Building Program Verifiers from Compilers and Theorem Provers

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Arie Gurfinkel

based on joint work with Teme Kahsai, Jorge A. Navas, Anvesh Komuravelli, and Nikolaj Bjorner
Copyright 2015 Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This material has been approved for public release and unlimited distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

DM-0002433
Automated Software Analysis

Program \rightarrow \text{Automated Analysis} \rightarrow \text{Correct} \rightarrow \text{Incorrect}

- Software Model Checking with Predicate Abstraction
  e.g., Microsoft’s SDV

- Abstract Interpretation with Numeric Abstraction
  e.g., ASTREE, Polyspace
Turing, 1936: “undecidable”
How can one check a routine in the sense of making sure that it is right? The programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.
SeaHorn

A fully automated verification framework for LLVM-based languages.

http://seahorn.github.io
SeaHorn Verification Framework

**Distinguishing Features**
- LLVM front-end(s)
- Constrained Horn Clauses to represent Verification Conditions
- Comparable to state-of-the-art tools at SV-COMP’15

**Goals**
- be a state-of-the-art Software Model Checker
- be a framework for experimenting and developing CHC-based verification
Related Tools

CPAChecker
- Custom front-end for C
- Abstract Interpretation-inspired verification engine
- Predicate abstraction, invariant generation, BMC, k-induction

SMACK / Corral
- LLVM-based front-end
- Reduces C verification to Boogie
- Corral / Q verification back-end based on Bounded Model Checking with SMT

UFO
- LLVM-based front-end (partially reused in SeaHorn)
- Combines Abstract Interpretation with Interpolation-Based Model Checking
- (no longer actively developed)
SeaHorn Philosophy

Build a state-of-the-art Software Model Checker
  • useful to “average” users
    – user-friendly, efficient, trusted, certificate-producing, …
  • useful to researchers in verification
    – modular design, clean separation between syntax, semantics, and logic, …

Stand on the shoulders of giants
  • reuse techniques from compiler community to reduce verification effort
    – SSA, loop restructuring, induction variables, alias analysis, …
    – static analysis and abstract interpretation
  • reduce verification to logic
    – verification condition generation
    – Constrained Horn Clauses

Build reusable logic-based verification technology
  • “SMT-LIB” for program verification
SeaHorn Usage

> sea pf FILE.c

Outputs sat for unsafe (has counterexample); unsat for safe

Additional options

- `--cex=trace.xml` outputs a counter-example in SV-COMP’15 format
- `--track={reg,ptr,mem}` track registers, pointers, memory content
- `--step={large,small}` verification condition step-semantics
  - `small` == basic block, `large` == loop-free control flow block
- `--inline` inline all functions in the front-end passes

Additional commands

- `sea smt` -- generates CHC in extension of SMT-LIB2 format
- `sea clp` -- generates CHC in CLP format (under development)
- `sea lfe-smt` -- generates CHC in SMT-LIB2 format using legacy front-end
Verification Pipeline

front-end

clang | pp | ms | opt | horn

compile | pre-process | mixed semantics | optimize | VC gen & solve
Constrained Horn Clauses
INTERMEDIATE REPRESENTATION
Constrained Horn Clauses (CHC)

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

$$\forall V \cdot (\bar{\bar{A}} \bar{\bar{\exists}} \ p_1[X_1] \bar{\bar{\exists}} \ldots \bar{\bar{\exists}} \ p_n[X_n] \rightarrow h[X]),$$

where

- $A$ is a background theory (e.g., Linear Arithmetic, Arrays, Bit-Vectors, or combinations of the above)
- $\bar{\bar{A}}$ is a constrained in the background theory $A$
- $p_1, \ldots, p_n, h$ are $n$-ary predicates
- $p_i[X]$ is an application of a predicate to first-order terms
Example Horn Encoding

\[
\begin{array}{l}
\text{int } x = 1; \\
\text{int } y = 0; \\
\text{while } (* ) \{ \\
\quad x = x + y; \\
\quad y = y + 1; \\
\}
\text{assert}(x \geq y);
\end{array}
\]

\[
\begin{array}{l}
l_0: \\
x = 1 \\
y = 0
\end{array}
\]

\[
l_1: b_1 = \text{nondet()}
\]

\[
l_2: \\
x = x + y \\
y = y + 1
\]

\[
l_3: \\
b_2 = x \geq y
\]

\[
l_4: \\

\]

\[
l_{err}: \\

\]

\[
\langle 1 \rangle \quad p_0.
\]

\[
\langle 2 \rangle \quad p_1(x, y) \leftarrow \\
p_0, x = 1, y = 0.
\]

\[
\langle 3 \rangle \quad p_2(x, y) \leftarrow p_1(x, y).
\]

\[
\langle 4 \rangle \quad p_3(x, y) \leftarrow p_1(x, y).
\]

\[
\langle 5 \rangle \quad p_1(x', y') \leftarrow \\
p_2(x, y), \\
x' = x + y, \\
y' = y + 1.
\]

\[
\langle 6 \rangle \quad p_4 \leftarrow (x \geq y), p_3(x, y).
\]

\[
\langle 7 \rangle \quad p_{err} \leftarrow (x < y), p_3(x, y).
\]

\[
\langle 8 \rangle \quad p_4 \leftarrow p_4.
\]

\[
\langle 9 \rangle \quad \bot \leftarrow p_{err}.
\]
CHC Terminology

Rule

\[ h[X] \overline{\land} p_1[X_1], \ldots, p_n[X_n], \overline{\land}. \]

Query

\[ \text{false} \overline{\land} p_1[X_1], \ldots, p_n[X_n], \overline{\land}. \]

Fact

\[ h[X] \overline{\land} \overline{\land}. \]

Linear CHC

\[ h[X] \overline{\land} p[X_1], \overline{\land}. \]

Non-Linear CHC

\[ h[X] \overline{\land} p_1[X_1], \ldots, p_n[X_n], \overline{\land}. \]

for \( n > 1 \)
CHC Satisfiability

A model of a set of clauses \( \models \) is an interpretation of each predicate \( p_i \) that makes all clauses in \( \models \) valid.

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

A model is **A-definable**, if each \( p_i \) is definable by a formula \( \tilde{A}_i \) in A.
Relationship between CHC and Verification

A program satisfies a property iff corresponding CHCs are satisfiable
• satisfiability-preserving transformations == safety preserving

Models for CHC correspond to verification certificates
• inductive invariants and procedure summaries

Unsatisfiability (or derivation of FALSE) corresponds to counterexample
• the resolution derivation (a path or a tree) is the counterexample

CAVEAT: In SeaHorn the terminology is reversed
• SAT means there exists a counterexample – a BMC at some depth is SAT
• UNSAT means the program is safe – BMC at all depths are UNSAT
FROM PROGRAMS TO CLAUSES
Hoare Triples

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

**Inductive Loop Invariant**

\[
\begin{align*}
\text{Pre} & \land \text{Inv} & \{\text{Inv} \land \text{C}\} \text{ Body} \{\text{Inv}\} & \text{Inv} \land \text{C} \land \text{Post} \\
\{\text{Pre}\} \textbf{while} \ C \ \textbf{do} \ \text{Body} \ {\text{Post}}
\end{align*}
\]

**Function Application**

\[
\begin{align*}
(\text{Pre} \land \text{Ep} = a) & \land \ P & \{P\} \ \text{Body}_f \ {Q} & \ (Q \land \text{Ep}, r = a, b) \land \text{Post} \\
\{\text{Pre}\} b = F(a) & \{\text{Post}\}
\end{align*}
\]

**Recursion**

\[
\begin{align*}
\{\text{Pre}\} b = F(a) & \{\text{Post}\} \ \land \ {\text{Pre}} \ \text{Body}_f \ {\text{Post}}
\end{align*}
\]

\[
\{\text{Pre}\} b = F(a) \ {\text{Post}}
\]
Weakest Liberal Pre-Condition

Validity of Hoare triples is reduced to FOL validity by applying a predicate transformer

Dijkstra’s weakest liberal pre-condition calculus [Dijkstra’75]

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

weakest pre-condition ensuring that executing P ends in Post

\[ \{\text{Pre}\} \ P \ \{\text{Post}\} \text{ is valid} , \text{ Pre } \) wlp (P, Post) \]
Horn Clauses by Weakest Liberal Precondition

Prog = def Main(x) { body_M }, ..., def P(x) { body_P }

\[ \text{wlp } (x=E, Q) = \text{let } x=E \text{ in } Q \]
\[ \text{wlp } (\text{assert } (E), Q) = E \land Q \]
\[ \text{wlp } (\text{assume}(E), Q) = E \rightarrow Q \]
\[ \text{wlp } (\text{while } E \text{ do } S, Q) = I(w) \land \]
\[ 8w . ((I(w) \land E) \rightarrow \text{wlp } (S, I(w))) \land ((I(w) \land :E) \rightarrow Q)) \]
\[ \text{wlp } (y = P(E), Q) = p_{pre}(E) \land (8 r. p(E, r) \rightarrow Q[r/y]) \]

ToHorn (def P(x) {S}) = wlp (x0=x ; assume (p_{pre}(x)); S, p(x0, ret))
ToHorn (Prog) = wlp (Main(), true) \land 8\{P 2 Prog\} . ToHorn (P)
Example of a WLP Horn Encoding

{Pre: $y \geq 0$}
\begin{align*}
    x_o &= x; \\
    y_o &= y; \\
    \text{while } y > 0 \text{ do} & \\
    x &= x + 1; \\
    y &= y - 1;
\end{align*}
{Post: $x = x_o + y_o$}

ToHorn

C1: $I(x, y, x, y) \land y \geq 0$.
C2: $I(x+1, y-1, x_o, y_o) \land I(x, y, x_o, y_o), y > 0$.
C3: false $\land I(x, y, x_o, y_o), y \cdot 0, x \neq x_o + y_o$

$\{y \geq 0\} \implies \{x = x_{old} + y_{old}\}$ is true iff the query $C_3$ is satisfiable.
Dual WLP

Dual weakest liberal pre-condition

\[ \text{dual-wlp} (P, \text{Post}) = :\text{wlp} (P, :\text{Post}) \]

\[ s^2 \text{dual-wlp} (P, \text{Post}) \text{ iff there exists an execution of } P \text{ that starts in } s \text{ and ends in } \text{Post} \]

\text{dual-wlp} (P, \text{Post}) \text{ is the weakest condition ensuring that an execution of } P \text{ can reach a state in } \text{Post} 
Horn Clauses by Dual WLP

Assumptions

• each procedure is represented by a control flow graph
  – i.e., statements of the form \( l_i:S; \text{goto} \ l_j \), where \( S \) is loop-free
• program is unsafe iff the last statement of Main() is reachable
  – i.e., no explicit assertions. All assertions are top-level.

For each procedure \( P(x) \), create predicates

• \( l(w) \) for each label, \( p_{en}(x_0,x,w) \) for entry, \( p_{ex}(x_0,r) \) for exit

The verification condition is a conjunction of clauses:

\[
\begin{align*}
p_{en}(x_0,x) & \land x_0 = x \\
l_i(x_0,w') & \land l_j(x_0,w) \land \forall \ l_i: S; \text{goto} \ l_j \\
p(x_0,r) & \land p_{ex}(x_0,r) \\
false & \land \text{Main}_{ex}(x, \text{ret})
\end{align*}
\]
Example Horn Encoding

```plaintext
int x = 1;
int y = 0;
while (*) {
    x = x + y;
    y = y + 1;
}
assert(x ≥ y);
```

```
\begin{align*}
\langle 1 \rangle & \quad p_0. \\
\langle 2 \rangle & \quad p_1(x, y) \leftarrow p_0, x = 1, y = 0. \\
\langle 3 \rangle & \quad p_2(x, y) \leftarrow p_1(x, y). \\
\langle 4 \rangle & \quad p_3(x, y) \leftarrow p_1(x, y). \\
\langle 5 \rangle & \quad p_1(x', y') \leftarrow p_2(x, y), \\
& \quad x' = x + y, \\
& \quad y' = y + 1. \\
\langle 6 \rangle & \quad p_4 \leftarrow (x ≥ y), p_3(x, y). \\
\langle 7 \rangle & \quad p_{err} \leftarrow (x < y), p_3(x, y). \\
\langle 8 \rangle & \quad p_4 \leftarrow p_4. \\
\langle 9 \rangle & \quad \bot \leftarrow p_{err}.
\end{align*}
```
**Single Static Assignment**

SSA == every value has a unique assignment (a \textit{definition})
A procedure is in SSA form if every variable has exactly one definition

SSA form is used by many compilers
- explicit def-use chains
- simplifies optimizations and improves analyses

PHI-function are necessary to maintain unique definitions in branching control flow

\[ x = \text{PHI} ( v_0 : \text{bb}_0, \ldots, v_n : \text{bb}_n ) \]  
\text{(phi-assignment)}

“x gets \( v_i \) if previously executed block was \( \text{bb}_i \)”
Large Step Encoding: Single Static Assignment

```c
int x, y, n;
x = 0;
while (x < N) {
    if (y > 0)
        x = x + y;
    else
        x = x - y;
y = -1 * y;
}
```

0: goto 1
1: x_0 = PHI(0:0, x_3:5);
y_0 = PHI(y:0, y_1:5);
   if (x_0 < N) goto 2 else goto 6
2: if (y_0 > 0) goto 3 else goto 4
3: x_1 = x_0 + y_0; goto 5
4: x_2 = x_0 - y_0; goto 5
5: x_3 = PHI(x_1:3, x_2:4);
y_1 = -1 * y_0;
goto 1
6:
Example: Single Static Assignment

```c
int x, y, n;
x = 0;
while (x < N) {
    if (y > 0)
        x = x + y;
    else
        x = x - y;
y = -1 * y;
}
```

```
0: goto 1
1: \( x_0 = PHI(0:0, x_3:5); \)
    \( y_0 = PHI(y:0, y_1:5); \)
    if (x_0 < N) goto 2 else goto 6
2: if (y_0 > 0) goto 3 else goto 4
3: x_1 = x_0 + y_0; goto 5
4: x_2 = x_0 - y_0; goto 5
5: x_3 = PHI(x_1:3, x_2:4);
    y_1 = -1 * y_0;
    goto 1
6:
```
Example: Large Step Encoding

0: goto 1

1: x_0 = PHI(0:0, x_3:5);
   y_0 = PHI(y:0, y_1:5);
   if (x_0 < N) goto 2 else goto 6

2: if (y_0 > 0) goto 3 else goto 4

3: x_1 = x_0 + y_0; goto 5

4: x_2 = x_0 - y_0; goto 5

5: x_3 = PHI(x_1:3, x_2:4);
   y_1 = -1 * y_0;
   goto 1

6:
Example: Large Step Encoding

\[ x_1 = x_0 + y_0 \]
\[ x_2 = x_0 - y_0 \]
\[ y_1 = -1 \times y_0 \]

1: \( \text{phi}(0:0, x_3:5); \)
\[ y_0 = \text{phi}(y:0, y_1:5); \]
if (\( x_0 < N \)) goto 2 else goto 6

2: if (\( y_0 > 0 \)) goto 3 else goto 4

3: \( x_1 = x_0 + y_0 \) goto 5

4: \( x_2 = x_0 - y_0 \) goto 5

5: \( \text{phi}(x_1:3, x_2:4); \)
\[ y_1 = -1 \times y_0; \]
goto 1
Example: Large Step Encoding

\[ x_1 = x_0 + y_0 \]
\[ x_2 = x_0 - y_0 \]
\[ y_1 = -1 \times y_0 \]

\[ B_2 \rightarrow x_0 < N \]
\[ B_3 \rightarrow B_2 \land y_0 > 0 \]
\[ B_4 \rightarrow B_2 \land y_0 \leq 0 \]
\[ B_5 \rightarrow (B_3 \land x_3=x_1) \lor (B_4 \land x_3=x_2) \]
\[ B_5 \land x'_0=x_3 \land y'_0=y_1 \]

1: \[ x_0 = \text{PHI}(0:0, x_3:5); \]
   \[ y_0 = \text{PHI}(y:0, y_1:5); \]
   \[ \text{if } (x_0 < N) \text{ goto 2 else goto 6} \]

2: \[ \text{if } (y_0 > 0) \text{ goto 3 else goto 4} \]

3: \[ x_1 = x_0 + y_0; \text{ goto 5} \]

4: \[ x_2 = x_0 - y_0; \text{ goto 5} \]

5: \[ x_3 = \text{PHI}(x_1:3, x_2:4); \]
   \[ y_1 = -1 \times y_0; \]
   \[ \text{goto 1} \]

\[ p_1(x'_0, y'_0) \models p_1(x_0, y_0), \Delta. \]
Mixed Semantics

PROGRAM TRANSFORMATION
Deeply nested assertions
Deeply nested assertions

Counter-examples are long
Hard to determine (from main) what is relevant
Mixed Semantics

Stack-free program semantics combining:

• operational (or small-step) semantics
  – i.e., usual execution semantics
• natural (or big-step) semantics: function summary [Sharir-Pnueli 81]
  – \((\frac{3}{4}, \frac{3}{4})\) \text{iff} the execution of \( f \) on input state \( \frac{3}{4} \) terminates and results in state \( \frac{3}{4}' \)
• some execution steps are big, some are small

Non-deterministic executions of function calls

• update top activation record using function summary, or
• enter function body, forgetting history records (i.e., no return!)

Preserves reachability and non-termination

Theorem: Let \( K \) be the operational semantics, \( K^m \) the stack-free semantics, and \( L \) a program location. Then,

\[ K^2 \text{EF (pc=L)} , \ K^m 2 \text{EF (pc=L)} \quad \text{and} \quad K^2 \text{EG (pc\neq L)} , \ K^m 2 \text{EG (pc\neq L)} \]
```python
def main():
    int x = nd();
    x = x + 1;
    while(x >= 0):
        x = f(x);
        if(x < 0):
            Error;
    END;

def f(int y):
    ret y
    if(y >= 10):
        y = y + 1;
        y = f(y);
    elif(y > 0):
        y = y + 1;
    else:
        y = y - 1
```

Summary of $f(y)$:

$1 \cdot x \cdot 9 \\not\leq y' = y$ $\lor$ $\ \ x \cdot 0 \ \not\leq x' = x - 1$ $\lor$ $y' = y + 1$ $\lor$ $y' = f(y)$
Mixed Semantics as Program Transformation

main ()
    p1 (); p1 ();
    assert (c1);
    p1 ()
    p2 ();
    assert (c2);
    p2 ()
    assert (c3);

mixed Semantics

main_new ()
    if (*) goto p1_{entry};
    else p1_{new} ();
    if (*) goto p1_{entry};
    else p1_{new} ();
    if (~c1) goto error;
    assume (false);

p1_{entry} :
    if (*) goto p2_{entry};
    else p2_{new} ();
    if (~c2) goto error;
    p2_{entry} :
    if (~c3) goto error;
    assume (false);

error : assert (false);

p1_{new} ()
    if (*) goto p2_{entry};
    else p2_{new} ();
    assume (c2);

p2_{new} ()
    assume (c3);
Implementing Mixed Semantics in LLVM

Something about how this can be implemented as a simple transformation in LLVM

in the Lab, show how to do this transformation by hand by modifying the bitcode and using opt to execute the optimization
SOLVING CHC WITH SMT
Programs, Cexs, Invariants

A program \( P = (V, \text{Init}, \frac{1}{2}, \text{Bad}) \)

- Notation: \( F(X) = 9 \ u . \ (X \notin \frac{1}{2}) \subseteq \text{Init} \)

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

\[
\text{Init}(v_0) \land \left( \bigwedge_{i=0}^{N-1} \rho(v_i, v_{i+1}) \right) \land \text{Bad}(v_N) \nRightarrow \bot
\]

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

\[
\begin{align*}
\text{Init}(u) & \Rightarrow \text{Inv}(u) \\
\text{Inv}(u) \land \rho(u, v) & \Rightarrow \text{Inv}(v) \\
\text{Inv}(u) & \Rightarrow \neg \text{Bad}(u)
\end{align*}
\]

\[
\{ \text{Inductive}, \text{Safe} \}
\]
### IC3/PDR Algorithm Overview

**Input:** Transition system $T = (\text{Init}, \text{Tr}, \text{Bad})$

```plaintext
1. $F_0 \leftarrow \text{Init}; N \leftarrow 0$
2. repeat
3. \quad $G \leftarrow \text{PdrMkSafe}([F_0, \ldots, F_N], \text{Bad})$
4. \quad if $G = []$ then return UNSAFE;
5. \quad $\forall 0 \leq i \leq N \cdot F_i \leftarrow G[i]$
6. \quad $F_0, \ldots, F_N \leftarrow \text{PdrPush}([F_0, \ldots, F_N])$
7. \quad \text{// $F_0, \ldots, F_N$ is a safe $\delta$-trace}
8. \quad if $\exists 0 \leq i \leq N \cdot F_i = \emptyset$ then return SAFE;
9. \quad $N \leftarrow N + 1; F_N \leftarrow \emptyset$
10. until $\infty$;
```
IC3/PDR in Pictures
IC3/PDR in Pictures
IC3/PDR in Pictures

PDR Invariants
- \( R_i \rightarrow \text{Bad} \)
- \( \text{Init} \rightarrow R_i \)
- \( R_i \rightarrow R_{i+1} \)
- \( R_i \not\in \frac{1}{2} \rightarrow R_{i+1} \)
Spacer: Solving CHC in Z3

Spacer: solver for SMT-constrained Horn Clauses
• stand-alone implementation in a fork of Z3
• [http://bitbucket.org/spacer/code](http://bitbucket.org/spacer/code)

Support for Non-Linear CHC
• model procedure summaries in inter-procedural verification conditions
• model assume-guarantee reasoning
• uses MBP to under-approximate models for finite unfoldings of predicates
• uses MAX-SAT to decide on an unfolding strategy

Supported SMT-Theories
• Best-effort support for arbitrary SMT-theories
  – data-structures, bit-vectors, non-linear arithmetic
• Full support for Linear arithmetic (rational and integer)
• Quantifier-free theory of arrays
  – only quantifier free models with limited applications of array equality
RESULTS
SV-COMP 2015

http://sv-comp.sosy-lab.org/2015/

4th Competition on Software Verification held (here!) at TACAS 2015

Goals
• Provide a snapshot of the state-of-the-art in software verification to the community.
• Increase the visibility and credits that tool developers receive.
• Establish a set of benchmarks for software verification in the community.

Participants:
• Over 22 participants, including most popular Software Model Checkers and Bounded Model Checkers

Benchmarks:
• C programs with error location (programs include pointers, structures, etc.)
• Over 6,000 files, each 2K – 100K LOC
• Linux Device Drivers, Product Lines, Regressions/Tricky examples
• http://sv-comp.sosy-lab.org/2015/benchmarks.php
Results for DeviceDriver category
Conclusion

SeaHorn ([http://seahorn.github.io](http://seahorn.github.io))
- a state-of-the-art Software Model Checker
- LLVM-based front-end
- CHC-based verification engine
- a framework for research in logic-based verification

The future
- making SeaHorn useful to users of verification technology
  - counterexamples, build integration, property specification, proofs, etc.
- targeting many existing CHC engines
  - specialize encoding and transformations to specific engines
  - communicate results between engines
- richer properties
  - termination, liveness, synthesis