InterpChecker: Reducing State Space via Interpolations
(Competition Contribution)

Zhao Duan\textsuperscript{1}, Cong Tian\textsuperscript{1(\textcopyright)}, Zhenhua Duan\textsuperscript{1}, and C.-H. Luke Ong\textsuperscript{2}
\textsuperscript{1} ICTT and ISN Lab, Xidian University, Xi'an 710071, People's Republic of China
ctian@mail.xidian.edu.cn
\textsuperscript{2} Department of Computer Science, University of Oxford, Oxford, UK

Abstract. InterpChecker is a tool for verifying safety properties of C programs. It reduces the state space of programs throughout the verification via two new kinds of interpolations and associated optimization strategies. The implementation builds on the open-source, configurable software verification tool, CPAChecker.

1 Verification Approach

Our approach to scalable CEGAR-based model checking is to exploit Craig interpolation \cite{3} to learn abstractions that can systematically reduce the program state space which must be explored for a given safety verification problem. In addition to the interpolants for parsimonious abstraction \cite{4} (called reachability interpolants (R-Interp) here for clarity), we introduce two new kinds of interpolants, called universal safety interpolants and existential error interpolants.

– A universal safety interpolant (or \textit{S-Interp}) is useful for determining whether all the paths emanating from a state are safe, without exploring all the possible branches from it.

– An existential error interpolant (or \textit{E-Interp}) is useful for determining whether there exists an unsafe path emanating from a state, without exploring all the possible branches from it.

The \textit{S-Interp} at a location of a control flow graph (CFG) collects predicates that are relevant to a yes-instance of the safety verification, so that whenever the \textit{S-Interp} is implied by the current path, all paths emanating from this location are guaranteed to be safe. Dually, whenever the \textit{E-Interp} at a location of a CFG is implied by the current path, there is an unsafe branch from it, and so, one can immediately conclude that the program is unsafe. We learn \textit{S-Interp} and
**E-Interp** from spurious error traces and apply them to reduce the state space of programs throughout the CEGAR-based program verification process. For convenience, we denote a CFG as a tuple $G = (L, T, l_0, f)$, where $L$ is the set of program locations, $l_0 \in L$ is the initial location, $f \in L$ is the final location, $T \subseteq L \times \text{Ops} \times L$ is the transition relation, and $\text{Ops}$ is the set of instructions.

When verifying a program, we first unwind the CFG to generate an Abstract Reachability Tree (ART). An ART $A = (S_A, E_A)$, obtained from a CFG $G = (L, T, l_0, f)$, consists of a set $S_A$ of abstract states and a set $E_A$ of edges. An abstract state $s \in S_A$ is a triple $s = (l, c, p)$ where $l$ is a location in the CFG, $c$ is the current call stack, and $p$ is an abstract predicate indicating the reachable region of the current state which is determined by the reachable interpolant, $R$-Interp. Given two states $s$ and $s'$, we say $s$ is covered by $s'$ just if $s[0] = s'[0]$, $s[1] = s'[1]$, and $s[2] \rightarrow s'[2]$. (Notation: for a tuple $e$, we write $e[i]$ for the $i$-th component of $e$.) Further, if $s$ is covered by $s'$ and all the future of $s'$ (i.e. all abstract states reachable from $s'$) has been explored, then it is safe to not explore the future of $s$. A branch (path) $\Pi$ of an ART, denoting a possible execution of the program, is a finite alternating sequence of states and edges, $\Pi = \langle s_0, e_0, \cdots, e_n−1, s_n \rangle$, such that for all $0 \leq i < n$, $e_i[0] = s_i$ and $e_i[2] = s_{i+1}$. Given a path $\Pi$ of an ART, we write $P_f(\Pi)$ for the path formula $\text{ssa}(e_0[1]) \land \cdots \land \text{ssa}(e_{n−1}[1])$ obtained from $\Pi$. Here $\text{ssa}(op)$ is the static single assignment (SSA) of an operation $op$ where every variable occurring in $\Pi$ is assigned a value at most once.

Given a CFG whose locations are enriched with default values of $R$-Interp, $S$-Interp, and $E$-Interp, we construct the ART for exploring a real counterexample by starting from the root, i.e. $s_0 : (l_0, -, \text{true})$. The flowchart in Fig. 1 gives a bird’s eye view of our approach to safety verification with reachability, safety and error interpolations. When a state $s : (l, c, p)$ is being explored and $l$ is not an error location:

1. Reversely traverse the current path for other possibilities if one of the following three conditions holds:
   - $p = \text{false}$;
   - $p \neq \text{false}$, $F(l) = f$, and $P_f(s_0, \cdots, s) \rightarrow I_s(l)$;
   - $p \neq \text{false}$ and $s$ is covered by a visited state $s'$.

2. Report the program is unsafe, if $p \neq \text{false}$ and $P_f(s_0, \cdots, s) \rightarrow E$-Interp($l$).
3. Explore the succeeding state $s'' : (\text{suc}(l), c, p')$, otherwise.

When $l$ of the current state $s : (l, c, p)$ is an error location, we first check whether the current path $\Pi = \langle s_0, \cdots, s \rangle$ is spurious. If $\Pi$ is not spurious, we conclude that the program is unsafe. Otherwise, by update $S$-Interp, update $E$-Interp, and update $R$-Interp [5], the $S$-Interp, $E$-Interp, and $R$-Interp of locations involved in $\Pi$ are updated, respectively. Subsequently, we reversely track the current path for other possibilities and treat a new current state $s : (l, c, p)$ in the same way until the program is reported as unsafe or there are no more states to be explored.

To maximise the effect of the proposed interpolations, we also present two kinds of optimizing strategies: pruning CFG and weight-guided search. In real-world
programs, there may exist some locations in a CFG which can never reach any error location. To avoid exploring these locations when verifying the program, the first strategy is to prune the CFG by removing these locations and the relative control flow edges in advance. A safety interpolant works only when it is full. Hence, the earlier full safety-interpolants are formed, the more effective the performance will be. To form full interpolants, the second strategy is to explore one side of a branch as early as possible if the other side has been explored. The goal is achieved by introducing an attribute *weight* to transitions of a CFG. Throughout the verification, the branch with the largest weight will be explored first.

### 2 Software Architecture

Our implementation of InterpChecker builds on the open-source, configurable software verification tool, CPAChecker [1]. Like CPAChecker, InterpChecker can verify safety properties of C program via reachability checking of the instrumented error labels. All extra functions are implemented in Java, using the existing libraries provided by CPAChecker. In Fig. 1, the white parts are new, while the grey parts are original CPAChecker functions. We set up the InterpChecker interpolants and optimizations as an option of CPAChecker, organised as a refinement-selection configuration, in the sense of [2].

### 3 Strengths and Weakness

The new interpolants implemented in InterpChecker do not affect the existing configurations of CPAChecker. InterpChecker supports the verification of safety properties of C program via reachability checking of the instrumented error labels. The power of InterpChecker is best illustrated when analysing large-scale programs because it can avoid exploring more paths. The current version does
not support the verification of the properties written as temporal logic formulas. Like CPAChecker, we skip recursive functions and treat them as pure functions. Thus, false negatives may occur for programs with recursive functions.

4 Tool Setup and Configuration

A zipped file containing InterpChecker 1.0 is available at http://github.com/duanzhao-dz/interpchecker. It contains all the required libraries: no installation of external tools is required. To run InterpChecker, first download the code from the website, then run the following command to install the package in Ubuntu 16.04: sudo apt-get install openjdk-8-jdk.

To process a benchmark example test.c, invoke the script by the following command: ./scripts/cpa.sh -sv-comp18-interpcpachecker test.c. The output of InterpChecker is written to the file output/Statistics.txt. When using BenchExec, the output can be translated by the interpchecker.py tool-info module. The categories verified by the competition candidate are listed in the file interpchecker.xml. The two files are contained in the zipped file. If the checked property does not hold, a human readable counterexample is written to output/ErrorPath.txt and an error witness is written to the zipped file witness.graphml.gz. Note that Java Runtime Environment is required, which should be at least Java 8 compatible.

5 Software Project and Contributors

Based on the open source tool CPAChecker, InterpChecker is developed by Xidian University, China, and the University of Oxford, UK. We thank Dirk Beyer and his team for their original contributions to CPAChecker.

References

1. Beyer, D., Keremoglu, M.E.: CPACHECKER: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
2. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 20–38. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_3
3. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig interpretation. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 300–316. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33125-1_21
4. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proceedings of POPL, pp. 232–244 (2004)
5. Tian, C., Duan, Z., Duan, Z.H., Ong, L.: More effective interpolations in software model checking. In: The 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pp. 183–193. IEEE (2017)
