Algorithmic Logic-Based Verification with SeaHorn

(Invited Tutorial)

Arie Gurfinkel
Software Engineering Institute
Carnegie Mellon University
http://arieg.bitbucket.org

***Abstract*—In this tutorial, I will present SEAHORN, a software verification framework. The key distinguishing feature of SEAHORN is its modular design that separates the concerns of the syntax of the programming language, its operational semantics, and the verification semantics. SEAHORN encompasses several novelties: it (a) encodes verification conditions using an efficient yet precise inter-procedural technique, (b) provides flexibility in the verification semantics to allow different levels of precision, (c) leverages the state-of-the-art in software model checking and abstract interpretation for verification, and (d) uses Horn-clauses as an intermediate language to represent verification conditions which simplifies interfacing with multiple verification tools based on Horn-clauses. SEAHORN provides users with a powerful verification tool and provides researchers with an extensible and customizable framework for experimenting with new software verification techniques.**

I. INTRODUCTION

Program verification – deciding whether a given program satisfies its specification, is one of the oldest problems in computer science. In his seminal paper, "Checking a Large Routine" [1], Alan Turing outlined a methodology to formally check whether a procedure (or a routine) is *right*. In particular, he proposed flowcharts as a concise program representation, and described a method based on the insight that a programmer should make a number of definite assertions which can be proven individually, and from which the correctness of the whole program follows easily. Almost two decades later, Floyd [2] and Hoare [3], inspired by the works of McCarthy [4] and Naur [5], independently, proposed a logic based on a deductive system that is known today as the *Floyd-Hoare logic*. The logic allowed proving correctness of programs in a rigorous manner in ways foreshadowed by Turing. Another decade later, Dijkstra [6] developed the first semi-algorithmic view of the Floyd-Hoare logic based on the notion of *predicate transformers*. The field of software verification has been growing rapidly ever since. Today, *Abstract Interpretation* [7], *Model Checking* [8], [9], and *Symbolic Execution* [10] are probably the most predominant algorithmic (i.e., fully automated) verification techniques.

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. This material has been approved for public release and unlimited distribution. DM-0002704

We argue that a significant disadvantage of the existing techniques is that each individual technique relies on its own specific set of intermediate representations and interpretation of the program's semantics. This makes it difficult, if not impossible, to effectively combine multiple techniques and reuse results between techniques.

We suggest an alternative, that we call *Algorithmic Logic-Based Verification* that is completely automated (i.e., algorithmic) and uses logic as its intermediate representation. In particular, we use *Constrained Horn Clauses (CHCs)*, a fragment of First Order Logic, as the basis for the intermediate representation.

Our approach separates the concerns of the programming language syntax, operational- and verification-semantics. The main idea is that the verification process can be partitioned into generating a Verification Condition (VC) in CHC, and determining satisfiability of VC using logic-based decision procedures. This basic idea is not new. For example, CHCs are similarly used as a basis of *Constraint Logic Programming* (CLP) [11]. What makes our approach unique is the use of novel powerful decision engines, called SMT solvers, that have been recently developed and perfected in the verification community.

In this tutorial, I will present SEAHORN, a state-of-the-art CHC-based automated verification framework. SEAHORN aims at providing developers and researchers a collection of modular and reusable verification components that can reduce the burden of building a new software verifier. Similar to modern compilers, SEAHORN is split into three main components: the front-end, the middle-end, and the back-end.

The front-end deals with the syntax of the input programming language and generates an internal intermediate representation. The middle-end encodes the verification condition as CHC. SEAHORN provides several out-of-the-box encodings that have been shown useful in practice. Finally, the back-end discharges the verification conditions using a variety of state-of-the-art SMT-based model checking and abstract interpretation-based solvers.

This versatile and flexible design supports multiple VC encodings and multiple verification engines. It also simplifies targeting new programming languages or language specifications by providing a translation to CHCs. SEAHORN an interesting verification infrastructure that allows developers and researchers to experiment with new techniques.

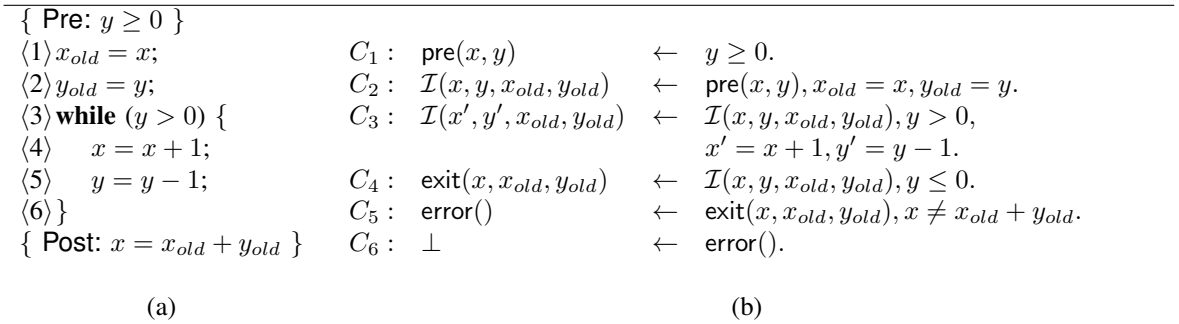

Fig. 1: A program and its verification conditions expressed as a set of CHCs.

II. Constrained Horn Clauses

In this section, we give a brief overview of Constrained Horn Clauses. More details are available in [12].

Given the sets $\mathcal{F}$ of function symbols, $\mathcal{P}$ of predicate symbols, and $\mathcal{V}$ of variables, a *Constrained Horn Clause (CHC)* is a formula:

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

where ϕ is a constraint over $\mathcal{F}$ and $\mathcal{V}$ with respect to some background theory $\mathcal{A}$; $X_i, X \subseteq \mathcal{V}$ are (possibly empty) vectors of variables; $p_i[X_i]$ is an application $p(t_1, \ldots, t_n)$ of an n-ary predicate symbol $p \in \mathcal{P}$ for first-order terms t_i constructed from $\mathcal{F}$ and X_i; and $h[X]$ is either defined analogously to p_i or is $\mathcal{P}$-free (i.e., no $\mathcal{P}$ symbols occur in h). We usually assume that the background theory $\mathcal{A}$ is a combination of Linear Arithmetic, Arrays, and Bit-Vectors.

We call h the *head* of the clause and $\phi \wedge p_1[X_1] \wedge \cdots \wedge p_k[X_k]$ — the *body*. A clause is called a *query* if its head is $\mathcal{P}$-free. Otherwise, it is called a *rule*. A rule with body true is called a *fact*. A clause is *linear* if its body contains at most one predicate symbol, otherwise, it is called *non-linear*. We often follow the CLP convention of writing Horn clauses as $h[X] \leftarrow \phi, p_1[X_1], \ldots, p_k[X_k]$ with all free variables implicitly universally quantified.

A set of CHCs is satisfiable if there exists an interpretation $\mathcal{J}$ of the predicate symbols $\mathcal{P}$ such that each constraint ϕ is true under $\mathcal{J}$.

CHCs naturally represent verification conditions obtained from Dijkstra's weakest liberal precondition calculus [6]. To illustrate, Figure 1(a) shows a simple imperative program with a pre-condition $y \geq 0$ and a post-condition $x = x_{old} + y = y_{old}$. The corresponding verification condition is shown in Figure 1(b). The predicate pre represents the precondition. The predicate $\mathcal{I}$ represents the loop invariant (which must be discovered to discharge our proof obligation). The predicate exit represents the program state at the end of the execution. Finally, the predicate error represents the error condition (i.e., the negation of the desired post-condition.

The program in Figure 1(a) satisfies its pre- post-condition pair iff the set of CHCs in Figure 1(b) is satisfiable. For example, giving this clauses to a CHC-solver SPACER [13], we get that the system is satisfiable, and the the safe inductive invariant is

$$\mathcal{I}(x, y, x_{old}, y_{old}) \leftrightarrow x + y = x_{old} + y_{old} \wedge y \geq 0\,.$$

III. SeaHorn Verification Framework

SEAHORN [14] is an analysis framework for verification of safety properties of programs. It is based on the LLVM compiler toolkit [15]. SEAHORN is a completely automated program analysis tool that checks user-supplied assertions. Additionally, SEAHORN provides many built-in instrumentations, such as checks for buffer and signed integer overflow. Moreover, SEAHORN is also a framework that simplifies development and integration of new verification techniques.

The design of SEAHORN provides users, developers, and researchers with an extensible and customizable environment for experimenting with and implementing new software verification techniques. SEAHORN is implemented in C++ in the LLVM compiler infrastructure [15]. Its architecture is shown in Figure 2. SEAHORN has been developed in a modular fashion; its architecture is layered in three parts:

Front-End: Takes an LLVM based program (e.g., C) input program and generates LLVM IR bitcode. Specifically, it performs the pre-processing and optimization of the bitcode for verification purposes.

Middle-End: Takes as input the optimized LLVM bitcode and emits verification condition as Constrained Horn Clauses (CHC). The middle-end is in charge of selecting the encoding of the VCs and the degree of precision.

Back-End: Takes CHC as input and outputs the result of the analysis. In principle, any verification engine that digests CHC clauses could be used to discharge the VCs. Currently, SEAHORN employs several SMT-based model checking engines based on PDR/IC3 [16], including SPACER [13], [17] and GPDR [18]. Complementary, SEAHORN uses the abstract interpretation-based analyzer IKOS (Inference Kernel for Open Static Analyzers) [19] for providing numerical invariants.

The effectiveness and scalability of SEAHORN are demonstrated by the results of the Software Verification Competition (SV-COMP 2015) [20].

We conclude with the discussion of the main features of SEAHORN:

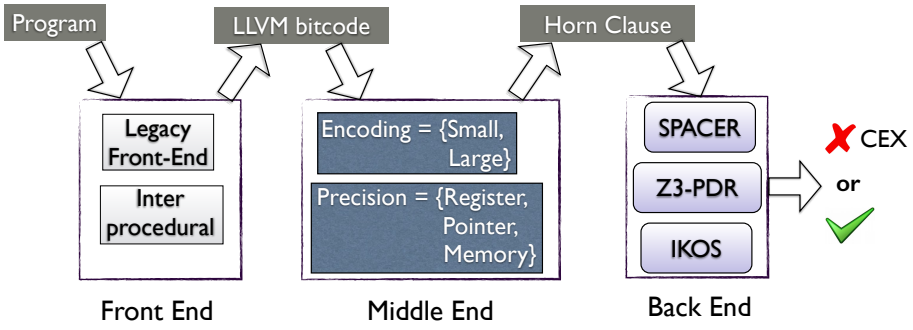

Fig. 2: Overview of SEAHORN architecture.

1) *It decouples a programming language syntax and semantics from the underlying verification technique.* Programming languages include a diverse assortments of features, many of which are purely syntactic. Handling them fully is a major effort for new tool developers. We tackle this problem in SEAHORN by separating the language syntax, its operational semantics, and the underlying verification semantics – the semantics used by the verification engine. Specifically, we use the LLVM front-end(s) to deal with the idiosyncrasies of the syntax. We use LLVM intermediate representation (IR), called the *bitcode*, to deal with the operational semantics, and apply a variety of transformations to simplify it further. Finally, we use Constrained Horn Clauses (CHC) to logically represent the verification condition (VC).
2) *It provides an efficient and precise analysis of programs with procedure using inter-procedural verification techniques.* SEAHORN summarizes the input-output behavior of procedures efficiently without inlining. Moreover, it uses program transformations that lifts deep assertions closer to the main procedure. This increases context-sensitivity of intra-procedural analyses (used both in verification and compiler optimization), and has a significant impact on our inter-procedural verification algorithms.
3) *It allows developers to customize the verification semantics and offers users verification semantics of various degrees of precision.* SEAHORN is fully parametric in the (small-step operational) semantics used for the generation of VCs. The level of abstraction in the built-in semantics varies from considering only LLVM numeric registers to considering the whole heap (modeled as a collection of non-overlapping arrays). In addition to generating VCs based on small-step semantics [21], it can also automatically lift small-step semantics to large-step [22], [23] (a.k.a. Large Block Encoding, or LBE).
4) *It uses Constrained Horn Clauses (CHC) as its intermediate verification language.* CHC provide a convenient and elegant way to formally represent many encoding styles of verification conditions. The recent popularity of CHC as an intermediate language for verification engines makes it possible to interface SEAHORN with a variety of new and emerging tools.
5) *It builds on the state-of-the-art in Software Model Checking (SMC) and Abstract Interpretation (AI).* SMC and AI have independently led over the years to the production of analysis tools that have a substantial impact on the development of real world software. Interestingly, the two exhibit complementary strengths and weaknesses (see e.g., [24]–[27]). While SMC so far has been proved stronger on software that is mostly control driven, AI is quite effective on data-dependent programs. SEAHORN combines SMT-based model checking techniques with program invariants supplied by an abstract interpretation-based tool.
6) Finally, *it is implemented on top of the open-source LLVM compiler infrastructure*. The latter is a well-maintained, well-documented, and continuously improving framework. It allows SEAHORN users to easily integrate program analyses, transformations, and other tools that targets LLVM. Moreover, since SEAHORN analyses LLVM IR, this allows to exploit a rapidly-growing frontier of LLVM front-ends, encompassing a diverse set of languages. SEAHORN itself is released as open-source as well (source code can be downloaded from http://seahorn.github.io).

ACKNOWLEDGMENT

The author would like to thank the group of amazing collaborators without whom this work would not have been possible: Nikolaj Bjørner, Temegshen Kahsai, Anvesh Komuravell, and Jorge A. Navas.

REFERENCES

[1] A. Turing, "Checking a Large Routine," 1949.
[2] R. W. Floyd, "Assigning meanings to programs," *Symposium Applied Mathematics*, no. 10, pp. 19–32, 1967.
[3] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.
[4] J. McCarthy, "A Basis for a Mathematical Theory of Computation," pp. 33–70, 1963.
[5] P. Naur, "Proof of algorithms by general snapshots," vol. 6, pp. 310–316, 1966.
[6] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, 1975.
[7] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages*, 1977, pp. 238–252.

[8] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, 1981, pp. 52–71.

[9] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, 1982, pp. 337–351.

[10] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[11] J. Jaffar and J.-L. Lassez, "Constraint logic programming," in *POPL*, 1987, pp. 111–119.

[12] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn Clause Solving for Program Verification," in *Proceedings of a Symposium on Logic in Computer Science celebrating Yuri Gurevich's 75th Birthday*, 2015.

[13] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-based model checking for recursive programs," in *CAV*, 2014, pp. 17–34.

[14] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The SeaHorn Verification Framework," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., vol. 9206. Springer, 2015, pp. 343–361. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21690-4_20

[15] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–88.

[16] A. R. Bradley, "IC3 and beyond: Incremental, inductive verification," in *CAV*, 2012, p. 4.

[17] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke, "Automatic abstraction in smt-based unbounded software model checking," in *CAV*, 2013, pp. 846–862.

[18] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, 2012, pp. 157–171.

[19] G. Brat, J. A. Navas, N. Shi, and A. Venet, "IKOS: A framework for static analysis based on abstract interpretation," in *SEFM*, 2014, pp. 271–277.

[20] D. Beyer, "Software Verification and Verifiable Witnesses (Report on SV-COMP 2015)," in *TACAS*, 2015.

[21] J. C. Peralta, J. P. Gallagher, and H. Saglam, "Analysis of imperative programs through analysis of constraint logic programs," in *SAS*, 1998, pp. 246–261.

[22] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *FMCAD*, 2009, pp. 25–32.

[23] A. Gurfinkel, S. Chaki, and S. Sapra, "Efficient Predicate Abstraction of Program Summaries," in *NFM*, 2011, pp. 131–145.

[24] A. Gurfinkel and S. Chaki, "Combining predicate and numeric abstraction for software model checking," *STTT*, vol. 12, no. 6, pp. 409–427, 2010.

[25] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "Craig Interpretation," in *SAS*, 2012, pp. 300–316.

[26] P. Garoche, T. Kahsai, and C. Tinelli, "Incremental invariant generation using logic-based automatic abstract transformers," in *NASA Formal Methods, 5th International Symposium, NFM 2013*, 2013, pp. 139–154.

[27] N. Bjørner and A. Gurfinkel, "Property directed polyhedral abstraction," in *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015*, 2015, pp. 263–281.