SeaHorn: A Framework for Verifying C Programs (Competition Contribution)[*]

Arie Gurfinkel[1], Temesghen Kahsai[2], and Jorge A. Navas[3]

[1] Software Engineering Institute / CMU, USA
[2] NASA Ames Research Center / CMU, USA
[3] NASA Ames Research Center / SGT, USA

Abstract. SeaHorn is a framework and tool for verification of safety properties in C programs. The distinguishing feature of SeaHorn is its modular design that separates how program semantics is represented from the verification engine. This paper describes its verification approach as well as the instructions on how to install and use it.

1 Verification Approach

SeaHorn is a framework and a tool for verification of safety properties for C programs. It is *parameterized* by the semantic representation of the program using Horn constraints and by the verification engine that leverages the latest advances made in constraint solving and Abstract Interpretation. The design of SeaHorn provides users with an extensible and customizable environment for experimenting and implementing with new software verification techniques.

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

Consider the simple program on the left. Using SeaHorn we encode it using, for instance, classical Hoare Logic:

$$\begin{aligned} (x = 1 \ \wedge \ y = 0) &\rightarrow I(x, y) \\ (I(x, y) \ \wedge \ x' = x + y \ \wedge \ y' = y + 1) &\rightarrow I(x', y') \\ (I(x, y) \ \wedge \ x < y) &\rightarrow \mathit{false} \end{aligned}$$

These logic formulas corresponding to the rule for while loops are indeed a set of recursive Horn clauses. Thus, the problem of proving whether the program is safe is reduced to checking whether these Horn clauses are satisfiable. Fortunately, they can be solved by a means of solvers (e.g., [5]), thus leveraging recent advances in Horn constraint solving.

[*] This material is based upon work funded and supported by NASA Contract No. NNX14AI09G, NSF Award No. 1422705 and by the Department of Defense under Contract No. FA8721-05-C-0003 with CMU for the operation of SEI, an FFRDC. 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. This material has been approved for public release and unlimited distribution. DM-0001865.

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 447–450, 2015.
DOI: 10.1007/978-3-662-46681-0_41

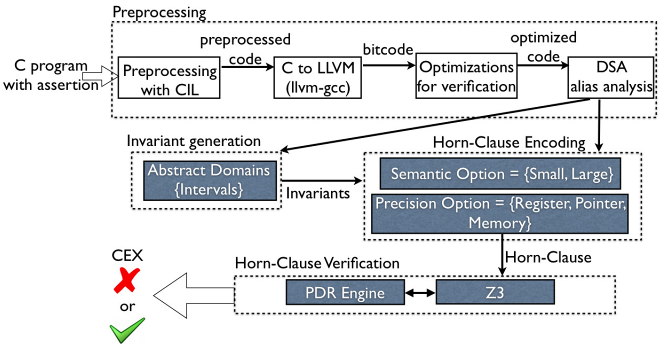

Fig. 1. Overview of SeaHorn architecture

2 Software Architecture

SeaHorn is implemented in C++ in the LLVM compiler infrastructure [6]. The overall approach is illustrated in Figure 1.

Preprocessing. To pre-process the competition benchmark, we utilize the front-end that was originally developed for UFO [1]. First, the input C program is pre-processed with CIL[1] to insert line markings for counterexamples, define missing functions, and initialize all local variables. Second, the result is translated into LLVM Intermediate Representation (IR), called *bitcode*, using `llvm-gcc`. Next, we perform compiler optimizations and preprocessing to simplify the verification task. As a preprocessing step, we further initialize any uninitialized registers using non-deterministic functions. This is used to bridge the gap between the verification semantics (which assumes a non-deterministic assignment) and compiler semantics, which tries to take advantage of the undefined behavior of uninitialized variables to perform code optimizations. We perform a number of program simplifications such as function inlining, static single assignment (SSA) form, dead code elimination, etc. Finally, we use a variant of Data Structure Analysis (DSA), an alias analysis that infers disjoint heap regions used to identify each memory access within a certain region.

Invariant Generation. Inductive invariants can be computed from the bytecode using a given abstract domain. SeaHorn uses the IKOS library [2] which is a collection of abstract domains and fixpoint iteration algorithms. SeaHorn runs in parallel with (using classical intervals) and without invariant generation.

Horn-Clause Encoding. Next, we translate bytecode to Horn constraints which acts as a very suitable intermediate representation for verification. SeaHorn is parametric on the semantics used for encoding. Currently, SeaHorn provides a Horn-clause style encoding based on small-step semantics [7] as well as a more efficient large-block encoding [3]. For the competition, we always use the

[1] `http://www.cs.berkeley.edu/~necula/cil/`

large-block encoding. The level of precision of the encoding can be also tuned. The options are: only registers (integer scalars), registers and pointer addresses (without content), and all of the above plus memory content (using theory of arrays). We use for the competition the latter which is the most precise level.

Horn-Clause Verification. SeaHorn is also parameterized by the solver. For the competition, SeaHorn uses pdr engine implemented in Z3 [4]. For the competition we improve pdr using invariants computed by IKOS. To motivate this decision, let us come back to our example described above. pdr alone can discover $x \geq y$ but it does not terminate, however, if populated with the inductive invariant $y \geq 0$, computed by IKOS, it proves it immediately.

3 Strength and Weaknesses

SeaHorn uses linear arithmetic to reason about scalars and pointer addresses, and theory of arrays for memory contents. However, SeaHorn provides little or no support for reasoning about dynamic linked data structures, bit-level precision, or concurrency. Another weakness of SeaHorn is inherited from the UFO front-end which relies on multiple tools: LLVM 2.6, LLVM 2.9, and CIL. The main strength of SeaHorn lies on its parameterized nature allowing experimenting with different encodings to model new semantics aspects, abstractions and verification algorithms.

4 Tool Setup

SeaHorn is available for download from `https://bitbucket.org/lememta/seahorn/wiki/Home`. SeaHorn is provided as a set of binaries and libraries for Linux x86-64 architecture. The options for running the tool are:

```
./bin/seahorn-svcomp-par.py [-m64] [--cex=CEX] [--spec=SPEC] INPUT
```

where `-m64` turns on 64-bit model, CEX is the destination directory for the witness file, SPEC is the property file, and `INPUT` is a C file. If it terminates the output of SeaHorn is ``` ``Result TRUE'' ``` when the program is safe, ``` ``Result FALSE'' ```, when a counterexample is found or ``` ``Result UNKNOWN'' ```, otherwise.

References

1. Albarghouthi, A., Gurfinkel, A., Li, Y., Chaki, S., Chechik, M.: UFO: Verification with interpolants and abstract interpretation. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 637–640. Springer, Heidelberg (2013)
2. Brat, G., Navas, J.A., Shi, N., Venet, A.: IKOS: A framework for static analysis based on abstract interpretation. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 271–277. Springer, Heidelberg (2014)

3. Gurfinkel, A., Chaki, S., Sapra, S.: Efficient predicate abstraction of program summaries. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 131–145. Springer, Heidelberg (2011)
4. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
5. Hoder, K., Bjørner, N., de Moura, L.: *μZ*– an efficient engine for fixed points with constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 457–462. Springer, Heidelberg (2011)
6. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO. pp. 75–88 (2004)
7. Peralta, J.C., Gallagher, J.P., Saglam, H.: Analysis of imperative programs through analysis of constraint logic programs. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 246–261. Springer, Heidelberg (1998)