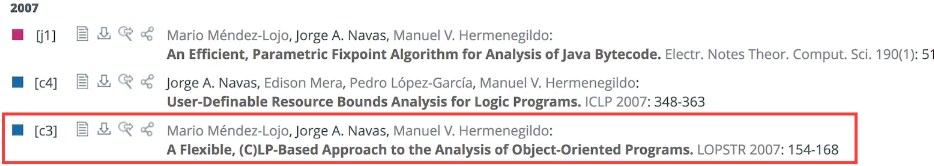
SeaHorn: A CHC-based Verification Tool

Jorge A. Navas

Stanford Research Institute

LOPSTR/PPDP, Sep 5, 2018



2007

[j1] Mario Méndez-Lojo, Jorge A. Navas, Manuel V. Hermenegildo:
An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode. Electr. Notes Theor. Comput. Sci. 190(1): 51

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

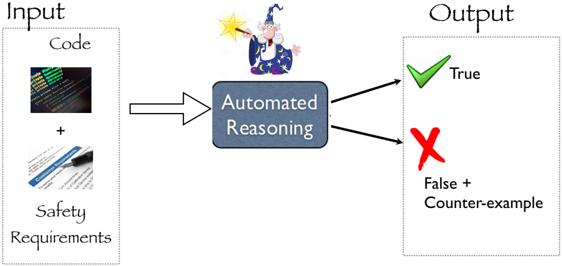
[c3] Mario Méndez-Lojo, Jorge A. Navas, Manuel V. Hermenegildo:
A Flexible, (C)LP-Based Approach to the Analysis of Object-Oriented Programs. LOPSTR 2007: 154-168

Automated Reasoning for Software

Model Checking

Abstract Interpretation

Symbolic Execution



True = code <u>satisfies</u> the safety requirement + certificate

False = code <u>violates</u> 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, . . .

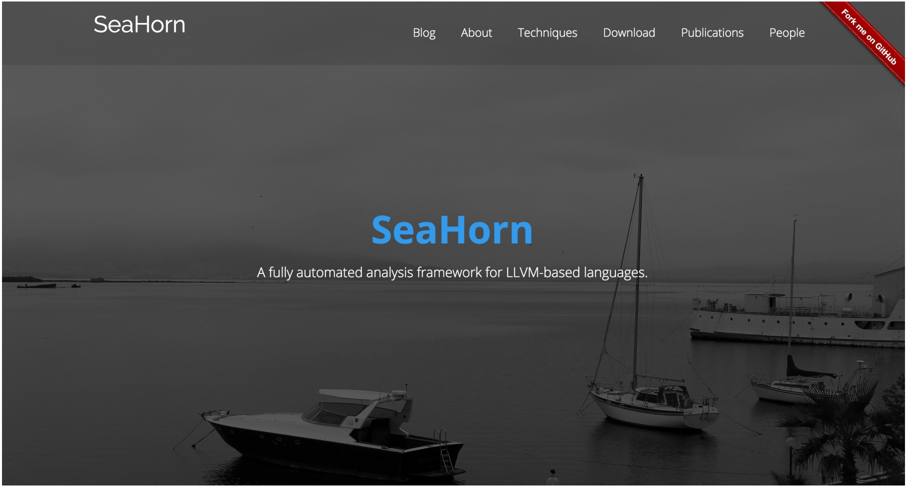
Useful to researchers in verification

- 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

http://seahorn.github.io



SeaHorn

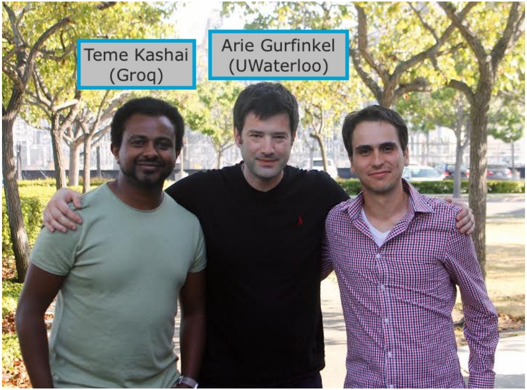
Blog About Techniques Download Publications People

Fork me on GitHub

SeaHorn

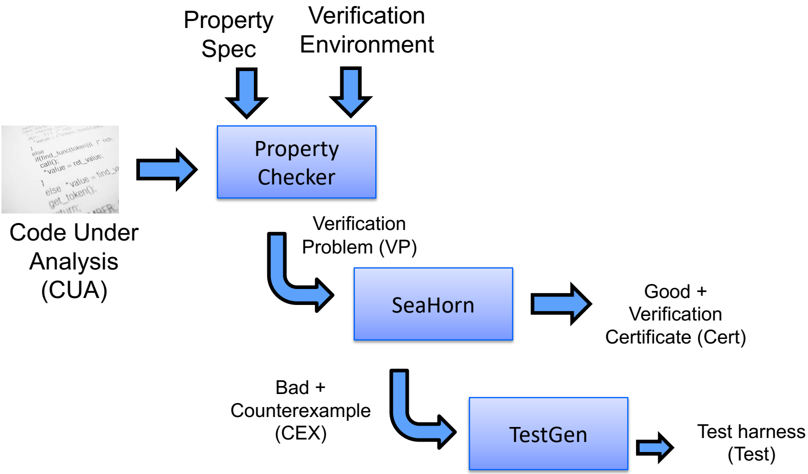
A fully automated analysis framework for LLVM-based languages.

SeaHorn Team



And many great collaborators such as Bjorner, Gange, Komuravelli, Sondergaard, Stuckey, etc

SeaHorn Workflow



Property Spec

Verification Environment

Code Under Analysis (CUA)

Property Checker

Verification Problem (VP)

SeaHorn

Good + Verification Certificate (Cert)

Bad + Counterexample (CEX)

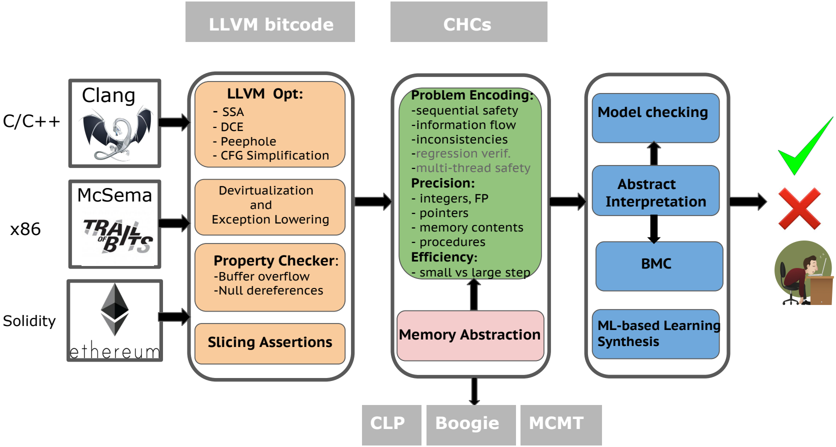
TestGen

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)
    - ongoing work to improve scalability and usability
  - taint analysis (developed by Princeton)

SeaHorn Architecture



Demo

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:

$$\forall \mathcal{V} \cdot (\underbrace{\underbrace{\phi}_{\textit{constraint}} \wedge \overbrace{\overbrace{p_1(X_1)}^{\textit{linear}} \wedge \cdots \wedge p_k(X_k)}^{\textit{non-linear}}}_{\textit{body}} \rightarrow \underbrace{h(X)}_{\textit{head}}), \text{ for } k \geq 0$$

$\mathcal{F}$: function symbols, $\mathcal{P}$:predicate symbols, and $\mathcal{V}$: variables

ϕ is a constraint over $\mathcal{F}$ and $\mathcal{V}$ wrt some background theory

$X_i, X \subseteq \mathcal{V}$ are (possibly empty) vectors of variables

$p_i(X_i)$ is an application $p(t_1, \ldots, t_n)$ of an n-ary $p \in \mathcal{P}$ for FO terms t_i constructed from $\mathcal{F}$ and X_i

$h(X)$ is either defined analogously to p_i or false

Satisfiability of CHCs

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

CHC for Verification

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

$$\begin{array}{l} \mathit{Init}(X) \rightarrow \mathit{Inv}(X) \\ \mathit{Inv}(X) \wedge \mathit{Step}(X, X') \rightarrow \mathit{Inv}(X') \\ \mathit{Inv}(X) \rightarrow \neg\, \mathit{Bad}(X) \end{array}$$

Verification of Multi-Threaded Programs

Predicate Abstraction and Refinement for Verifying Multi-Threaded Programs

Ashutosh Gupta Corneliu Popeea Andrey Rybalchenko

$$
\begin{array}{l}
\bigwedge_{i \in \{1,\ldots,N\}} (Init(X) \rightarrow Inv_i(X)) \\
\bigwedge_{i \in \{1,\ldots,N\}} (Inv_i(X) \wedge Step_i(X, X') \rightarrow Inv_i(X')) \\
\bigwedge_{i \in \{1,\ldots,N\}} (Inv_i(X) \wedge Env_i(X, X') \rightarrow Inv_i(X')) \\
\bigwedge_{i,j \in \{1,\ldots,N\}, i \neq j} (Inv_j(X) \wedge Step_j(X, X') \rightarrow Env_i(X, X')) \\
Inv_1(X) \wedge \ldots \wedge Inv_N(X) \rightarrow \neg\, Bad(X)
\end{array}
$$

Verification of Array Manipulating Programs

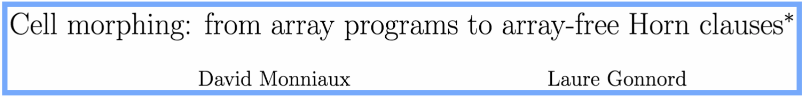
$$\begin{array}{l} \mathit{Init}(X, A) \rightarrow \mathit{Inv}(X, A) \\ I(X, A) \wedge \mathit{Step}(X, A, X', A') \rightarrow I(X', A') \\ \mathit{Inv}(X, A) \rightarrow \neg\ \mathit{Bad}(X, A) \end{array}$$

Step can contain array constraints of the form:

$$\begin{array}{l} a' = \mathtt{write}(a, i, v) \\ v = \mathtt{read}(a, i) \end{array}$$

where $i, v \in X \cup X'$, $a \in A$, 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. $a[k] = a_k$

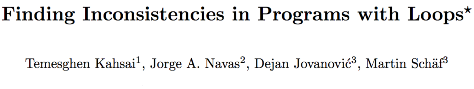
$a' = \texttt{write}(a, i, v)$ (“the value at i is v, the rest unchanged”):

$$i = k \wedge Inv(X, v, i, a_k) \rightarrow Inv(X, v, i, v)$$
$$i \neq k \wedge Inv(X, v, k, a_k) \rightarrow Inv(X, v, k, a_k)$$

$v = \texttt{read}(a, i)$ (”v has new value, the rest is preserved”):

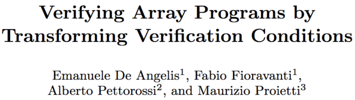
$$i = k \wedge Inv(X, v, i, a_i) \rightarrow Inv(X, a_i, i, a_i)$$
$$i \neq k \wedge \underline{Inv(X, v, k, a_k)} \wedge Inv(X, v, i, a_i) \rightarrow Inv(X, a_i, k, a_k)$$

And many more ...



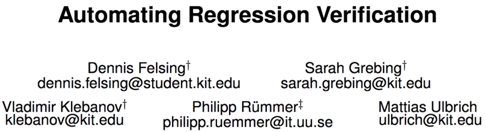
Finding Inconsistencies in Programs with Loops*

Temesghen Kahsai[1], Jorge A. Navas[2], Dejan Jovanović[3], Martin Schäf[3]



Verifying Array Programs by Transforming Verification Conditions

Emanuele De Angelis[1], Fabio Fioravanti[1],
Alberto Pettorossi[2], and Maurizio Proietti[3]



Automating Regression Verification

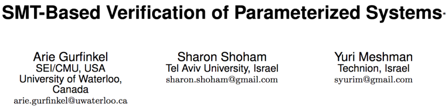
Dennis Felsing†
dennis.felsing@student.kit.edu

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@uwaterloo.ca

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

Yuri Meshman
Technion, Israel
syurim@gmail.com

Horn Clauses for Communicating Timed Systems

Hossein Hojjat
Cornell University, USA

Philipp Rümmer, Pavle Subotic, Wang Yi
Uppsala University, Sweden

From Programs to CHCs

A Hoare triple $\{Pre\}P\{Post\}$ is valid iff every terminating execution of P that starts in a state satisfying Pre ends in a state satisfying $Post$

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

$$\{Pre\}P\{Post\} \iff Pre \Rightarrow \mathrm{wlp}(P, Post)$$

Translating to CHCs Using Weakest Liberal Preconditions

$$\boxed{Pre \rightarrow \mathrm{wlp}(Main, Post) \wedge \bigwedge_{f \in P} \forall x, r.\mathrm{wlp}(B_f, \mathcal{S}_f(x, r))}$$

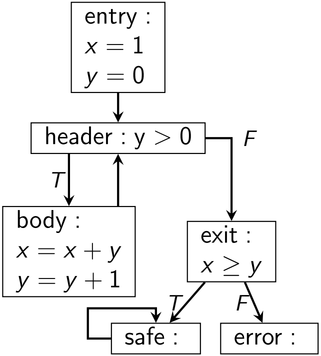
$$\begin{array}{ll}
\mathrm{wlp}(\texttt{if } C\ S_1 \texttt{ else } S_2, \phi) & \rightsquigarrow C \rightarrow \mathrm{wlp}(S_1, \phi) \wedge \neg\ C \rightarrow \mathrm{wlp}(S_2, \phi) \\
\mathrm{wlp}(S_1; S_2, \phi) & \rightsquigarrow \mathrm{wlp}(S_1, \mathrm{wlp}(S_2, \phi)) \\
\mathrm{wlp}(x = e, \phi) & \rightsquigarrow \phi[x \leftarrow e] \\
\mathrm{wlp}(\texttt{error}, \phi) & \rightsquigarrow \bot \\
\mathrm{wlp}(\texttt{while } C\ B, \phi) & \rightsquigarrow \mathcal{I}(\overline{x}) \wedge \\
 & \quad \forall \overline{x}((\mathcal{I}(\overline{x}) \wedge C \wedge \rightarrow \mathrm{wlp}(B, \mathcal{I}(\overline{x}))) \wedge \\
 & \qquad (\mathcal{I}(\overline{x}) \wedge \neg\ C \rightarrow \phi)) \\
\mathrm{wlp}(x = f(y), \phi) & \rightsquigarrow \forall r.\mathcal{S}_f(y, r) \rightarrow \phi[x \leftarrow r]
\end{array}$$

And apply negation, prenex, and conjunctive normal form

Translating to CHCs Using Dual WLP

```
$main()$ {
    $x = 1;$
    $y = 0;$
    while ($y > 0$) {
        $x = x + y;$
        $y = y + 1;$
    }
    assert($x \geq y$)
}
```

Translating to CHCs Using Dual WLP



entry :
$x = 1$
$y = 0$

header : y > 0

body :
$x = x + y$
$y = y + 1$

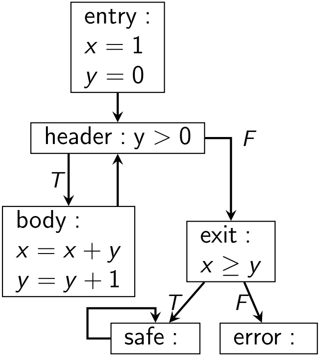
exit :
$x \geq y$

safe :

error :

Translating to CHCs Using Dual WLP

$$\boxed{\mathrm{wlp}(P, Post) = \neg\, \mathrm{wlp}(P, \neg\, Post)}$$



entry :
$x = 1$
$y = 0$

header : $y > 0$

body :
$x = x + y$
$y = y + 1$

exit :
$x \geq y$

safe :

error :

$$\begin{aligned}
\mathrm{entry}(x, y) &\leftarrow \mathit{true}.\\
\mathrm{h}(x, y) &\leftarrow \mathrm{entry}(x, y), x = 1,\ y = 0.\\
\mathrm{b}(x, y) &\leftarrow \mathrm{h}(x, y), y > 0.\\
\mathrm{h}(x', y') &\leftarrow \mathrm{b}(x, y),\\
&\quad\ x' = x + y, y' = y + 1.\\
\mathrm{exit}(x, y) &\leftarrow \mathrm{h}(x, y), y \leq 0.\\
\mathrm{error}(x, y) &\leftarrow \mathrm{exit}(x, y), x < y.
\end{aligned}$$

Efficient Encoding of Programs in SeaHorn

Rule for `if-then-else` can cause the resulting CHCs to be exponentially larger than the original program

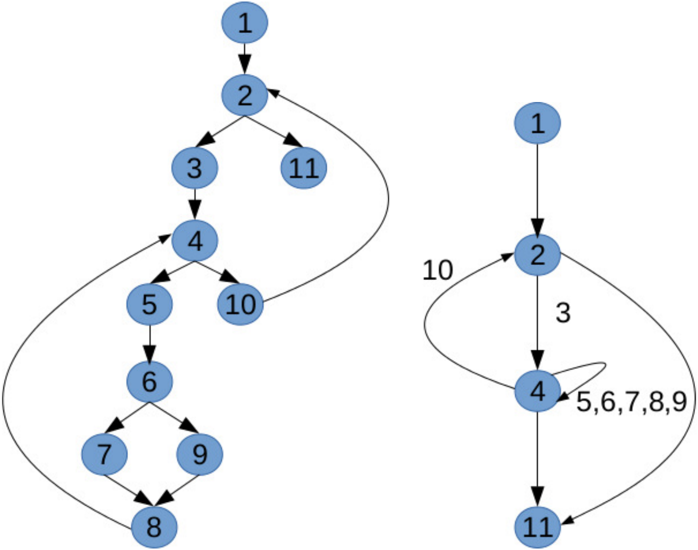
Solution: generate compact VCs for loop-free code

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



Large-Step Encoding using Cut-point Graphs

Single Static Assignment (SSA): every value has a unique definition

```
**int** $x$, $y$, $n$;
$\ell_0$: $x = 0$; $y = *$;
$\ell_1$: **while** $(x < n)$ {
$\ell_2$:     **if** $(y > 0)$
$\ell_3$:         $x = x + y$;
              **else**
$\ell_4$:         $x = x - y$;
$\ell_5$:     $y = -1 \times y$ ;
}
$\ell_6$:
```

```
$\ell_0$:    **goto** $\ell_1$
$\ell_1$:    $x_0 = \phi(0 : \ell_0, x_3 : \ell_5)$;
             $y_0 = \phi(y : \ell_0, y_1 : \ell_5)$;
             **if** $(x_0 < n)$ **goto** $\ell_2$ **else goto** $\ell_6$
$\ell_2$:    **if** $(y_0 > 0)$ **goto** $\ell_3$ **else goto** $\ell_4$
$\ell_3$:    $x_1 = x_0 + y_0$; **goto** $\ell_5$
$\ell_4$:    $x_2 = x_0 - y_0$; **goto** $\ell_5$
$\ell_5$:    $x_3 = \phi(x_1 : \ell_3, x_2 : \ell_4)$;
             $y_1 = -1 \times y_0$
             **goto** $\ell_1$
$\ell_6$:
```

Large-Step Encoding using Cut-point Graphs

ϕ :

$$
\begin{aligned}
& x_1 = x_0 + y_0 \wedge \\
& x_2 = x_0 - y_0 \wedge \\
& y_1 = -1 \times y_0 \wedge \\
& B_2 \rightarrow x_0 < n \wedge \\
& B_3 \rightarrow B_2 \wedge y_0 > 0 \wedge \\
& B_4 \rightarrow B_2 \wedge y_0 \leq 0 \wedge \\
& B_5 \rightarrow ((B_3 \wedge x_3 = x_1) \vee \\
& \qquad\quad (B_4 \wedge x_3 = x_2)) \wedge \\
& B_5 \wedge x_0' = x_3 \wedge y_0' = y_1
\end{aligned}
$$

```
ℓ0: ~~goto ℓ1~~
ℓ1: $x_0 = \phi(0 : \ell_0, x_3 : \ell_5)$;
    $y_0 = \phi(y : \ell_0, y_1 : \ell_5)$;
    if ($x_0 < n$) goto ℓ2 else goto ℓ6
ℓ2: if ($y_0 > 0$) goto ℓ3 else goto ℓ4
ℓ3: $x_1 = x_0 + y_0$; goto ℓ5
ℓ4: $x_2 = x_0 - y_0$; goto ℓ5
ℓ5: $x_3 = \phi(x_1 : \ell_3, x_2 : \ell_4)$;
    $y_1 = -1 \times y_0$
    ~~goto ℓ1~~
ℓ6:
```

$$\boxed{p_1(x_0', y_0') \leftarrow p_1(x_0, y_0) \wedge \phi}$$

SeaHorn Memory Model $\equiv$ C Memory Model

Block-based memory model: a pointer is a pair $\langle ref, o \rangle$ where *ref* uniquely defines a memory object and *o* defines the byte in the object being point to

$$\boxed{Env : \mathbb{V} \rightarrow Ptr \qquad \mathrm{Ptr} = \mathrm{Ref} \times \mathrm{Int} \qquad Mem : \mathrm{Ptr} \rightarrow \mathrm{Ptr}}$$

Concrete memory model:

- each allocation (e.g. **malloc**) creates a fresh new object
- the number of objects is infinite

Abstract memory model:

- the number of allocation regions is finite
  - allocation site used as an object reference

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

From Pointer Analysis to CHCs

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

  $$\begin{array}{rl} \texttt{v := *(\&p + i)} \mapsto & v = \texttt{read}(A_o, i) \\ \texttt{*(\&p + i) := v} \mapsto & A'_o = \texttt{write}(A_o, i, v) \end{array}$$

- 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:

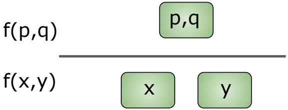
- resolve aliasing at encoding time

CHCs Using a Context-Insensitive Pointer Analysis

```
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

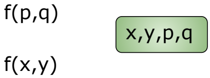


CHCs Using a Context-Insensitive Pointer Analysis

```
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)

x,y,p,q

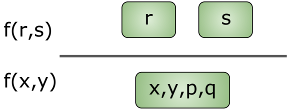
f(x,y)

CHCs Using a Context-Insensitive Pointer Analysis

```
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

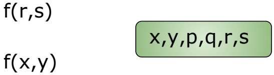


CHCs Using a Context-Insensitive Pointer Analysis

```
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



CHCs Using a Context-Insensitive Pointer Analysis

```
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);
}
```

$$
\begin{array}{l}
S_f(x, y, a_{xy}, a''_{xy}) \leftarrow \\
\quad a'_{xy} = \mathtt{write}(a_{xy}, x, 1) \land \\
\quad a''_{xy} = \mathtt{write}(a'_{xy}, y, 2)
\end{array}
$$

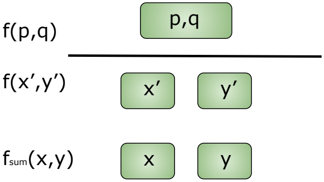
$$
\begin{array}{l}
S_g(p, q, r, s, a_{pqrs}, a''_{pqrs}) \leftarrow \\
\quad S_f(p, q, a_{pqrs}, a'_{pqrs}) \land \\
\quad S_f(r, s, a'_{pqrs}, a''_{pqrs})
\end{array}
$$

Sound CHCs Using a Context-Sensitive Pointer Analysis

```
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

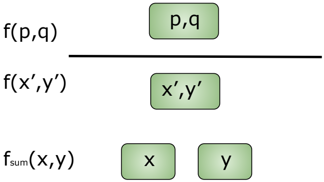


Sound CHCs Using a Context-Sensitive Pointer Analysis

```
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

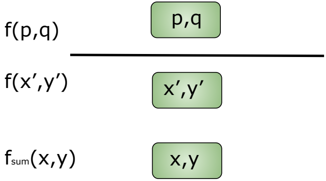


Sound CHCs Using a Context-Sensitive Pointer Analysis

```
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

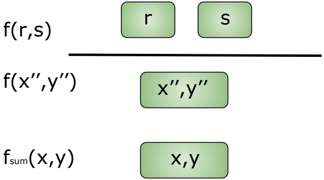


Sound CHCs Using a Context-Sensitive Pointer Analysis

```
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

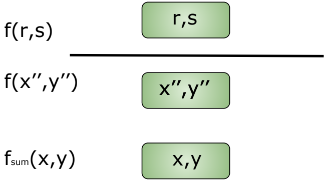


Sound CHCs Using a Context-Sensitive Pointer Analysis

```
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

Sound CHCs Using a Context-Sensitive Pointer Analysis

```
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);
}
```

$$
\begin{array}{l}
S_f(x, y, a_{xy}, a''_{xy}) \leftarrow \\
\quad a'_{xy} = \mathtt{write}(a_{xy}, x, 1) \land \\
\quad a''_{xy} = \mathtt{write}(a'_{xy}, y, 2)
\end{array}
$$

$$
\begin{array}{l}
S_g(p, q, r, s, a_{pq}, a_{rs}, a'_{pq}, a'_{rs}) \leftarrow \\
\quad S_f(p, q, a_{pq}, a'_{pq}) \land \\
\quad S_f(r, s, a_{rs}, a'_{rs})
\end{array}
$$

Sound CHCs Using a Context-Sensitive Pointer Analysis

```
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);
}
```

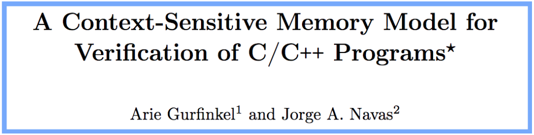
$$\begin{array}{l} S_f(x, y, a_{xy}, a''_{xy}) \leftarrow \\ \quad a'_{xy} = \mathtt{write}(a_{xy}, x, 1) \wedge \\ \quad a''_{xy} = \mathtt{write}(a'_{xy}, y, 2) \end{array}$$

$$\begin{array}{l} S_g(p, q, r, s, a_{pq}, a_{rs}, a'_{pq}, a'_{rs}) \leftarrow \\ \quad S_f(p, q, a_{pq}, a'_{pq}) \wedge \\ \quad S_f(r, s, a_{rs}, a'_{rs}) \end{array}$$

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 Gurfinkel[1] and Jorge A. Navas[2]

- 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

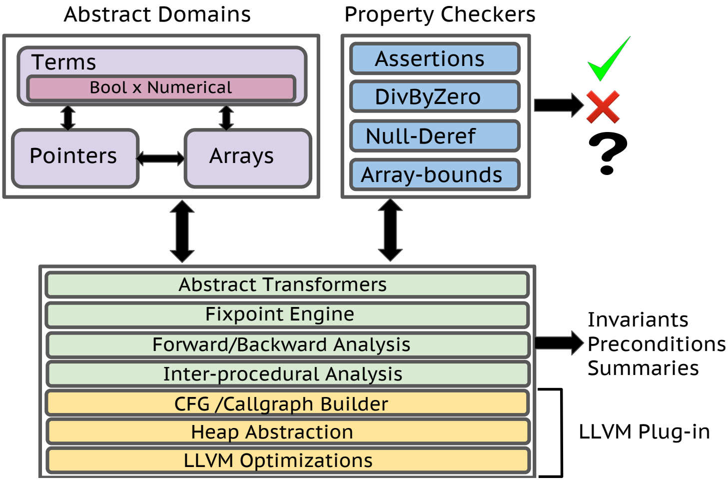
Crab: an Abstract Interpretation Library

- 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: $5 \leq x \leq 10 \wedge x \bmod 2 = 0$
  - zones: $x - y \leq k$
  - wrapped intervals: intervals on machine-arithmetic integers
  - non-convex:
    - DisIntervals: $x \leq -1 \vee x \geq 1$
    - boxes: boolean combination of intervals
- Symbolic domains
  - terms: numerical domains + uninterpreted functions
    - $x \leq 10 \wedge y = f(\ldots) \wedge z = f(\ldots) \rightarrow x \leq 10 \wedge y = z$
    - $b = \texttt{write}(a, i, x) \wedge y = \texttt{read}(a, i) \rightarrow x = y$
- 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

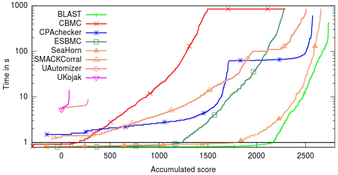
Integration with other tools and other solvers

- 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

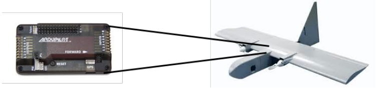
SeaHorn in Real World

SV-COMP

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

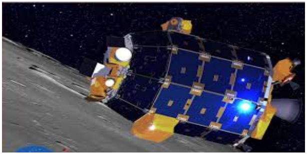


Autopilot code (absence of buffer overflows)



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!

SeaHorn Software

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`

SeaHorn Publications

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**

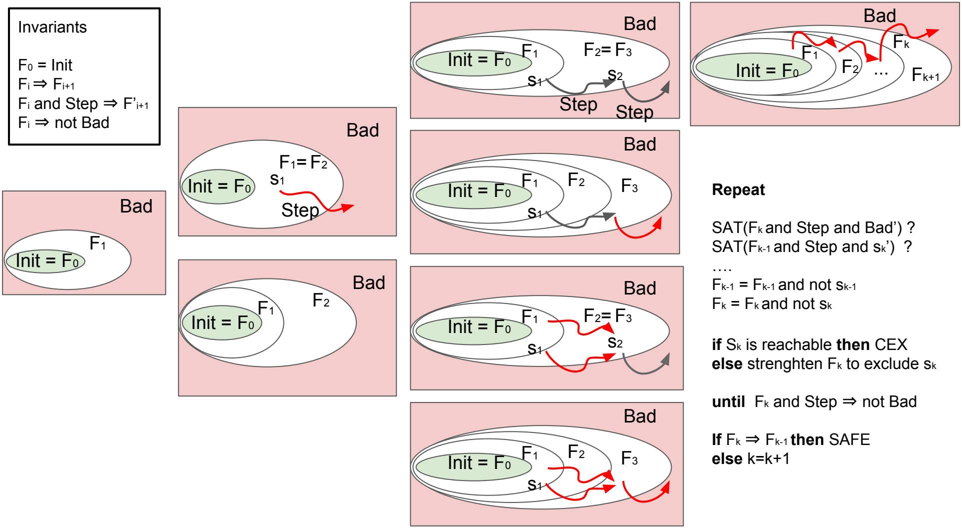
SMT-Based Model Checking for Recursive Programs. **CAV 2014**

Extra Material

IC3/PDR in One Slide

Invariants

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



Repeat

SAT(F_k and Step and Bad') ?
SAT(F_{k-1} and Step and s_k') ?
....
$F_{k-1} = F_{k-1}$ and not s_{k-1}
$F_k = F_k$ and not s_k

if S_k is reachable **then** CEX
else strenghten F_k to exclude s_k

until F_k and Step $\Rightarrow$ 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 $F_k \wedge Step \wedge Bad'$ 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 $F_{k-1} \wedge Step \wedge s_{k'}$ is sat
4. If none exists, then $F_k = F_k \wedge \neg\ s_k$ 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

Solving CHCs using IC3/PDR

Generalize predecessors: $F_{k-1} \wedge Step \wedge s'_k$

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

Block more than one state

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

$F_{k-1} \wedge Step \Rightarrow \neg\, s$ iff $\neg\, s \wedge F_{k-1} \wedge Step \Rightarrow \neg\, s$

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

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

If one exists $F_k = F_k \wedge c$

Moreover, for every $i \leq k$ $F_i = F_i \wedge c$ 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 $F_k \wedge c \wedge Step \Rightarrow c'$ then

$F_{k+1} = F_{k+1} \wedge c$ (for all $1 \leq k \leq N-1$)