SeaHorn: Software Model Checking with SMT and AI

Arie Gurfinkel

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

http://ece.uwaterloo.ca/~agurfink

based on work with Teme Kahsai, Jorge Navas, Anvesh Komuravelli, Jeffrey Gennari, Ed Schwartz, and many others
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?

A 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
SeaHorn

A fully automated verification framework for LLVM-based languages.

http://seahorn.github.io
http://seahorn.github.io

Temesghen Kahsai (Amazon)

Jorge Navas (SRI)
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”
Algorithmic Logic-Based Verification

1. Program + Spec
2. Verification Condition (in Logic)
3. Decision Procedure

- Yes
- No

- Safety Properties
- Constrained Horn Clauses
- Spacer
SeaHorn Usage

Example: in test.c, check that $x$ is always greater than or equal to $y$

```c
extern int nd();
extern void __VERIFIER_error() __attribute__((noreturn));
void assert (int cond) { if (!cond) __VERIFIER_error (); } 
int main(){
    int x,y;
    x=1; y=0;
    while (nd ()
    {
        x=x+y;
        y++;
    } 
assert (x>=y);
return 0;
}
```

SeaHorn command:
```
$>$ sea pf test.c
```

SeaHorn result:
```
PROPERTY (line 12) | TRUE
TIME(ms) | 0.06
```
SeaHorn Workflow

Code Under Analysis (CUA)

Property Spec

Verification Environment

Property Checker

Verification Problem (VP)

SeaHorn

Good + Verification Certificate (Cert)

Bad + Counterexample (CEX)

TestGen

Test harness (Test)
SeaHorn workflow components

Code Under Analysis (CUA)
- code being analyzed. Device driver, component, library, etc.

Verification Environment
- stubs for the environment with which CUA interacts
- e.g., libc, memcpy, malloc, OS system calls, user input, socket, file, …

Property Checker
- static instrumentation of a program with a monitor that indicates when an error has happened
- similar to dynamic sanitizers, but can use verifier-specific API to perform symbolic actions
- property spec is specific to a property checker

Verification Problem
- a prepared instance of program with embedded assertions, potentially simplified by abstracting away irrelevant parts of execution

Test Gen
- generates a test harness that includes all stubs and stimuli to guide CUA to a property failure discovered by the verifier
Developing a Static Property Checker

A static property checker is similar to a dynamic checker

- e.g., clang sanitizer (address, thread, memory, etc.)

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
  - e.g., memory safety, null-dereference, taint checking, use-after-free, etc.

SeaHorn property checkers:

- memory safety (out of bound uses, null pointer)
  - ongoing work to improve scalability and usability
- taint analysis (being developed by Princeton)
DEMO
Architecture of Seahorn

Front-end
- Clang
- C/C++
- LLVM bitcode
- LLVM Opt:
  - SSA
  - DCE
  - Peephole
  - CFG Simplification
- Devirtualization and Exception Lowering
- Property Instr:
  - Buffer overflow
  - Null dereferences
- Slicing Assertions

Middle-end
- Heap Abstraction
- VC Generation
  - Precision:
    - Integers
    - Floating point
    - Pointers
    - Memory contents
- Array Abstraction

Back-end
- PDR/IC3-based Model checking
- Abstract Interp.
  - Intervals
  - DBMs
  - LDDs
- Template-based (Houdini)
- BMC bitvectors
Crab Abstract Interpretation Library

Crab – Cornucopia of Abstract Domains
- Numerical domains (intervals, zones, boxes)
- 3rd party domains (apron, elina)
- arrays, uninterpreted functions, null, pointer

Language independent core with plugins for LLVM bitcode
- fixedpoint engine
- widening / narrowing strategies
- crab-llvm: integrates LLVM optimizations and analysis of LLVM bitcode

Support for inter-procedural analysis
- pre-, post-conditions, function summaries

Extensible, publicly available on GitHub, open C++ API
Crab Abstract Domains

Numerical Domains
- interval with congruence: $0 \leq x \leq 10 \land x \mod 2 = 0$
- zone: $x - y \leq k$
- non-convex
  - DisIntervals: $x \leq -1 \lor x \geq 1$
  - Boxes: Boolean combinations of intervals

Symbolic Domains
- numeric domains extended with uninterpreted functions
- $0 \leq x \leq 10 \land y = f(\ldots) \land z = f(\ldots) \Rightarrow 0 \leq x \leq 10 \land y = z$

Array Domains
- array smashing: common properties of all array cells
- array graph domain:

Domains from Apron and Elina 3rd party libraries
- octagons, polyhedra, etc.
Architecture of Crab and Crab-Llvm

Crab

- Abstract Domains
  - Arrays
  - Null
  - Numerical
- Fixpoint Engines
- Forward/Backward Analyses
- Interprocedural Analyses
- CFG/Callgraph Builder
- Heap Abstraction
- LLVM Optimizations

Checkers
- Assert
- ABC
- Null
- DBZ

Crab-Llvm

- LLVM Bitcode
- C++ API
- SeaHorn

https://github.com/seahorn/crab-llvm
SeaHorn Memory Model

Block-based memory model
- each allocation (malloc/alloca/etc) creates a new object
- a pointer is a pair (id, off), called cell, where id is an object identifier and off is a positive numeric offset
- similar to the C memory model

Abstract Memory Model
- the number of allocation regions is finite
- allocation site is used as an object identifier
- custom pointer-analysis is used to approximate abstract points to graph

Pointer Analysis: Sea-DSA
- unification-based (like LLVM-DSA)
- context-, field-, and array-sensitive
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
Logic-based Program Verification

Low-Level Bounded Model Checking (BMC)
• decide whether a low level program/circuit has an execution of a given length that violates a safety property
• effective decision procedure via encoding to propositional SAT

High-Level (Word-Level) Bounded Model Checking
• decide whether a program has an execution of a given length that violates a safety property
• efficient decision procedure via encoding to SMT

What is an SMT-like equivalent for Safety Verification?
• Logic: SMT-Constrained Horn Clauses
• Decision Procedure: Spacer / GPDR
  – extend IC3/PDR algorithms from Hardware Model Checking
System S is safe iff there exists an inductive invariant $\text{Inv}$:

- **Initiation:** $\text{Initial} \subseteq \text{Inv}$
- **Safety:** $\text{Inv} \cap \text{Bad} = \emptyset$
- **Consecution:** $\text{TR}(\text{Inv}) \subseteq \text{Inv}$  
  i.e., if $s \in \text{Inv}$ and $s \xrightarrow{\cdot} t$ then $t \in \text{Inv}$
Inductive Invariants

System State Space

Initiation: \( \text{Initial} \subseteq \text{Inv} \)

Safety: \( \text{Inv} \cap \text{Bad} = \emptyset \)

Consecution: \( \text{TR}(\text{Inv}) \subseteq \text{Inv} \) i.e., if \( s \in \text{Inv} \) and \( s \sim t \) then \( t \in \text{Inv} \)

System S is safe if \( \text{Reach} \cap \text{Bad} = \emptyset \)
Symbolic Reachability Problem

\[ P = (V, \text{Init}, Tr, Bad) \]

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

\[
\text{Init}(X_0) \land \left( \bigwedge_{i=0}^{N-1} Tr(X_i, X_{i+1}) \right) \land \text{Bad}(X_N) \not\Rightarrow \bot
\]

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

\[
\text{Init} \Rightarrow \text{Inv} \\
\text{Inv}(X) \land Tr(X, X') \Rightarrow \text{Inv}(X') \\
\text{Inv} \Rightarrow \neg \text{Bad}
\]
Constrained Horn Clauses (CHC)

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

$$
\forall V \cdot (\phi \land p_1[X_1] \land \ldots \land p_n[X_n] \rightarrow h[X]),
$$

where

- $A$ is a background theory (e.g., Linear Arithmetic, Arrays, Bit-Vectors, or combinations of the above)
- $\phi$ 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
Horn Clauses for Program Verification

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

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 $V, V'$,

CM1: \( \text{init}(V) \rightarrow R_1(V) \)

CM2: \( R_2(V) \land \rho_i(V, V') \rightarrow R_1(V') \)

CM3: \( \bigvee_{i \in 1, \ldots, N} \bigvee_{j \in \{1, \ldots, m \}} R_i(V) \land \rho_i(V, V') \rightarrow E_2(V, V') \)

CM4: \( R_4(V) \land E_4(V, V') \land \rho_i(V, V') \rightarrow R_3(V') \)

CM5: \( R_5(V) \land \cdots \land R_N(V) \land \text{error}(V) \rightarrow \text{false} \)

\[ \text{multi-threaded program } P \text{ is safe} \]

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

Hojjat et al. Horn Clauses for Communicating Timed Systems. HCVS’14

Hoenicke et al. Thread Modularity at Many Levels. POPL’17

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

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 $r = \max\{m, k\}$.

\[
\begin{align*}
\{ & R(g, p_{\sigma(1)}, l_{\sigma(1)}, \ldots, p_{\sigma(k)}, l_{\sigma(k)}) \leftarrow \text{dist}(p_1, \ldots, p_k) \land R(g, p_1, l_1, \ldots, p_k, l_k) \} \sigma \in S_k \\
R(g, p_1, l_1, \ldots, p_k, l_k) & \leftarrow \text{dist}(p_1, \ldots, p_k) \land \text{Init}(g, l_1) \land \cdots \land \text{Init}(g, l_k) \\
R(g', p_1, l_1, \ldots, p_k, l_k) & \leftarrow \text{dist}(p_1, \ldots, p_k) \land \bigwedge_{i \in \{1, \ldots, m\}} (g, l_i)_{\rho_1} (g', l_i') \land R(g, p_1, l_1, \ldots, p_k, l_k) \\
R(g', p_1, l_1, \ldots, p_k, l_k) & \leftarrow \text{dist}(p_0, p_1, \ldots, p_k) \land ((g, l_0)_{\rho_1} (g', l_0')) \land R\text{Conj}(0, \ldots, k) \\
\text{false} & \leftarrow \text{dist}(p_1, \ldots, p_r) \land \left( \bigwedge_{i = 1}^{m} (p_j = p_j \land (g, j) \in E_j) \right) \land R\text{Conj}(1, \ldots, r)
\end{align*}
\]

Figure 6. Horn clause encoding for thread modularity at level $k$ (where $(\ell, s, s')$ and $(\ell', s, \cdots)$ refer to statement $s$ on a trace from $\ell$ to $\ell'$ and, respectively, from $\ell'$ to some other location in the control flow graph).

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

$\text{Init}(i, j, \overline{v}) \land \text{Init}(j, i, \overline{v})$ $\land$ $\text{Init}(i, i, \overline{v}) \land \text{Init}(j, j, \overline{v}) \Rightarrow I_2(i, j, \overline{v})$ $\land$ $I_2(i, j, \overline{v}) \land \text{Tr}(i, \overline{v}, \overline{v'}) \Rightarrow I_2(i, j, \overline{v'})$ $\land$ $I_2(i, j, \overline{v}) \land \text{Tr}(j, \overline{v}, \overline{v'}) \Rightarrow I_2(i, j, \overline{v'})$ $\land$ $I_2(i, j, \overline{v}) \lor I_2(i, k, \overline{v}) \lor I_2(j, k, \overline{v})$ $\land$ $\text{Tr}(k, \overline{v}, \overline{v'}) \land k \neq i \land k \neq j \Rightarrow I_2(i, j, \overline{v'})$ $\land$ $I_2(i, j, \overline{v}) \Rightarrow \neg \text{Bad}(i, j, \overline{v})$
From Programs to Logic

Program

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

CFG

```
\text{l}_0: 
\begin{align*}
    x &= 1 \\
    y &= 0
\end{align*}
\text{l}_1: b_1 = \text{nondet()}
\text{l}_2: 
\begin{align*}
    x &= x + y \\
    y &= y + 1
\end{align*}
\text{l}_3: 
\begin{align*}
    b_2 &= x \geq y
\end{align*}
\text{l}_4: 
\text{l}_\text{err}:
```

CHC

```
\langle 1 \rangle \ p_0.
\langle 2 \rangle \ p_1(x, y) \leftarrow
    \begin{align*}
        p_0, &\ x = 1, y = 0.
    \end{align*}
\langle 3 \rangle \ p_2(x, y) \leftarrow p_1(x, y).
\langle 4 \rangle \ p_3(x, y) \leftarrow p_1(x, y).
\langle 5 \rangle \ p_1(x', y') \leftarrow
    \begin{align*}
        p_2(x, y), \\
        x' &= x + y, \\
        y' &= y + 1.
    \end{align*}
\langle 6 \rangle \ p_4 \leftarrow (x \geq y), p_3(x, y).
\langle 7 \rangle \ p_\text{err} \leftarrow (x < y), p_3(x, y).
\langle 8 \rangle \ p_4 \leftarrow p_4.
\langle 9 \rangle \bot \leftarrow p_\text{err}.
```
Spacer: Solving SMT-constrained CHC

Spacer: a solver for SMT-constrained Horn Clauses
• now part of Z3
  – https://github.com/Z3Prover/z3 since commit 72c4780
  – use option fixedpoint.engine=spacer
• development version at http://bitbucket.org/spacer/code

Supported SMT-Theories
• Best-effort support for many SMT-theories
  – data-structures, bit-vectors, non-linear arithmetic
• Linear Real and Integer Arithmetic
• Quantifier-free theory of arrays
• Universally quantified theory of arrays + arithmetic (work in progress)

Support for Non-Linear CHC
• for procedure summaries in inter-procedural verification conditions
• for compositional reasoning: abstraction, assume-guarantee, thread modular, etc.
IC3, PDR, and Friends (1)

IC3: A SAT-based Hardware Model Checker
- Incremental Construction of Inductive Clauses for Indubitable Correctness
- A. Bradley: SAT-Based Model Checking without Unrolling. VMCAI 2011

PDR: Explained and extended the implementation
- Property Directed Reachability
- N. Eén, A. Mishchenko, R. K. Brayton: Efficient implementation of property directed reachability. FMCAD 2011

PDR with Predicate Abstraction (easy extension of IC3/PDR to SMT)
GPDR: Non-Linear CHC with Arithmetic constraints
• Generalized Property Directed Reachability
• K. Hoder and N. Bjørner: Generalized Property Directed Reachability. SAT 2012

SPACER: Non-Linear CHC with Arithmetic
• fixes an incompleteness issue in GPDR and extends it with under-approximate summaries
• A. Komuravelli, A. Gurfinkel, S. Chaki: SMT-Based Model Checking for Recursive Programs. CAV 2014

PolyPDR: Convex models for Linear CHC
• simulating Numeric Abstract Interpretation with PDR
• N. Bjørner and A. Gurfinkel: Property Directed Polyhedral Abstraction. VMCAI 2015

ArrayPDR: CHC with constraints over Arithmetic + Arrays
• Required to model heap manipulating programs
• A. Komuravelli, N. Bjørner, A. Gurfinkel, K. L. McMillan: Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays. FMCAD 2015
Generalizing from Bounded Proofs

A counterexample of length $N$ exists?

- Yes: No + bounded proof
- No: $N := N + 1$

If no counterexample exists, check if the candidate $Inv$ is a safe inductive invariant.

- Yes: T, $N = 0$
- No: Generalize proof
Algorithm Overview

**Input**: Safety problem \( \langle \text{Init}(X), \text{Tr}(X, X'), \text{Bad}(X) \rangle \)

\[ F_0 \leftarrow \text{Init}; N \leftarrow 0 \text{ repeat} \]

\[ G \leftarrow \text{PdrMkSafe}([F_0, \ldots, F_N], \text{Bad}) \]

if \( G = [ ] \) then return \( \text{Reachable} \);

\( \forall 0 \leq i \leq N \cdot F_i \leftarrow G[i] \)

\[ F_0, \ldots, F_N \leftarrow \text{PdrPush}([F_0, \ldots, F_N]) \]

if \( \exists 0 \leq i < N \cdot F_i = F_{i+1} \) then return \( \text{Unreachable} \);

\( N \leftarrow N + 1; F_N \leftarrow \emptyset \)

until \( \infty \);
Spacer/IC3/PDR In Pictures: MkSafe

\[ x = 3, y = 0 \]

\[ x = 1, y = 0 \]

\[ x < y \]
Algorithm Invariants

\[ F_i \rightarrow \neg \text{Bad} \quad \text{Init} \rightarrow F_i \]

\[ F_i \rightarrow F_{i+1} \quad F_i \land Tr \rightarrow F'_{i+1} \]
Logic-based Algorithmic Verification

Simulink

CoCoSim

Lustre

Zustre

Termination for C

T2

Java

JayHorn

C/C++

SeaHorn

concurrent/distributed systems

CPR

Spacer
SV-COMP 2015

4\textsuperscript{th} Competition on Software Verification held 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
• \texttt{http://sv-comp.sosy-lab.org/2015/benchmarks.php}
Results for DeviceDriver category
Applications of SeaHorn at NASA

Absence of Buffer Overflows
- Open source auto-pilots
  - paparazzi and mnav autopilots
- Automatically instrument buffer accesses with runtime checks
- Use SeaHorn to validate that run-time checks never fail
  - slower than pure abstract interpretation
  - BUT, much more precise!

Verify Level 5 requirements of the LADEE software stack
- Manually encode requirements in Simulink model
- Verify that the requirements hold in auto-generated C

Memory safety of C++ controller code
- ongoing…
SeaHorn at a glance

Publicly Available ([http://seahorn.github.io](http://seahorn.github.io))
state-of-the-art Software Model Checker

Industrial-strength front-end based on Clang and LLVM

Advanced Abstract Interpretation engine: Crab

SMT-based verification engine:Spacer

Bit-precise Bounded Model Checker and Symbolic Execution

Executable Counter-Examples

A framework for research and application of logic-based verification
Current and Future Work

Precise Memory Analysis
• pointer / alias analysis for LLVM
• bug discovery using symbolic execution
• verification of buffer overflows, null-deref, memory safety
• specialized checkers / proof rules / verification conditions

Verification of Concurrent / Distributed / Parametrized Systems
• modular verification (per thread, per task, per node)
• scale to systems with large / unbounded interacting components

Scalability and Precision
• develop and implement new algorithms to increase scalability and/or precision
• effective modular reasoning / slicing / lemma learning
• bit-precise verification
References

Tools:
• SeaHorn: http://seahorn.github.io/

Papers:
• Blog: http://seahorn.github.io/blog/
• A. Gurfinkel, T. Kahsai, A. Komuravelli, J.A. Navas: The SeaHorn Verification Framework. CAV (1) 2015: 343-361
• C. Urban, A. Gurfinkel, T. Kahsai: Synthesizing Ranking Functions from Bits and Pieces. TACAS 2016: 54-70A.
2000 started PhD in MC at UofT
multi-valued model checking

2006 SMC Yasm: safety, liveness, multi-valued abstraction for MC

2010 Boxes abstract domain (SAS’10)

2012 UFO: MC + AI: SAS’12

2015 SeaHorn: MC (Spacer) and AI (Crab)