n) \rightarrow \mathit{Inv}(g, \ell_{\mathsf{init}}, x_1, \ldots, \ell_{\mathsf{init}}, x_k)$

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

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

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

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 ar
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

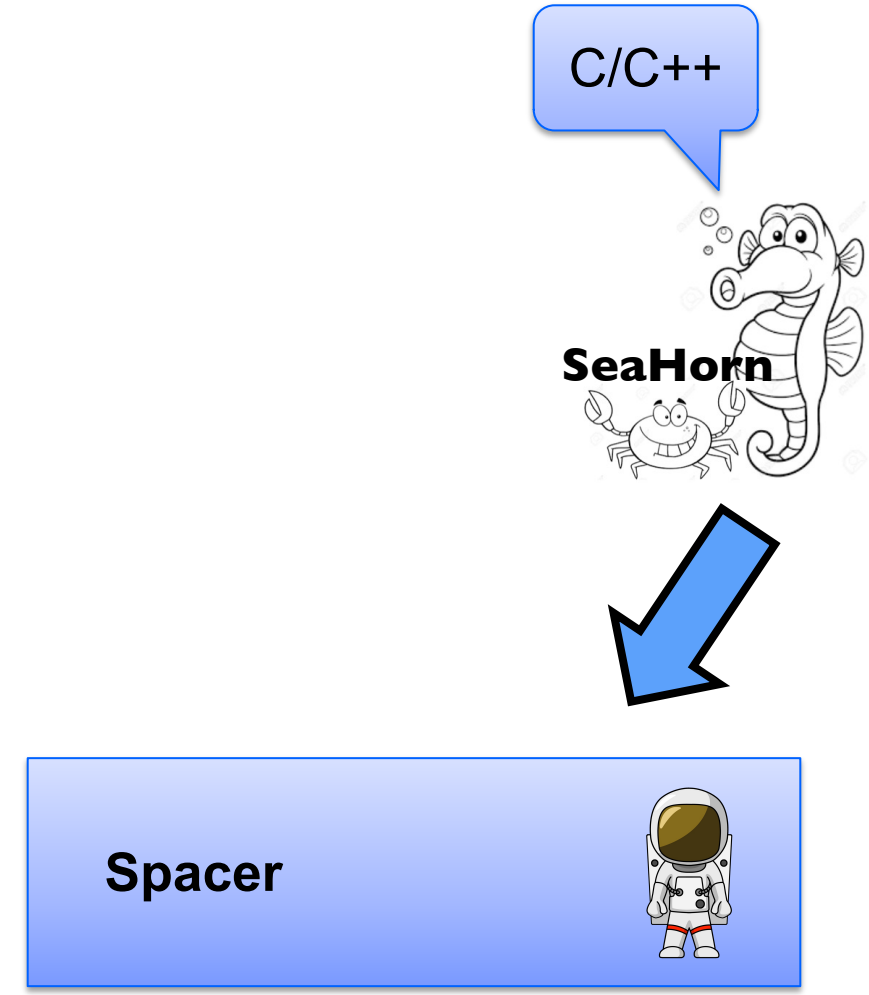
$$
\begin{aligned}
&\mathit{Init}(i, j, \overline{v}) \wedge \mathit{Init}(j, i, \overline{v}) \wedge \\
&\quad \mathit{Init}(i, i, \overline{v}) \wedge \mathit{Init}(j, j, \overline{v}) \Rightarrow I_2(i, j, \overline{v}) \\
&\quad I_2(i, j, \overline{v}) \wedge \mathit{Tr}(i, \overline{v}, \overline{v}')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

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

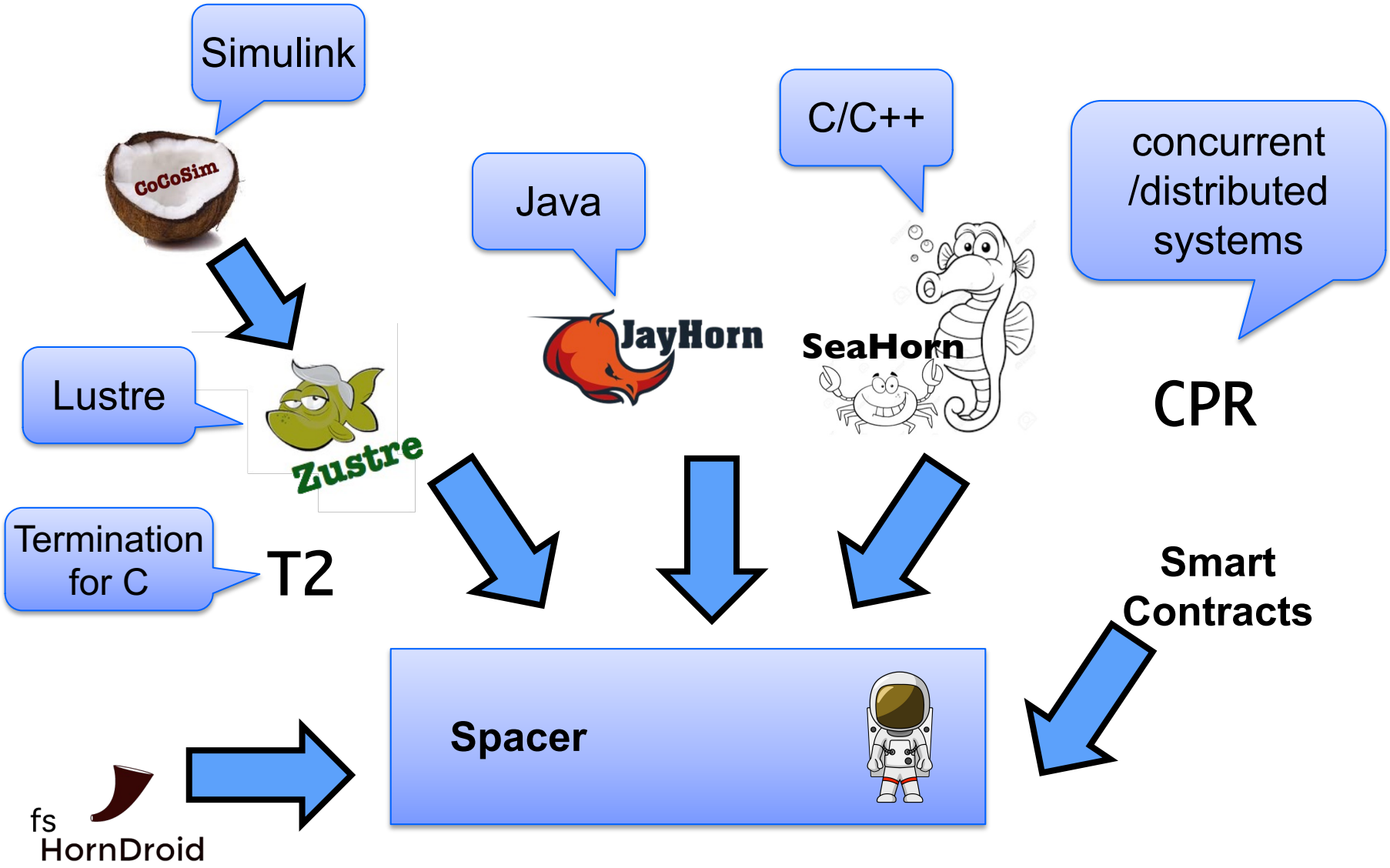
Gurfinkel et al. SMT-Based Verification of Parameterized Systems. FSE 2016

UNIVERSITY OF WATERLOO

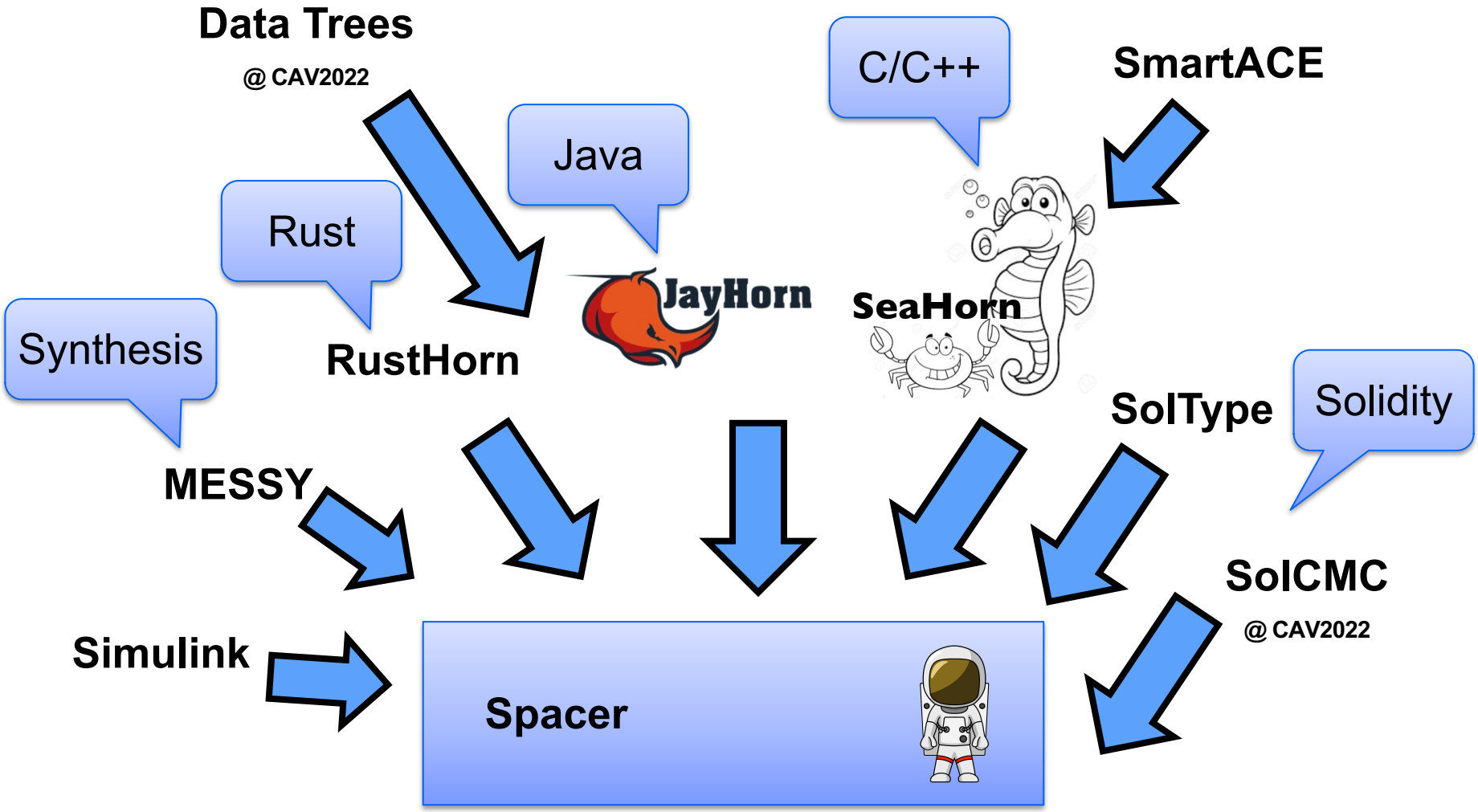
Logic-based Algorithmic Verification



Logic-based Algorithmic Verification



Logic-based Algorithmic Verification (in 2022)



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/

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

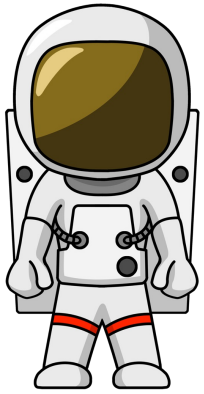
Growing collection of benchmarks

- 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, …

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

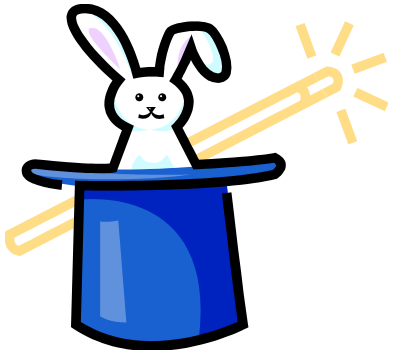
Science: Progress, termination (when decidable)

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

Magic: really solving useful problems

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

END



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, …)

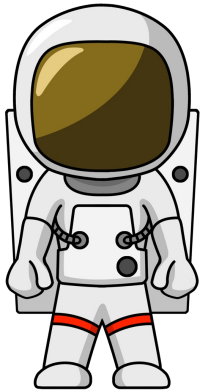
Interpolation-based Model Checking

- Duality, QARMC, …

SMT-based Unbounded Model Checking (building on IC3/PDR)

- **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
  - *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
- 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

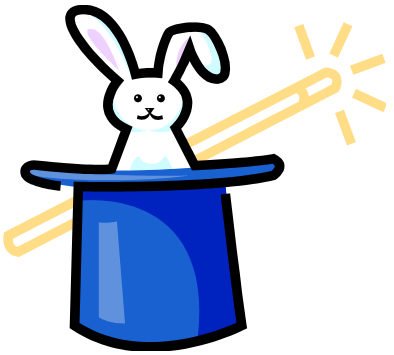
Develop a procedure ***P*** for a decidable problem

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

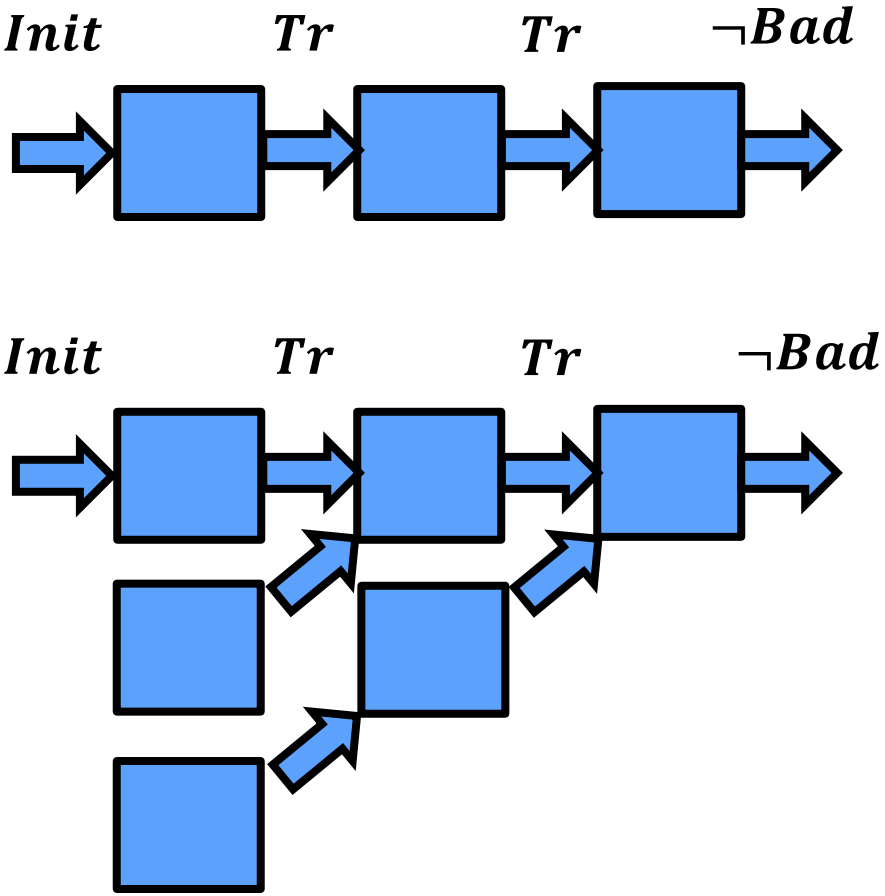
Make Progress

- 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?

IC3, PDR, and friends



Finite State Machines

(HW model checking)

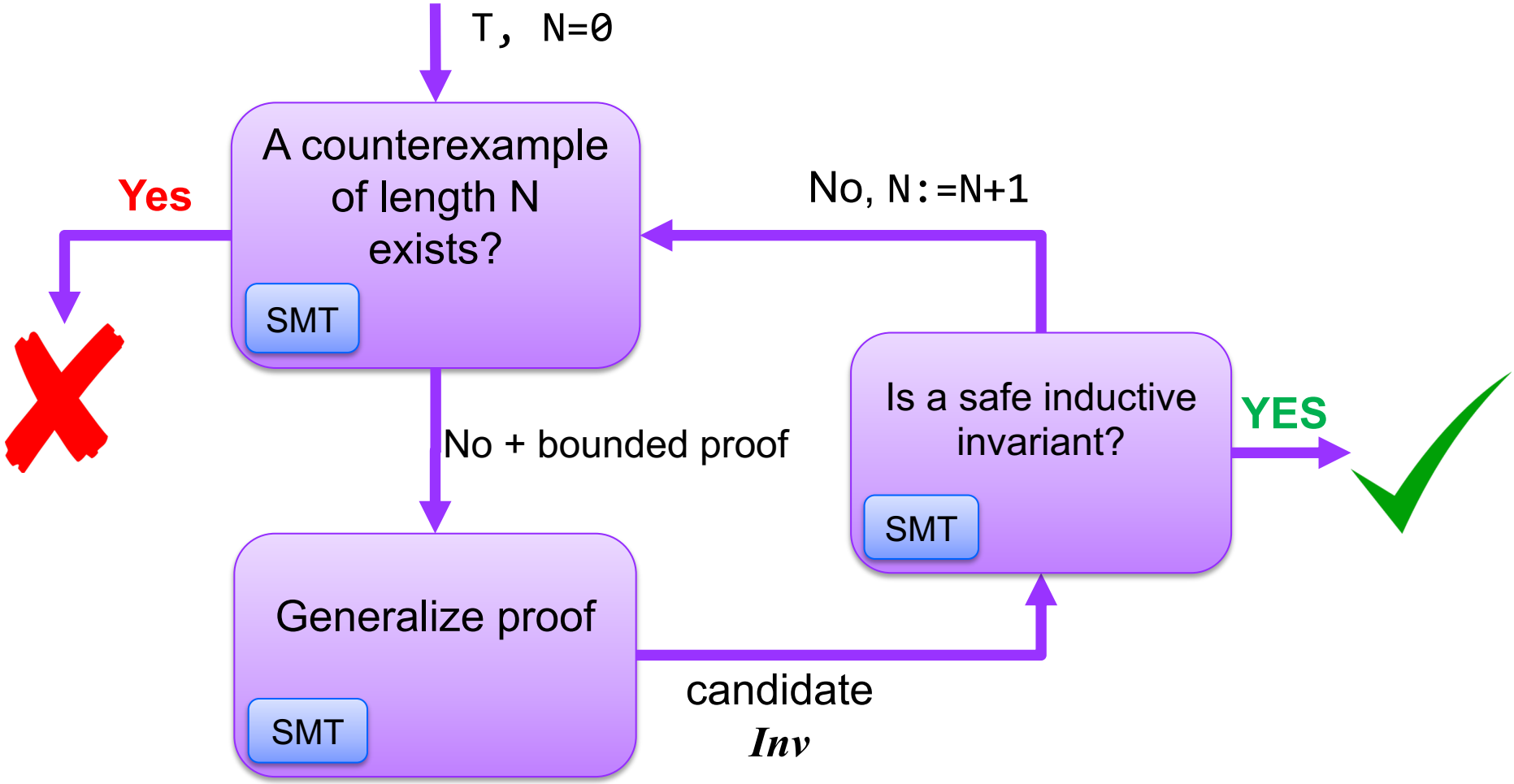
[Bradley, VMCAI 2011]

Push Down Machines

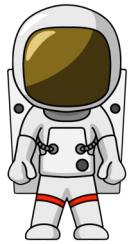
(SW model checking)

[Hoder&Bjørner, SAT 2012]

Verification by Incremental Generalization



SPACER

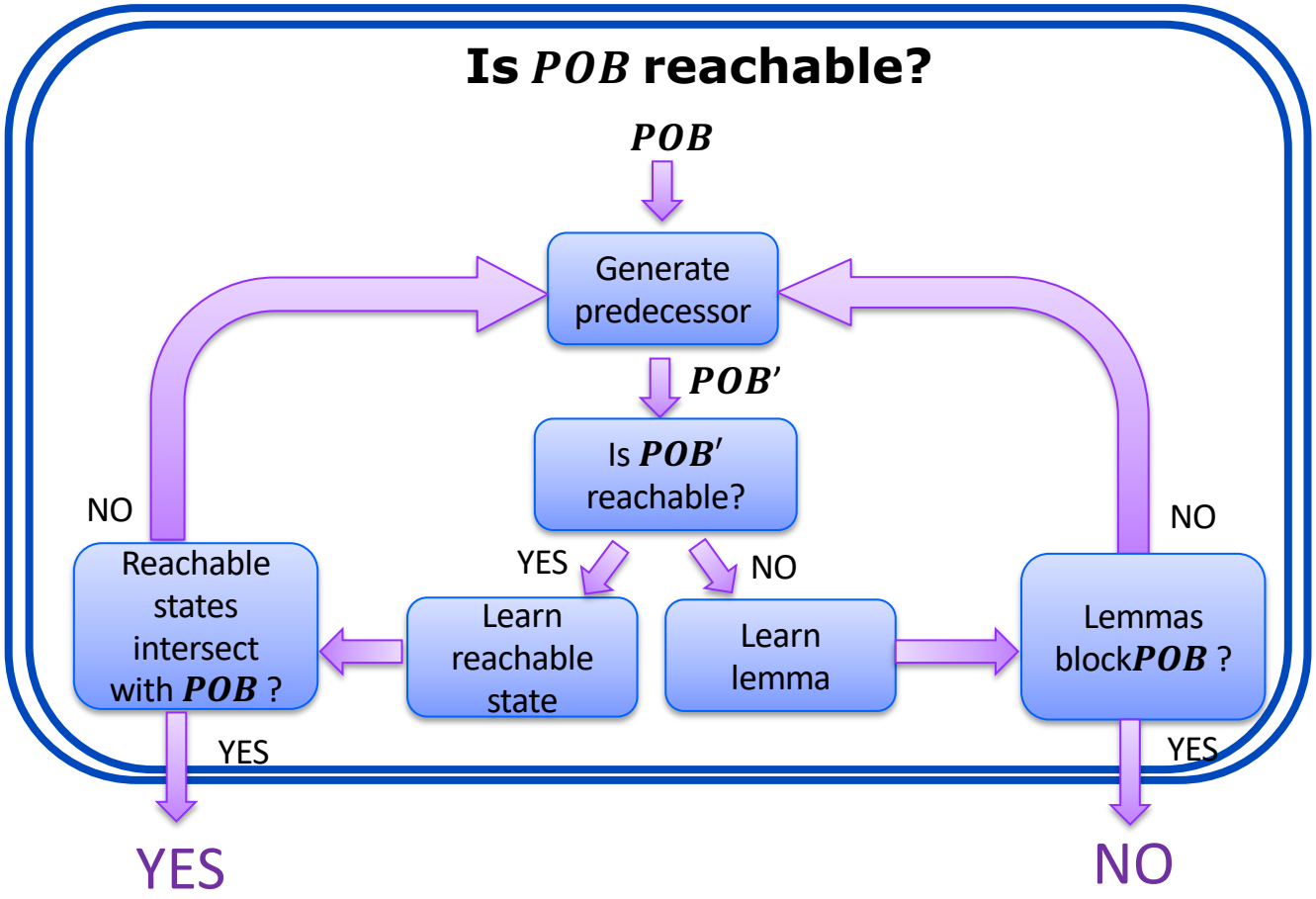


IC3-style search for solutions to CHCs

Works by recursively *blocking* ***proof obligations*** (POB)

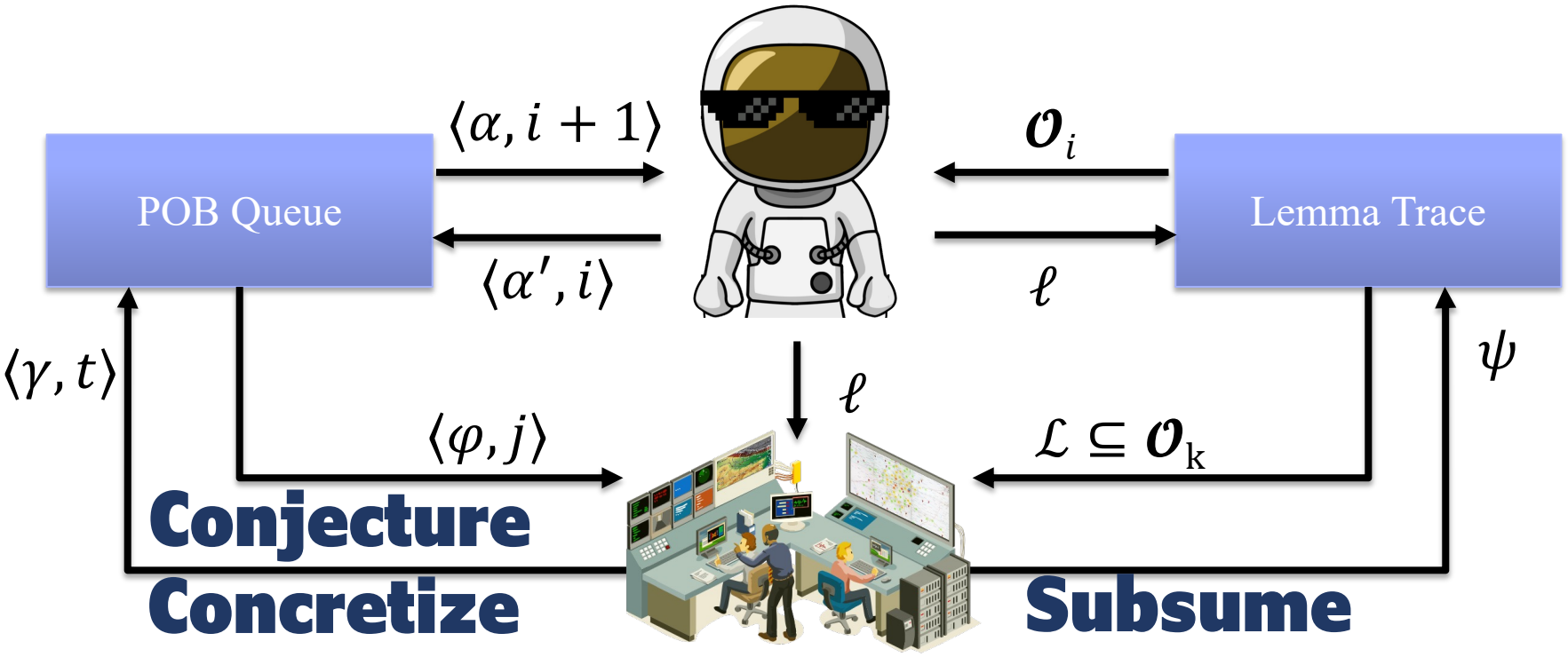
POB
- BAD states
- Predecessors to BAD states

Generate predecessors using quantifier elimination (Model Based Projection)



Ground Control to Spacer Tom:

Global Guidance



beyond Spacer

CHC SOLVERS

Report on the 2022 edition

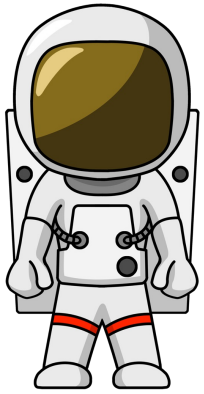
https://chc-comp.github.io/

Emanuele De Angelis, Inst. for Systems Analysis and Computer Science - National Research Council, Italy

Hari Govind V K, University of Waterloo, Canada

https://chc-comp.github.io/CHC-COMP2022_presentation.pdf

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: really solving useful problems

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

END