Algorithmic Logic-Based Verification



Arie Gurfinkel
SEI/CMU

Temesghen Kahsai
NASA Ames/CMU

Jorge A. Navas
NASA Ames/SGT

1. INTRODUCTION

Turing in his seminal paper "Checking a Large Routine" [Turing 1949] already asked the question whether it was possible to check a routine was *right*. Among other contributions, he proposed flowcharts as a concise program representation. He also 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 could easily follow. It took several years until Floyd [Floyd 1967] and Hoare [Hoare 1969], inspired by McCarthy [McCarthy 1963] and Naur [Naur 1966]'s works, established a logic based on a deductive system what is called today *Floyd-Hoare logic* that allowed proving correctness of programs in a rigorous manner. Dijkstra [Dijkstra 1975] presented the first semi-algorithmic view of the Floyd-Hoare logic based on the ideas of *predicate transformers*. Since then, the field of software verification has been growing rapidly during the last decades with many available techniques. Among them, *Abstract Interpretation* [Cousot and Cousot 1977], *Model Checking* [Clarke and Emerson 1981; Queille and Sifakis 1982], and *Symbolic Execution* [King 1976] are probably the most predominant algorithmic (i.e., fully automated) techniques today.

Regardless of the underlying techniques, most software verifiers aim at proving some correctness claims by computing the meaning of the program by either (a) inspecting directly the source code of the program or (b) analyzing some specification describing all program behaviors. Since the problem of computing the meaning of a program is undecidable, most software verifiers offer different trade-offs between completeness, efficiency and accuracy. Therefore, it is highly desirable to combine different techniques to get their maximal advantages. Unfortunately, due to the existence of a myriad of program representations and language specifications, the communication between verifiers is not so simple and the results are often hard to combine and reuse.

In this article, we make a case for *Constrained Horn Clauses (CHCs)*, a fragment of First Order Logic, as the basis for software verification. CHCs are a uniform way to formally represent transition systems while allowing many encoding styles of verification conditions (VCs). Moreover, CHCs allow separating the concerns of the programming

This material is based upon work funded and supported by NASA Contract No. NNX14AI09G, NSF Award No. 1422705 and 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, NASA or NSF. This material has been approved for public release and unlimited distribution. DM-0002273.

language syntax and the verification techniques. The main idea is that Verification Condition Generators (VCGs) translate the input program together with its assertions into a set of VCs represented by means of CHCs while pure logic-based algorithms using, for instance, abstract interpretation and model checking techniques can focus on solving the CHCs. Finally, CHCs provide a formal logical foundations that simplify the sharing of intermediate results.

Although the use of CHC as the basis to represent transition systems is relatively new in the verification community, CHCs have been used for decades in other fields. For example, CHCs are the basis of *Constraint Logic Programming* (CLP) [Jaffar and Lassez 1987]. CLP has been successfully used in many different contexts such as management decision problems, trading, scheduling, electrical circuit analysis, mapping in genetics, *etc.* (see [Marriott and Stuckey 1998] for a survey). Although the standard execution model for CLP is based on *depth-first search* which is incomplete in presence of recursive CHCs, CLP systems are usually augmented with *tabling* capabilities to record calls and their answers for reuse in future calls that can avoid unnecessary infinite computations.

As a result of the success of CLP and LP (i.e., unconstrained Horn clauses) as programming languages, abstract-interpretation-based static analysis of these languages has been a very active area since the 80's. The primary target is code optimization in LP/CLP compilers (see e.g., [Søndergaard 1986; Bruynooghe et al. 1987; Warren et al. 1988; Muthukumar and Hermenegildo 1989]). CLP has been also used as the basis for software model checking [Delzanno and Podelski 1999; Flanagan 2003; Jaffar et al. 2004] of concurrent and timed automata systems as well as in the context of static analysis of imperative and object-oriented languages (e.g., [Peralta et al. 1998; Méndez-Lojo et al. 2007]).

Therefore, it is a fair question to ask why now this renewed interest in the use of CHCs as the basis of analysis and verification? The answer lies in the new powerful decision engines, called SMT solvers, that have been recently developed and perfected in the verification community. Recently, new SMT-based techniques have emerged (e.g., [Hoder and Bjørner 2012a; Grebenshchikov et al. 2012; Komuravelli et al. 2014]) that are able to automatically solve recursive CHCs which were beyond the capabilities of tabled CLP systems. Moreover, together with smart VC encodings larger systems of CHCs can be now solved much faster. These advances have facilitated the implementation of efficient CHC solvers that can combine many existing verification techniques based on abstract interpretation and model checking in more sophisticated ways and compete with existing state-of-the-art approaches.

To provide a concrete example of a state-of-the-art CHC-based verifier, we present in this article SEAHORN an efficient 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 and semantics of the input programming language and generates an internal intermediate representation (IR) more suitable for verification. SEAHORN relies on LLVM's front-ends for this and it uses the LLVM [Lattner and Adve 2004] infrastructure to optimize IR (LLVM bitcode). Although the role of the front-ends are often played down and most research papers tend to omit them, we argue that its role is a predominant one and our experience with SEAHORN demonstrates clearly that the front-end must be a major component in the design of any verifier. Note that with this front-end, SEAHORN does not verify source code but instead the optimized internal representation used by a real compiler (e.g., Clang [1]). Although this is not yet machine code it is a more realistic approach than the one adopted by source code-based verifiers since it takes into consideration the *WYSINWYX (What-You-See-Is-Not-What-You-Execute)* phenomenon. The middle-end uses CHCs to encode the verification conditions that arise from the verification of the LLVM bitcode and it is fully parametric on the semantics used to encode the VCs. SEAHORN provides several out-of-the-box encodings which have been shown useful in practice. Finally, the back-end discharges the verification conditions. Since this is a hard problem SEAHORN uses a variety of state-of-the-art SMT-based model checking and abstract interpretation-based solvers.

This versatile and flexible design not only allows easily interchanging multiple VC encodings and solvers but also it makes possible the verification of new programming languages or language specifications assuming a translation to CHCs is provided. This makes SEAHORN an interesting verification infrastructure that allows developers and researchers experimenting with new techniques.

In spite of the fact that efficiency is not the primary aspect in the design of SEAHORN, it has demonstrated its practicality by its performance at the annual Competition on Software Verification (SV-COMP 2015) [Beyer 2015] as well as a successful experience at verifying industrial software.

2. BACKGROUND

In this section, we describe how verification conditions that arise from a verification problem can be encoded into CHCs so that specialized solvers can check their (un)-satisfiability. This approach has been adopted by an increasing number of verifiers such as Threader [Gupta et al. 2011], UFO [Albarghouthi et al. 2012], SEAHORN [Gurfinkel et al. 2015], HSF [Grebenshchikov et al. 2012], VeriMAP [De Angelis et al. 2014], Eldarica [Rümmer et al. 2013], and TRACER [Jaffar et al. 2012].

2.1. Constrained Horn Clauses

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; $X_i, X \subseteq \mathcal{V}$ 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).

Here, h is called the *head* of the clause and $\phi \wedge p_1[X_1] \wedge \cdots \wedge p_k[X_k]$ is called the *body*. A clause is called a *query* if its head is $\mathcal{P}$-free, and otherwise, it is called a *rule*. A rule with body true is called a *fact*. We say a clause is *linear* if its body contains at most one predicate symbol, otherwise, it is called *non-linear*. In this article, we follow the CLP convention of writing Horn clauses as $h[X] \leftarrow \phi, p_1[X_1], \ldots, p_k[X_k]$.

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 $\mathcal{J}$.

2.2. Weakest preconditions calculus

Dijkstra's weakest preconditions calculus [Dijkstra 1975] is a classical method for proving correctness of programs. The main idea is to reduce the problem of verifying a *Hoare triple* $\{Pre\}P\{Post\}$ to proving a pure first-order logic formula by applying a

[1] A C language family front-end for LLVM (http://clang.llvm.org).

$$
\begin{array}{lcl}
\mathtt{wp}(\mathtt{if}\ C\ S_1\ \mathtt{else}\ S_2, \phi) & \leadsto & (C \wedge \mathtt{wp}(S_1, \phi)) \vee (\neg\, C \wedge \mathtt{wp}(S_2, \phi)) \\
\mathtt{wp}(S_1; S_2, \phi) & \leadsto & \mathtt{wp}(S_1, \mathtt{wp}(S_2, \phi)) \\
\mathtt{wp}(x = e, \phi) & \leadsto & \phi[x \leftarrow e] \\
\mathtt{wp}(\mathtt{error}, \phi) & \leadsto & \bot \\
\mathtt{wp}(\mathtt{while}\ C\ B, \phi) & \leadsto & \mathcal{I}(\overline{x}) \wedge \\
& & \forall \overline{x}, \kappa((\mathcal{I}(\overline{x}) \wedge C \wedge \kappa = \mathcal{B}(\overline{x}) \rightarrow \mathtt{wp}(B, \mathcal{I}(\overline{x}) \wedge \kappa \prec \mathcal{B}(\overline{x}))) \wedge \\
& & (\mathcal{I}(\overline{x}) \wedge \neg\, C \rightarrow \phi)) \\
\mathtt{wp}(\mathtt{return}, \phi) & \leadsto & \phi \\
\mathtt{wp}(f(\bar{i}, \overline{o}), \phi) & \leadsto & \mathcal{S}_f(\bar{i}, \overline{o}) \rightarrow \phi
\end{array}
$$

Fig. 1: Weakest precondition calculus rules for a simple imperative language.

predicate transformer. A well-known transformer is the weakest precondition of P with respect to a formula ϕ denoted by $\mathsf{wp}(P, \phi)$. Formally, $\mathsf{wp}(P, \phi)$ is the weakest condition that needs to hold before executing P such that the execution terminates and the post-condition ϕ holds at the end of the execution. Informally, a Hoare triple $\{Pre\}P\{Post\}$ is valid if and only if $Pre \rightarrow \mathsf{wp}(P, Post)$. The rules that define the wp transformer are shown in Figure 1. The symbol $\mathcal{I}$ denotes a loop invariant and $\mathcal{B}$ denotes a loop variant. The symbol $\mathcal{S}$ denotes the function summary and $\phi[x \leftarrow e]$ represents the formula obtained by syntactically replacing all occurrences of x by e. The symbol $\prec$ is a well-founded relation, i.e., it does not admit any infinite chain.

2.3. From Weakest preconditions calculus to CHCs

We can obtain a set of CHCs by first applying exhaustively the rules in Figure 1 to the formula:

$$
Pre \rightarrow \mathsf{wp}(P, Post) \wedge \bigwedge_{f \in P} \forall \bar{i}, \overline{o}. \mathsf{wp}(B_f, S_f(\bar{i}, \overline{o}))
$$

where B_f is the body of the function f. While the result is not syntactically CHC, it can be put into the syntactically correct form by applying *negation normal form*, *prenex normal form*, and finally *conjunctive normal form* transformations[2]. Finally, we can use many of the abstract interpretation and SMT-based model checking CHC solvers [Komuravelli et al. 2013; McMillan and Rybalchenko 2013; Hoder and Bjørner 2012b; Bjørner et al. 2013; Gange et al. 2013; Hermenegildo et al. 2003; Henriksen and Gallagher 2006; Rümmer et al. 2013] for inferring the unknown relations $\mathcal{I}, \mathcal{B}, \mathcal{S}$.

To illustrate, Figure 2(a) shows a program which adds two numbers. We would like establish validity of the Hoare triple $\{y \geq 0\}P\{x = x_{old} + y_{old}\}$, where P encodes lines 1–6. Figure 2(b) shows the corresponding verification conditions obtained after applying exhaustively the weakest preconditions calculus rules from Figure 1. Note that the VCs are expressed as Constrainted Horn Clauses. The relation pre represents the preconditions of the program. The relation $\mathcal{I}$ expresses the loop invariant which we must infer in order to prove our Hoare triple. The relation exit represents the state after the loop exit is executed, and finally, the relation error expresses our error condition. The clauses C_3, C_4, and C_5 are originated from the rule for `while`. Clause C_1 represents the preconditions of our program and clause C_2 is originated from the rule for sequential composition. Finally, for proving our postcondition we actually generate the following code `if` $(x \neq x_{old} + y_{old})$ `error`. This is the reason why clause C_6 describes our postcondition in negated form.

[2]Note that the variant $\mathcal{B}$ is a function. Thus, the result is non-CHC. In practice, $\mathcal{B}$ is dropped for safety or reachability properties, and turned into a well-founded relation for termination properties.

{ Pre: $y \geq 0$ }
$\langle 1 \rangle x_{old} = x;$
$\langle 2 \rangle y_{old} = y;$
$\langle 3 \rangle$ **while** $(y > 0)$ {
$\langle 4 \rangle \quad x = x + 1;$
$\langle 5 \rangle \quad y = y - 1;$
$\langle 6 \rangle$ }
{ Post: $x = x_{old} + y_{old}$ }

(a)

$$
\begin{array}{lllll}
C_1: & \mathsf{pre}(x,y) & \leftarrow & y \geq 0\ . \\
C_2: & \mathcal{I}(x,y,x_{old},y_{old}) & \leftarrow & \mathsf{pre}(x,y), \\
 & & & x_{old} = x,\ y_{old} = y\ . \\
C_3: & \mathcal{I}(x,y,x_{old},y_{old}). & & \\
C_4: & \mathcal{I}(x',y',x_{old},y_{old}) & \leftarrow & \mathcal{I}(x,y,x_{old},y_{old}) \\
 & & & y > 0, \\
 & & & x' = x + 1, \\
 & & & y' = y - 1. \\
C_5: & \mathsf{exit}(x,x_{old},y_{old}) & \leftarrow & \mathcal{I}(x,y,x_{old},y_{old}), \\
 & & & y \leq 0. \\
C_6: & \mathsf{error}(x,x_{old},y_{old}) & \leftarrow & x \neq x_{old} + y_{old} \\
C_7: & \bot & \leftarrow & \mathsf{error}(x,x_{old},y_{old})
\end{array}
$$

(b)

Fig. 2: Program and its Verification Conditions encoded as CHCs.

The Hoare triple $\{y \geq 0\}P\{x = x_{old} + y_{old}\}$ holds if the query C_7 is satisfiable. If we solve this query together with clauses $C_1, \ldots, C_6$ using SPACER [Komuravelli et al. 2014], we obtain the safe inductive invariant :

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

3. SEAHORN

In this section, we describe SEAHORN, a concrete example of an algorithmic logic-based verification framework. SEAHORN is a fully automated verifier that proves user-supplied assertions as well as a number of built-in safety properties. For example, SEAHORN provides built-in checks for buffer and signed integer overflows. It is released as open-source and its source code is publicly available at http://tinyurl.com/GetSeaHorn.

3.1. Design and implementation

The design of SEAHORN provides users, developers, and researchers with an extensible and customizable environment for experimenting with and implementing new software verification techniques. It has been developed in a modular fashion, inspired by the design of modern compilers. SEAHORN overall architecture is illustrated in Figure 3. Its architecture is layered in three parts:

— **Front-End**: Takes an LLVM-based (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 CHC. The middle-end is in charge of selecting 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 [Bradley 2012], including SPACER [Komuravelli et al. 2013; Komuravelli et al. 2014] and GPDR [Hoder and Bjørner 2012b]. Complementary, SEAHORN uses the abstract interpretation-based analyzer IKOS (Inference Kernel for Open Static Analyzers) [Brat et al. 2014] for providing numerical invariants.

This layered architecture allows to separate the concerns of the input language syntax, its operational semantics, and and the underlying verification semantics – the semantics used by the verification engine.

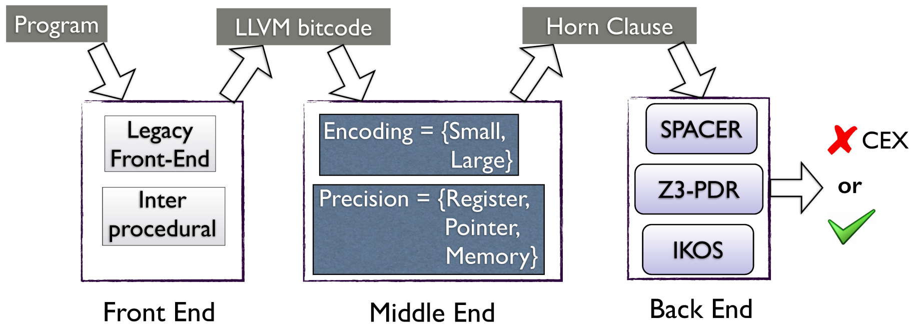

Fig. 3: Overview of SEAHORN architecture.

In the front-end, SEAHORN provides two options: a *legacy* front-end and an *inter-procedural* front-end. The former, has been originally developed for UFO [Albarghouthi et al. 2013], and it has been very effective for solving SV-COMP (2013, 2014, and 2015) problems. However, it has its own limitations: its design is not modular and it relies on multiple unsupported legacy tools (such as `llvm-gcc` and LLVM versions 2.6 and 2.9). Thus, it is difficult to maintain and extend. The inter-procedural front-end, is a generic, modular and easy to maintain front-end. It takes any input program that can be translated into LLVM bitcode. Currently, SEAHORN uses `clang` and `gcc` via `DragonEgg` [3]. In a long run, our goal is to make SEAHORN not to be limited to C programs, but applicable (with various degrees of success) to a broader set of languages based on LLVM (e.g., C++, Objective C, and Swift). The generated LLVM bitcode is then preprocessed and optimized in order to simplify the verification task. Moreover, the inter-procedural front-end provides a transformation based on the concept of *mixed semantics*[4] [Gurfinkel et al. 2008; Lal and Qadeer 2014]. Such transformation, is essential when proving safety of large programs and assertions are nested deeply inside the call graph.

In the middle-end, SEAHORN is fully parametric in the semantics (e.g., small-step, big-step, *etc*) used for the generation of VCs. In addition to generating VCs based on small-step semantics [Peralta et al. 1998], SEAHORN can also automatically lift small-step semantics to large-step [Beyer et al. 2009; Gurfinkel et al. 2011] (*a.k.a.* Large Block Encoding, or LBE). The level of abstraction in the built-in semantics varies from considering only LLVM numeric registers (scalars) to considering the whole heap (modeled as a collection of non-overlapping arrays).

In the back-end, SEAHORN 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., [Gurfinkel and Chaki 2010; Albarghouthi et al. 2012; Garoche et al. 2013; Bjørner and Gurfinkel 2015]). 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.

[3]`DragonEgg` (http://dragonegg.llvm.org/) is a GCC plugin that replaces GCC's optimizers and code generators with those from LLVM. As result, the output can be LLVM bitcode.

[4]The term *mixed* semantics refers to a combination of small- with big-step operational semantics.

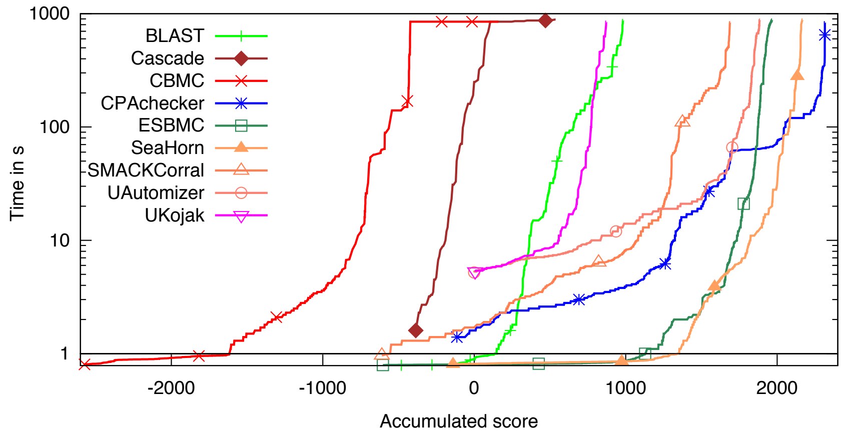

Fig. 4: Quantile graph of the results for the *Control Flow* category.

3.2. Comparative evaluation with other software verifiers

SEAHORN has participated in the International Competition of Software Verification[5] (SV-COMP 2015) [Beyer 2015]. In this competition, SEAHORN the legacy non-inter-procedural front-end. It was configured to use the large step semantics and IKOS with interval abstract domain.

Overall, SEAHORN won one gold medal in the *Simple* category – benchmarks that depend mostly on control-flow structure and integer variables – two silver medals in the categories *Device Drivers* and *Control Flow*. The former is a set of benchmarks derived from the Linux device drivers and includes a variety of C features including pointers. The latter is a set of benchmarks dependent mostly on the control-flow structure and integer variables. In the device drivers category, SEAHORN was beaten only by BLAST [Beyer et al. 2007] – a tool tuned to analyzing Linux device drivers. Specifically, BLAST got 88% of the maximum score while SEAHORN got 85%. The *Control Flow* category, was won by CPAChecker [Beyer and Keremoglu 2011] getting 74% of the maximum score, while SEAHORN got 69%. However, as can be seen in the quantile plot reported in the Figure 4, SEAHORN is significantly more efficient than most other tools solving most benchmarks much faster.

Subsequently, we have tested SEAHORN inter-procedural verification capabilities. We ran several experiments on the 215 benchmarks that we either could not verify or took more than a minute to verify in SV-COMP 2015. For example, we compared the running times with and without inlining in the front-end. Figure 5 shows a scatter plot of the running times and we see that SPACER takes less time on many benchmarks when inlining is disabled.

3.3. Evaluation on an industrial case-study

We also evaluated the SEAHORN built-in buffer overflow checks on two autopilot control software. We have used two open-source autopilot control software mnav[6] (160K LOC) and paparazzi[7] (20K LOC). Both are versatile autopilot control software for a fixed-wing aircrafts and multi-copters. Overall, SEAHORN was able to prove the ab-

[5]Detailed results can be found at http://tinyurl.com/svcomp15
[6]Micro NAV Autopilot Software available at http://sourceforge.net/projects/micronav/.
[7]Paparazzi Autopilot Software available at http://wiki.paparazziuav.org/wiki/Main_Page.

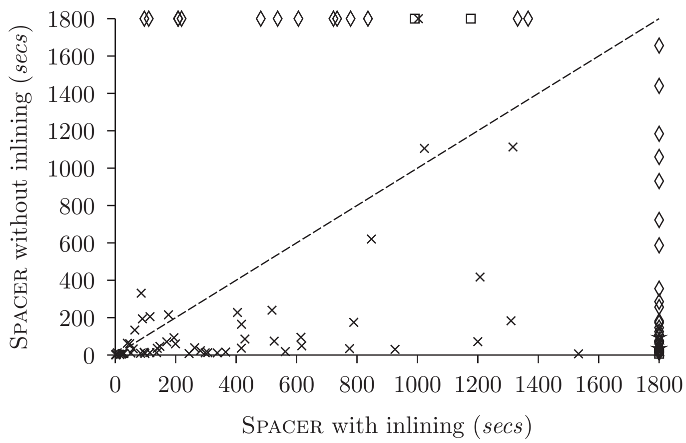

Fig. 5: A plot showing the advantage of inter-procedural (y-axis) versus intra-procedural (x-axis) encodings using SPACER back-end.

sence of buffer overflows for both benchmarks. To the best of our knowledge, this is the first time that absence of buffer overflows has been proven for mnav.

4. CONCLUSIONS

Developing new tools for automated software verification is a tedious and very difficult task. First, due to the undecidability of the problem tools must be highly tuned and engineered to provide reasonable efficiency/precision trade-offs. Second, there is a very diverse assortments of syntactic and semantic features in the different programming languages. In this article, we advocate for a design that allows the decoupling of programming language syntax and semantics from the underlying verification technique. We claim that *Constrained Horn Clauses (CHCs)* is the ideal candidate to be the intermediate formal language for software verification. CHCs are a uniform way to formally represent transition systems while allowing many different encoding styles of verification conditions. This is inline with recent trends in the software verification community and advocated by Bjørner et al. [Bjørner et al. 2012].

We also presented, SEAHORN, an LLVM-based automated verification framework. By its very nature, a verifier shares many of the complexities of an optimizing compiler and of an efficient automated theorem prover. From the compiler perspective, the issues include idiomatic syntax, parsing, intermediate representation, static analysis, and equivalence preserving program transformations. From the theorem proving perspective, the issues include verification logic, verification condition generation, synthesizes of sufficient inductive invariants, deciding satisfiability, interpolation, and consequence generation. By reducing verification to satisfiability of CHC, SEAHORN cleanly separates between compilation and verification concerns and lets us re-use many of the existing components (from LLVM and Z3). SEAHORN is a versatile and highly customizable framework that helps significantly in building new tools by allowing researchers to experiment only on their particular techniques of interest. We have shown that SEAHORN is a highly competitive verifier for safety properties both for verification benchmarks (SV-COMP) and large industrial software (autopilot code).

This is an exciting time for algorithmic software verification. The advances in the computational capabilities of hardware and maturity of verification algorithms make the technology scalable, accessible, and applicable to serious industrial applications. We believe that the line of work presented in this article provides the necessary foundations for building the next-generation verification tools, and will facilitate simpler designs and better communication of verification results between tools and their users.

5. ACKNOWLEDGMENTS

We would like to thank Anvesh Komuravelli and Nikolaj Bjørner for numerous discussions that helped shape this work.

REFERENCES

Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. 2012. Craig Interpretation. In *SAS*. 300–316.

Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. 2013. UFO: Verification with Interpolants and Abstract Interpretation - (Competition Contribution). In *TACAS*. 637–640.

Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. 2012. Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification. In *CAV*. 672–678.

Dirk Beyer. 2015. Software Verification and Verifiable Witnesses (Report on SV-COMP 2015). In *TACAS*.

Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. 2009. Software model checking via large-block encoding. In *FMCAD*. 25–32.

Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The software model checker Blast. *STTT* 9, 5-6 (2007), 505–525.

Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *CAV*. 184–190.

Nikolaj Bjørner and Arie Gurfinkel. 2015. Property Directed Polyhedral Abstraction. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015*. 263–281.

Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2012. Program Verification as Satisfiability Modulo Theories. In *SMT*. 3–11.

Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *SAS*. 105–125.

Aaron R. Bradley. 2012. IC3 and beyond: Incremental, Inductive Verification. In *CAV*. 4.

Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. 2014. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In *SEFM*. 271–277.

Maurice Bruynooghe, Gerda Janssens, Alain Callebaut, and Bart Demoen. 1987. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proceedings of the 1987 Symposium on Logic Programming, San Francisco, California, USA, August 31 - September 4, 1987*. 192–204.

Edmund M. Clarke and E. Allen Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*. 52–71.

Patrick Cousot and Radhia Cousot. 1977. 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*. 238–252.

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2014. VeriMAP: A Tool for Verifying Programs through Transformations. In *TACAS*. 568–574.

Giorgio Delzanno and Andreas Podelski. 1999. Model Checking in CLP. In *TACAS*. 223–239.

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457.

Cormac Flanagan. 2003. Automatic Software Model Checking Using CLP. In *ESOP*. 189–203.

Robert W. Floyd. 1967. Assigning meanings to programs. *Symposium Applied Mathematics* 10 (1967), 19–32.

Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2013. Failure tabled constraint logic programming by interpolation. *TPLP* 13, 4-5 (2013), 593–607.

Pierre-Loïc Garoche, Temesghen Kahsai, and Cesare Tinelli. 2013. Incremental Invariant Generation Using Logic-Based Automatic Abstract Transformers. In *NASA Formal Methods, 5th International Symposium, NFM 2013*. 139–154.

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. In *PLDI*. 405–416.

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Threader: A Constraint-Based Verifier for Multi-threaded Programs. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 412–417.

Arie Gurfinkel and Sagar Chaki. 2010. Combining predicate and numeric abstraction for software model checking. *STTT* 12, 6 (2010), 409–427.

Arie Gurfinkel, Sagar Chaki, and Samir Sapra. 2011. Efficient Predicate Abstraction of Program Summaries. In *NFM*. 131–145.

Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. 2015. SeaHorn: A Framework For Verifying C Programs - (Competition Contribution). In *To appear in TACAS*.

Arie Gurfinkel, Ou Wei, and Marsha Chechik. 2008. Model Checking Recursive Programs with Exact Predicate Abstraction. In *ATVA*. 95–110.

Kim S. Henriksen and John P. Gallagher. 2006. CHA: Convex Hull Analyser for constraint logic programs. (2006).

Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. 2003. Program Development Using Abstract Interpretation (And The Ciao System Preprocessor). In *SAS*. 127–152.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.

Krystof Hoder and Nikolaj Bjørner. 2012a. Generalized Property Directed Reachability. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. 157–171.

Krystof Hoder and Nikolaj Bjørner. 2012b. Generalized Property Directed Reachability. In *SAT*. 157–171.

Joxan Jaffar and Jean-Louis Lassez. 1987. Constraint Logic Programming. In *POPL*. 111–119.

Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. 2012. TRACER: A Symbolic Execution Tool for Verification. In *CAV*. 758–766.

Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. 2004. A CLP Proof Method for Timed Automata. In *RTSS*. 175–186.

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *CAV*. 17–34.

Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. 2013. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *CAV*. 846–862.

Akash Lal and Shaz Qadeer. 2014. A program transformation for faster goal-directed search. In *FMCAD*. 147–154.

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. 75–88.

Kim. Marriott and Peter. J. Stuckey. 1998. *Introduction to Constraint Logic Programming*. MIT Press, Cambridge, MA, USA.

John McCarthy. 1963. A Basis for a Mathematical Theory of Computation. (1963), 33–70.

Ken McMillan and Andrey Rybalchenko. 2013. *Solving Constrained Horn Clauses using Interpolation*. Technical Report. MSR-TR-2013-6.

Mario Méndez-Lojo, Jorge A. Navas, and Manuel V. Hermenegildo. 2007. A Flexible, (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *LOPSTR*. 154–168.

Kalyan Muthukumar and Manuel V. Hermenegildo. 1989. Determination of Variable Dependence Information through Abstract Interpretation. In *Logic Programming, Proceedings of the North American Conference*. 166–185.

Peter Naur. 1966. Proof of algorithms by general snapshots. 6 (1966), 310–316. Issue 4.

Julio C. Peralta, John P. Gallagher, and Hüseyin Saglam. 1998. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In *SAS*. 246–261.

Jean-Pierre Queille and Joseph Sifakis. 1982. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*. 337–351.

Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2013. Disjunctive Interpolants for Horn-Clause Verification. In *CAV*. 347–363.

Harald Søndergaard. 1986. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. In *ESOP*. 327–338.

Alan Turing. 1949. Checking a Large Routine. (1949).

Richard Warren, Manuel V. Hermenegildo, and Saumya K. Debray. 1988. On the Practicality of Global Flow Analysis of Logic Programs. In *ICLP*. 684–699.