SeaHorn: A CHC-based Verification Tool

Jorge A. Navas

Stanford Research Institute

LOPSTR/PPDP, Sep 5, 2018
2007

- [j1] Mario Méndez-Lojo, Jorge A. Navas, Manuel V. Hermenegildo:

- [c4] Jorge A. Navas, Edison Mera, Pedro López-García, Manuel V. Hermenegildo:
  User-Definable Resource Bounds Analysis for Logic Programs. ICLP 2007: 348-363

- [c3] Mario Méndez-Lojo, Jorge A. Navas, Manuel V. Hermenegildo:
  A Flexible, (C)LP-Based Approach to the Analysis of Object-Oriented Programs. LOPSTR 2007: 154-168
True = code satisfies the safety requirement + certificate
False = code violates the safety requirement + cex
Building Automated Reasoning Tools is Time Consuming

Parse the program

Produce an optimized intermediate representation with a reduced number of cases

Build a verification engine

Support for procedures, pointers, arrays, etc.
Goals and Audience

Minimize effort when facing a new verification task
build reusable logic-based verification technology and static analysis techniques

Useful to **software developers**: efficient, user-friendly, trusted, certificate-producing, . . .

Useful to **researchers** in verification
help to assess the effectiveness of a new idea as quick as possible
In this talk ...

1. SeaHorn Overview
2. Demo
3. Constrained Horn Clauses for Verification
4. Solving CHCs
5. Conclusions and Current/Future Work
And many great collaborators such as Bjørner, Gange, Komuravelli, Sondergaard, Stuckey, etc.
SeaHorn Workflow

- **Property Spec**
- **Verification Environment**
- **Property Checker**
- **Code Under Analysis (CUA)**
- **Verification Problem (VP)**
- **SeaHorn**
- **Good + Verification Certificate (Cert)**
- **Bad + Counterexample (CEX)**
- **TestGen**
- **Test harness (Test)**
Writing a Property Checker

Similar to a dynamic checker (e.g., clang sanitizers)
A significant development effort for each new property
new specialized static analyses to rule out trivial cases
different instrumentations have affect on performance
Developed by a domain expert
understanding of verification techniques is useful (but not required)
3-6 month effort for a new property
but many things can be reused between similar properties (out-of-bounds, null-deref, taint checking, use-after-free, etc)

SeaHorn property checkers
memory safety (out of bounds, null pointer)
going work to improve scalability and usability
taint analysis (developed by Princeton)
SeaHorn Architecture

Problem Encoding:
- sequential safety
- information flow
- inconsistencies
- regression verification
- multi-thread safety

Precision:
- integers, floating-point numbers
- pointers
- memory contents
- procedures

Efficiency:
- small vs large step

LLVM Opt:
- SSA
- DCE
- Peephole
- CFG Simplification

Devirtualization and Exception Lowering

Property Checker:
- Buffer overflow
- Null dereferences

Slicing Assertions

Model checking

Abstract Interpretation

BMC

ML-based Learning Synthesis

CLP

Boogie

MCMT

C/C++

McSema

x86

Solidity

SeaHorn

LOPSTR/PPDP, Sep 5, 2018
1 SeaHorn Overview

2 Demo

3 Constrained Horn Clauses for Verification

4 Solving CHCs

5 Conclusions and Current/Future Work
Constrained Horn Clauses (CHCs)

A Constrained Horn Clause (CHC) is a formula:

\[ \forall V \cdot ( \phi \land p_1(X_1) \land \cdots \land p_k(X_k) \rightarrow h(X)), \text{ for } k \geq 0 \]

- \( \phi \): a constraint over \( F \) and \( V \) wrt some background theory
- \( p_i(X_i) \): an application \( p(t_1, \ldots, t_n) \) of an \( n \)-ary \( p \in P \) for FO terms \( t_i \) constructed from \( F \) and \( X_i \)
- \( h(X) \): either defined analogously to \( p_i \) or false

\( F \): function symbols, \( P \): predicate symbols, and \( V \): variables
A model of a set of CHCs is an interpretation $\mathcal{J}$ of each predicate $p_i$ that makes all clauses valid.

A set of CHCs is satisfiable if it has a model, and is unsatisfiable otherwise.

In the context of verification:

- a program satisfies a property iff its corresponding CHCs are satisfiable.
- models for CHCs correspond to inductive invariants and summaries.
- derivations to false correspond to counterexample.
CHCs are expressive enough to model a broad set of interesting verification and inference problems.

CHCs are very amenable for abstractions.
Verification of Sequential Programs

\[
\begin{align*}
\text{Init}(X) & \rightarrow \text{Inv}(X) \\
\text{Inv}(X) \land \text{Step}(X, X') & \rightarrow \text{Inv}(X') \\
\text{Inv}(X) & \rightarrow \neg \text{Bad}(X)
\end{align*}
\]
Verification of Multi-Threaded Programs

Predicate Abstraction and Refinement for Verifying Multi-Threaded Programs

Ashutosh Gupta  Corneliu Popescu  Andrey Rybalchenko

\[
\begin{align*}
\land_{i \in \{1, \ldots, N\}} (Init(X) \rightarrow Inv_i(X)) \\
\land_{i \in \{1, \ldots, N\}} (Inv_i(X) \land Step_i(X, X') \rightarrow Inv_i(X')) \\
\land_{i \in \{1, \ldots, N\}} (Inv_i(X) \land Env_i(X, X') \rightarrow Inv_i(X')) \\
\land_{i, j \in \{1, \ldots, N\}, i \neq j} (Inv_j(X) \land Step_j(X, X') \rightarrow Env_i(X, X')) \\
Inv_1(X) \land \ldots \land Inv_N(X) \rightarrow \neg Bad(X)
\end{align*}
\]
Verification of Array Manipulating Programs

\[
\begin{align*}
\text{Init}(X, A) & \rightarrow \text{Inv}(X, A) \\
I(X, A) \land \text{Step}(X, A, X', A') & \rightarrow I(X', A') \\
\text{Inv}(X, A) & \rightarrow \neg \text{Bad}(X, A)
\end{align*}
\]

Step can contain array constraints of the form:

\[
\begin{align*}
a' &= \text{write}(a, i, v) \\
v &= \text{read}(a, i)
\end{align*}
\]

where \(i, v \in X \cup X', a \in A, \text{ and } a' \in A'\)
Verification of Array Manipulating Programs

Cell morphing: from array programs to array-free Horn clauses*

David Monniaux         Laure Gonnord

Abstract array \( a \) into a pair \( (k, a_k) \) st. \( a[k] = a_k \)

\[
\begin{align*}
a' &= \text{write}(a, i, v) \quad ("\text{the value at } i \text{ is } v, \text{ the rest unchanged}"
\quad):
\quad i &= k \land \text{Inv}(X, v, i, a_k) \rightarrow \text{Inv}(X, v, i, v) \\
\quad i \neq k \land \text{Inv}(X, v, k, a_k) \rightarrow \text{Inv}(X, v, k, a_k)
\end{align*}
\]

\[
\begin{align*}
v &= \text{read}(a, i) \quad ("v \text{ has new value, the rest is preserved}"):
\quad i &= k \land \text{Inv}(X, v, i, a_i) \rightarrow \text{Inv}(X, a_i, i, a_i) \\
\quad i \neq k \land \text{Inv}(X, v, k, a_k) \land \text{Inv}(X, v, i, a_i) \rightarrow \text{Inv}(X, a_i, k, a_k)
\end{align*}
\]
Finding Inconsistencies in Programs with Loops*
Temesghen Kahsaı, Jorge A. Navas, Dejan Jovanović, Martin Schäf

Automating Regression Verification
Dennis Felsing†
dennis.felsing@student.kit.edu
Sarah Grebing†
sarah.grebing@kit.edu
Vladimir Klebanov†
klebanov@kit.edu
Philipp Rümmer‡
philipp.ruemmer@it.uu.se
Mattias Ulbrich
ulbrich@kit.edu

SMT-Based Verification of Parameterized Systems
Arie Gurfinkel
SEI/CMU, USA
University of Waterloo, Canada
arie.gurfinkel@uwwaterloo.ca
Sharon Shoham
Tel Aviv University, Israel
sharon.shoham@gmail.com
Yuri Meshman
Technion, Israel
syurim@gmail.com

Verifying Array Programs by Transforming Verification Conditions
Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti

Horn Clauses for Communicating Timed Systems
Hossein Hojjat
Cornell University, USA
Philipp Rümmer
Uppsala University, Sweden
Pavle Subotic
Wang Yi

* And many more ...
A Hoare triple \( \{\text{Pre}\} P \{\text{Post}\} \) is valid iff every terminating execution of \( P \) that starts in a state satisfying \( \text{Pre} \) ends in a state satisfying \( \text{Post} \).

Validity of Hoare triples can be reduced to FOL validity by applying a predicate transformer, e.g., the Dijkstra’s weakest liberal precondition:

\[
\{\text{Pre}\} P \{\text{Post}\} \iff \text{Pre} \Rightarrow \text{wlp}(P, \text{Post})
\]
Translating to CHCs Using Weakest Liberal Preconditions

\[ Pre \rightarrow \text{wlp}(\text{Main}, \text{Post}) \land \bigwedge_{f \in \mathcal{P}} \forall x, r. \text{wlp}(B_f, S_f(x, r)) \]

\[
\begin{align*}
\text{wlp}(\text{if } C \text{ S}_1 \text{ else } S_2, \phi) & \rightarrow \ C \rightarrow \text{wlp}(S_1, \phi) \land \neg \ C \rightarrow \text{wlp}(S_2, \phi) \\
\text{wlp}(S_1; S_2, \phi) & \rightarrow \text{wlp}(S_1, \text{wlp}(S_2, \phi)) \\
\text{wlp}(x = e, \phi) & \rightarrow \phi[x \leftarrow e] \\
\text{wlp}(\text{error, } \phi) & \rightarrow \bot \\
\text{wlp}(\text{while } C \text{ B}, \phi) & \rightarrow \mathcal{I}(\overline{x}) \land \forall \overline{x}((\mathcal{I}(\overline{x}) \land C \rightarrow \text{wlp}(B, \mathcal{I}(\overline{x}))) \land (\mathcal{I}(\overline{x}) \land \neg C \rightarrow \phi)) \\
\text{wlp}(x = f(y), \phi) & \rightarrow \forall \ r. S_f(y, r) \rightarrow \phi[x \leftarrow r]
\end{align*}
\]

And apply negation, prenex, and conjunctive normal form
main() {
    x = 1;
    y = 0;
    while (y > 0) {
        x = x + y;
        y = y + 1;
    }
    assert(x ≥ y)
}
Translating to CHCs Using Dual WLP

entry:
\[
\begin{align*}
x &= 1 \\
y &= 0
\end{align*}
\]

header:
\[y > 0\]

body:
\[
\begin{align*}
x &= x + y \\
y &= y + 1
\end{align*}
\]

exit:
\[
\begin{align*}
x &\geq y
\end{align*}
\]

safe:

error:
Translating to CHCs Using Dual WLP

\[ \text{wlp}(P, \text{Post}) = \neg \text{wlp}(P, \neg \text{Post}) \]

\[
\begin{align*}
\text{entry : } & \\
& x = 1 \\
& y = 0 \\
\text{header : } & y > 0 \\
\text{body : } & x = x + y \\
& y = y + 1 \\
\text{exit : } & x \geq y \\
\text{safe : } & \\
\text{error : } & \\
\end{align*}
\]

\[
\begin{align*}
\text{entry}(x, y) & \leftarrow \text{true}. \\
\text{h}(x, y) & \leftarrow \text{entry}(x, y), x = 1, y = 0. \\
\text{b}(x, y) & \leftarrow \text{h}(x, y), y > 0. \\
\text{h}(x', y') & \leftarrow \text{b}(x, y), \\
& x' = x + y, y' = y + 1. \\
\text{exit}(x, y) & \leftarrow \text{h}(x, y), y \leq 0. \\
\text{error}(x, y) & \leftarrow \text{exit}(x, y), x < y.
\end{align*}
\]
Rule for if-then-else can cause the resulting CHCs to be exponentially larger than the original program

Solution: generate compact VCs for loop-free code

Use of **Cut-point graph (CPG)** rather than the original CFG

A CPG is a summarized CFG, where each node represents a cut-point (loop head) and each edge represents multiple loop-free paths through the CFG

CPGs preserve reachability of control locations
From CFG to CPG
Single Static Assignment (SSA): every value has a unique definition

```plaintext
int x, y, n;
l_0: x = 0; y = *;
l_1: while (x < n) {
l_2:   if (y > 0)
l_3:     x = x + y;
   else
l_4:     x = x - y;
l_5:     y = -1 * y;
}
l_6:
goto l_1

l_0: goto l_1
l_1: x_0 = \phi(0 : l_0, x_3 : l_5);
y_0 = \phi(y : l_0, y_1 : l_5);
if (x_0 < n) goto l_2 else goto l_6
l_2: if (y_0 > 0) goto l_3 else goto l_4
l_3: x_1 = x_0 + y_0; goto l_5
l_4: x_2 = x_0 - y_0; goto l_5
l_5: x_3 = \phi(x_1 : l_3, x_2 : l_4);
y_1 = -1 * y_0

Jorge A. Navas (SRI)
SeaHorn
LOPSTR/PPDP, Sep 5, 2018 27 / 49
Large-Step Encoding using Cut-point Graphs

$\phi :$

\begin{align*}
x_1 &= x_0 + y_0 \land \\
x_2 &= x_0 - y_0 \land \\
y_1 &= -1 \times y_0 \land \\
B_2 &\rightarrow x_0 < n \land \\
B_3 &\rightarrow B_2 \land y_0 > 0 \land \\
B_4 &\rightarrow B_2 \land y_0 \leq 0 \land \\
B_5 &\rightarrow ((B_3 \land x_3 = x_1) \lor (B_4 \land x_3 = x_2)) \land \\
B_5 \land x'_0 &= x_3 \land y'_0 = y_1
\end{align*}

\begin{align*}
\ell_0: & \textbf{goto } \ell_1 \\
\ell_1: x_0 &= \phi(0 : \ell_0, x_3 : \ell_5); \\
y_0 &= \phi(y : \ell_0, y_1 : \ell_5); \\
&\textbf{if } (x_0 < n) \textbf{ goto } \ell_2 \textbf{ else goto } \ell_6 \\
\ell_2: & \textbf{if } (y_0 > 0) \textbf{ goto } \ell_3 \textbf{ else goto } \ell_4 \\
\ell_3: x_1 &= x_0 + y_0; \textbf{ goto } \ell_5 \\
\ell_4: x_2 &= x_0 - y_0; \textbf{ goto } \ell_5 \\
\ell_5: x_3 &= \phi(x_1 : \ell_3, x_2 : \ell_4); \\
&y_1 = -1 \times y_0 \\
&\textbf{ goto } \ell_1 \\
\ell_6: &
\end{align*}

\[ p_1(x'_0, y'_0) \leftarrow p_1(x_0, y_0) \land \phi \]
Block-based memory model: a pointer is a pair \( \langle \text{ref}, o \rangle \) where \text{ref} uniquely defines a memory object and \( o \) defines the byte in the object being point to

\[
\text{Env} : \mathbb{V} \rightarrow \text{Ptr} \quad \text{Ptr} = \text{Ref} \times \text{Int} \quad \text{Mem} : \text{Ptr} \rightarrow \text{Ptr}
\]

Concrete memory model:

- each allocation (e.g. \texttt{malloc}) creates a fresh new object
- the number of objects is \textit{infinite}

Abstract memory model:

- the number of allocation regions is \textit{finite}
- allocation site used as an object reference

Use a whole-program pointer analysis to compute an abstract points-to graph
Run a pointer analysis to disambiguate memory

Produce a side-effect-free encoding by:

replacing each memory object $o$ to a logical array $A_o$

replacing memory accesses to a pointer $p$ within object $o$ to array reads and writes over $A_o$

$$v := *(&p + i) \mapsto v = \text{read}(A_o, i)$$

$$*(&p + i) := v \mapsto A'_o = \text{write}(A_o, i, v)$$

each write on $A_o$ produces a new version of $A'_o$ representing the array after the execution of the memory write

Accuracy of pointer analysis is vital for CHC solver’s scalability: resolve aliasing at encoding time
void \texttt{f}(\texttt{int}^* x, \texttt{int}^* y) \{ \\
    *x = 1; \\
    *y = 2; \\
\}

void \texttt{g}(\texttt{int}^* p, \texttt{int}^* q, \\
                \texttt{int}^* r, \texttt{int}^* s) \{ \\
    \texttt{f}(p,q); \\
    \texttt{f}(r,s); \\
\}

Assume \( p \) and \( q \) may alias
void f(int* x, int* y) {
    *x = 1;
    *y = 2;
}

void g(int* p, int* q, int* r, int* s) {
    f(p, q);
    f(r, s);
}

Assume p and q may alias

\[
f(p, q) \quad x, y, p, q
\]
\[
f(x, y)
\]
### CHCs Using a Context-Insensitive Pointer Analysis

```c
void f(int* x, int* y) {
    *x = 1;
    *y = 2;
}

void g(int* p, int* q, int* r, int* s) {
    f(p, q);
    f(r, s);
    f(r, s);
}
```

Assume \( p \) and \( q \) may alias

- \( f(r, s) \)
- \( f(x, y) \)
- \( x, y, p, q \)
void f(int* x, int* y) {
    *x = 1;
    *y = 2;
}

void g(int* p, int* q, int* r, int* s) {
    f(p, q);
    f(r, s);
}

Assume p and q may alias

f(r, s)

f(x, y)
void f(int* x, int* y) {
    *x = 1;
    *y = 2;
}

void g(int* p, int* q, int* r, int* s) {
    f(p, q);
    f(r, s);
}

\[
\begin{align*}
S_f(x, y, a_{xy}, a'_{xy}) &\leftarrow \\
a'_{xy} &\leftarrow \text{write}(a_{xy}, x, 1) \land \\
a''_{xy} &\leftarrow \text{write}(a'_{xy}, y, 2)
\end{align*}
\]

\[
\begin{align*}
S_g(p, q, r, s, a_{pqrs}, a''_{pqrs}) &\leftarrow \\
S_f(p, q, a_{pqrs}, a'_{pqrs}) &\land \\
S_f(r, s, a'_{pqrs}, a''_{pqrs})
\end{align*}
\]
Assume $p$ and $q$ may alias

$$f(p, q)$$

$$f(x', y')$$

$$f_{\text{sum}}(x, y)$$

Good compromise: context-sensitive: calls to $f$ do not merge

$$\{p, q\}$$

$$\{r, s\}$$

ensure CHCs are sound
Sound CHCs Using a Context-Sensitive Pointer Analysis

```c
void f(int* x, int* y) {
    *x = 1;
    *y = 2;
}

void g(int* p, int* q, int* r, int* s) {
    f(p, q);
    f(r, s);
}
```

Assume \( p \) and \( q \) may alias

\[
\begin{align*}
\text{f}(p,q) \\
\text{f}(x',y')
\end{align*}
\]

\[
\begin{align*}
\text{f}_{\text{sum}}(x,y)
\end{align*}
\]
sound CHCs using a context-sensitive pointer analysis

```c
void f (int* x, int* y) {
    *x = 1;
    *y = 2;
}

void g (int* p, int* q, int* r, int* s) {
    f(p, q);
    f(r, s);
}
```

Assume p and q may alias

\[
\begin{align*}
&f(p, q) \quad p, q \\
&f(x', y') \quad x', y' \\
&f_{sum}(x, y) \quad x, y
\end{align*}
\]
Assume p and q may alias

```c
void f(int* x, int* y) {
    *x = 1;
    *y = 2;
}

void g(int* p, int* q, int* r, int* s) {
    f(p, q);
    f(r, s);
}
```

Good compromise:
context-sensitive: calls to 

- `f(r, s)` ensures CHCs are sound
void f(int* x, int* y) {
    *x = 1;
    *y = 2;
}

void g(int* p, int* q, int* r, int* s) {
    f(p, q);
    f(r, s);
}

Assume p and q may alias

f(r,s)  \[r,s\]
f(x'',y'')  \[x'',y''\]
f\sum(x,y)  \[x,y\]
void f(int* x, int* y) {
    *x = 1;
    *y = 2;
}

void g(int* p, int* q, int* r, int* s) {
    f(p, q);
    f(r, s);
}

\[
S_f(x, y, a_{xy}, a''_{xy}) \leftarrow \\
a'_{xy} = \text{write}(a_{xy}, x, 1) \land \\
a''_{xy} = \text{write}(a'_{xy}, y, 2)
\]

\[
S_g(p, q, r, s, a_{pq}, a_{rs}, a'_{pq}, a'_{rs}) \leftarrow \\
S_f(p, q, a_{pq}, a'_{pq}) \land \\
S_f(r, s, a_{rs}, a'_{rs})
\]
void f(int* x, int* y) {
    *x = 1;
    *y = 2;
}

void g(int* p, int* q, int* r, int* s) {
    f(p, q);
    f(r, s);
}

S_f(x, y, a_{xy}, a''_{xy}) ←
    a'_{xy} = \text{write}(a_{xy}, x, 1) \land
    a''_{xy} = \text{write}(a'_{xy}, y, 2)

S_g(p, q, r, s, a_{pq}, a_{rs}, a'_{pq}, a'_{rs}) ←
S_f(p, q, a_{pq}, a'_{pq}) \land
S_f(r, s, a_{rs}, a'_{rs})

Good compromise:
context-sensitive: calls to $f$ do not merge $\{p,q\}$ and $\{r,s\}$
ensure CHCs are sound
SeaHorn Pointer Analysis

A Context-Sensitive Memory Model for Verification of C/C++ Programs*

Arie Gurfinkel¹ and Jorge A. Navas²

it is unification-based (as LLVM-DSA)
it is context-, field-, and array-sensitive
it covers a relevant subset of C/C++ programs that supports:

dynamic memory allocation
type unions, pointer arithmetic, pointer casts
inheritance, function/method calls, etc

it significantly boosts CHC solvers

https://github.com/seahorn/sea-dsa
1. SeaHorn Overview

2. Demo

3. Constrained Horn Clauses for Verification

4. Solving CHCs

5. Conclusions and Current/Future Work
Spacer: a solver for SMT-constrained Horn Clauses

Main solving engine in SeaHorn
now the default (and only) CHC solver in Z3
https://github.com/Z3Prover/z3
dev branch: https://github.com/agurfinkel/z3

Supported SMT-theories:
LIA and LRA
quantifier-free theory of arrays
universally quantified theory of arrays + arithmetic
best-effort support for bit-vectors, non-linear arithmetic, etc

Support for non-linear CHCs:
for procedure summaries in inter-procedural verification
conditions
for compositional reasoning: assume-guarantee, thread
modular, etc.

Based on IC3/PDR-based model checking
Crab: an Abstract Interpretation Library

Abstract Domains
- numerical domains: intervals, zones, boxes, etc
- 3rd party libraries: apron and elina
- arrays and symbolic domains

Analysis of a language-independent core with plugin for LLVM
- fixpoint engine based on Bourdoncle’s WTO
- widening/narrowing strategies

**Crab-Llvm**: translates to Crab language and integrates optimizations/analysis of LLVM bitecode

Support for inter-procedural and backward analysis

Extensible and open C++ library

Publicly available

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

Numerical domains
- intervals + congruences: \(5 \leq x \leq 10 \land x \mod 2 = 0\)
- zones: \(x - y \leq k\)
- wrapped intervals: intervals on machine-arithmetic integers
- non-convex:
  - DisIntervals: \(x \leq -1 \lor x \geq 1\)
- boxes: boolean combination of intervals

Symbolic domains
- terms: numerical domains + uninterpreted functions
  - \(x \leq 10 \land y = f(...) \land z = f(...) \rightarrow x \leq 10 \land y = z\)
  - \(b = \text{write}(a, i, x) \land y = \text{read}(a, i) \rightarrow x = y\)

Array domains
- array smashing: one summarized variable per array (weak updates)
- array expansion: one scalar variable per array element (strong updates)
- partition-based: weak+strong updates

Apron and Elina: octagons, polyhedra, etc
Crab Architecture and LLVM plug-in

Abstract Domains
- Terms
  - Bool x Numerical
- Pointers
- Arrays

Property Checkers
- Assertions
- DivByZero
- Null-Deref
- Array-bounds

Abstract Transformers
- Fixpoint Engine
- Forward/Backward Analysis
- Inter-procedural Analysis
- CFG /Callgraph Builder
- Heap Abstraction
- LLVM Optimizations

Invariants
Preconditions
Summaries
LLVM Plug-in

Heap Abstraction
LLVM Optimizations

Pointers
Arrays

Terms
Bool x Numerical
Integration with other tools and other solvers

SeaHorn translate CHCs to different formats
  SMTLIB2, Boogie, CLP, MCMT, etc

Spacer and Crab generate invariants

Invariant generation is a hard problem
  BMC engine for bit-level precision
  ML-based learning synthesis engine to complement Spacer and Crab
SeaHorn in Real World

SV-COMP

6,000+ files, each 1K-100K LOC

Autopilot code (absence of buffer overflows)

Verify Level 5 requirements of the NASA LADEE software stack:
Manually encode requirements in Simulink models
Verify that the requirements hold in auto-generated C
Conclusions and Current/Future Work

Build verification technology from scratch is hard
We have built many reusable verification components:

- C/C++ front-ends by reusing compiler technology
- model checking algorithms
- abstract interpretation techniques
- symbolic execution/BMC engines
- pointer analyses

Tested on C device drivers and embedded C/C++ software

Current/future work:

- Making more efficient memory safety checker
- Building executable counterexamples
- Boosting BMC and Spacer with abstract interpretation
- Arrays, machine-arithmetic, FP, new memory models
Thank you!
For latest news, blog posts, publications
http://seahorn.github.io/

Open-source software components:
https://github.com/seahorn/seahorn
https://github.com/seahorn/sea-dsa
https://github.com/agurfinkel/z3
https://github.com/seahorn/crab
https://github.com/seahorn/crab-llvm
Executable Counterexamples in Software Model Checking. **VSTTE 2018**

A Context-Sensitive Memory Model for Verification of C/C++. **SAS 2017**

Synthesizing Ranking Functions from Bits and Pieces. **TACAS 2016**

Exploiting Sparsity in Difference-Bound Matrices. **SAS 2016**

An Abstract Domain of Uninterpreted Functions. **VMCAI 2016**

Finding Inconsistencies in Programs with Loops. **LPAR 2015**

Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays. **FMCAD 2015**

The SeaHorn Verification Framework. **CAV 2015**

SMT-Based Model Checking for Recursive Programs. **CAV 2014**
IC3/PDR in One Slide

Invariants

\[ F_0 = \text{Init} \]
\[ F_i \Rightarrow F_{i+1} \]
\[ F_i \text{ and Step } \Rightarrow F'_{i+1} \]
\[ F_i \Rightarrow \text{not Bad} \]

Repeat

\[ \text{SAT}(F_k \text{ and Step and Bad'}) ? \]
\[ \text{SAT}(F_{k-1} \text{ and Step and } s_{k'}) ? \]
\[ \ldots \]
\[ F_{k-1} = F_{k-1} \text{ and not } s_{k-1} \]
\[ F_k = F_k \text{ and not } s_k \]

if \( S_k \) is reachable then CEX
else strengthen \( F_k \) to exclude \( s_k \)

until \( F_k \text{ and Step } \Rightarrow \text{not Bad} \)

If \( F_k \Rightarrow F_{k-1} \) then SAFE
else \( k=k+1 \)
IC3/PDR: General case

Given $F_0, F_1, \ldots F_k$, set $F_{k+1} = \neg Bad$

Apply a backward search:

1. Find predecessor $s_k$ in $F_k$ that can reach $Bad$
   check if $F_k \land Step \land Bad'$ is sat

2. If none exists, then if $F_{k+1} \Rightarrow F_k$ return “safe”. Otherwise, move to next iteration

3. If exists, then try to find a predecessor $s_{k-1}$ to $s_k$ in $F_{k-1}$
   check if $F_{k-1} \land Step \land s_k'$ is sat

4. If none exists, then $F_k = F_k \land \neg s_k$ and go back to 3

5. Otherwise, recur on $(s_{k-1}, F_{k-1})$

If we reach Init then exits a CEX!
From finite IC3/PDR to solving CHCs

Theories with infinite models:
- cannot block one state at a time
- cannot enumerate all possible predecessors

Non-linear CHCs:
- increase the number of predecessors
Generalize predecessors: $F_{k-1} \land \text{Step} \land s'_k$

Find a cube $m$ st $m \Rightarrow \exists V'. F_{k-1} \land \text{Step} \land s'_k$

Block more than one state

$s \models F_k \land \text{Step} \land \text{Bad}$ and $F_{k-1} \land \text{Step} \land s$ is unsat

$F_{k-1} \land \text{Step} \Rightarrow \neg s$ iff $\neg s \land F_{k-1} \land \text{Step} \Rightarrow \neg s$

$\neg s$ is inductive relative to $F_{k-1}$

Find $c$ st $c \Rightarrow \neg s$, $c \land F_{k-1} \land \text{Step} \Rightarrow c$, and $\text{Init} \Rightarrow c$.

If one exists $F_k = F_k \land c$

Moreover, for every $i \leq k$ $F_i = F_i \land c$ because $c$ is also inductive relative to $F_{k-2}, \ldots, F_0$!

Push forward

if $c \in F_k$ and $c \notin F_{k+1}$ and $F_k \land c \land \text{Step} \Rightarrow c'$ then $F_{k+1} = F_{k+1} \land c$ (for all $1 \leq k \leq N - 1$)