UFO: A Framework for Abstraction- and Interpolation-Based Software Verification

Aws Albarghouthi[1], Yi Li[1], Arie Gurfinkel[2], and Marsha Chechik[1]

[1]Department of Computer Science, University of Toronto, Canada
[2]Software Engineering Institute, Carnegie Mellon University, USA

Abstract. In this paper, we present UFO, a framework and a tool for verifying (and finding bugs in) sequential C programs. The framework is built on top of the LLVM compiler infrastructure and is targeted at researchers designing and experimenting with verification algorithms. It allows definition of different abstract post operators, refinement strategies and exploration strategies. We have built three instantiations of the framework: a predicate abstraction-based version, an interpolation-based version, and a combined version which uses a novel and powerful combination of interpolation-based and predicate abstraction-based algorithms.

1 Introduction

Software model checking tools prove that programs satisfy a given safety property by computing inductive invariants that preclude erroneous program states. Over the past decade, software model checking tools have adopted a number of different techniques for computing invariants which we categorize as *Over-approximation-Driven* (OD) and *Under-approximation-driven* (UD).

OD tools, e.g., SLAM [4], BLAST [6], and SATABS [10], utilize an abstract domain based on predicate abstraction [11] to compute an over-approximation of the set of reachable states of a program. In the case of false positives, such techniques employ an abstraction refinement loop [9] to refine the abstract domain and eliminate false positives.

UD tools, spearheaded by IMPACT [15] and YOGI [17], compute invariants by generalizing from infeasible symbolic program paths, thus bypassing the potentially expensive computation of the abstract post operator. For example, IMPACT and WOLVERINE [13] use Craig interpolants, extracted from the proofs of unsatisfiability of formulas encoding an infeasible path to error, in order to eliminate a potentially large number of paths to error and prove a program safe. WHALE [2] extends IMPACT to the interprocedural case by using under-approximations of functions to compute function summaries. Similarly, YOGI uses weakest-preconditions (instead of interpolants) along infeasible program paths, chosen based on concrete test-case executions, in order to strengthen a partition-graph of the state space of a program.

In this paper, we present UFO, a framework and a tool for *verifying* and *falsifying* safety properties of sequential C programs. The features of UFO are:

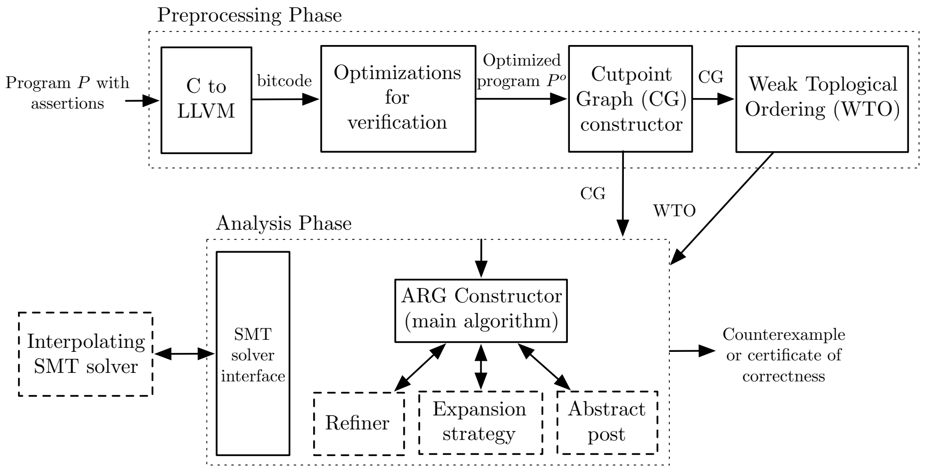
Preprocessing Phase
Program P with assertions
C to LLVM
bitcode
Optimizations for verification
Optimized program P^o
Cutpoint Graph (CG) constructor
CG
Weak Toplogical Ordering (WTO)
CG
WTO
Analysis Phase
Interpolating SMT solver
SMT solver interface
ARG Constructor (main algorithm)
Refiner
Expansion strategy
Abstract post
Counterexample or certificate of correctness

Fig. 1. The architecture of UFO.

1. It is a framework for building and experimenting with UD, OD, and combined UD/OD verification algorithms. It is *parameterized* by the abstract post operator, refinement strategy, and expansion strategy.

2. It comes with a number of instantiations whose novel features are described in [1]: several predicate abstraction-based OD instantiations, an interpolation-based UD instantiation, and several combined OD/UD instantiations that use different forms of predicate abstraction to augment and strengthen interpolation-based analysis, a technique that we have shown to be quite powerful in [1]. To the best of our knowledge, UFO is the first available tool to implement a combined UD/OD algorithm. Moreover, these instantiation of UFO implement a novel interpolation-based refinement strategy that computes interpolants (abstractions) for multiple program paths encoded in a single SMT formula. That is, unlike other tools that enumerate paths explicitly, e.g., [15, 13, 6], UFO delegates path enumeration to an SMT solver.

3. It is implemented on top of the open-source LLVM compiler infrastructure [14]. Since LLVM is a well-maintained, well-documented, and continuously improving framework, it allows UFO users to easily integrate program analyses, transformations, and other tools built on LLVM (e.g., KLEE [8]), as they become available. Furthermore, since UFO analyzes LLVM bitcode (intermediate language), it is possible to experiment with verifying programs written in other languages compilable to LLVM bitcode, such as C++, Ada, and Fortran.

The architecture and parameterization of UFO and the underlying LLVM framework provide users with an extensible environment for experimenting with different software verification algorithms.

UFO is available at `http://www.cs.toronto.edu/~aws/ufo`.

2 The Implementation and Architecture of UFO

UFO is implemented on top of the LLVM compiler infrastructure [14] – see Figure 1 for an architectural overview. UFO accepts as input a C program P with assertions. For simplicity of presentation, let $P = (V, T, \phi_{\mathcal{I}}, \phi_{\mathcal{E}})$, where V is the set of program variables, T is the transition relation of the program (over V and V', the set of primed variables), $\phi_{\mathcal{I}}$ is a formula describing the set of initial states, and $\phi_{\mathcal{E}}$ is a formula describing the set of error states.

First, P goes through a *preprocessing phase* where it is compiled into LLVM bitcode (intermediate representation) and optimized for verification purposes, resulting in a semantically equivalent but optimized program $P^o = (V^o, T^o, \phi^o_{\mathcal{I}}, \phi^o_{\mathcal{E}})$.

Then, the *analysis phase* verifies P^o and either outputs a certificate of correctness or a counterexample. A certificate of correctness for P^o is a safe inductive invariant I s.t. (1) $\phi^o_{\mathcal{I}} \Rightarrow I$, (2) $I \wedge T^o \Rightarrow I'$, and (3) $I \wedge \phi^o_{\mathcal{E}}$ is UNSAT.

2.1 Preprocessing Phase

We now describe the various components of the preprocessing phase.

C to LLVM. The first step converts the program P to LLVM bitcode using the `llvm-gcc` or `clang` compilers.

Optimizations for Verification. A number of native LLVM optimizations are then applied to the bitcode, the most important of which are *function inlining* (`inline`) and *static single assignment (SSA) conversion* (`mem2reg`). Since UFO implements an intraprocedural analysis, it requires all functions to be inlined into `main`. In order to exploit efficient SMT program encoding techniques like [12], UFO expects the program to be in SSA form. A number of standard program simplifications are also performed at this stage, with the goal of simplifying verification. The final result is the optimized program P^o. Mapping counterexamples from P^o back to the original C program P is made possible by the debugging information inserted into the generated bitcode by `clang`.

```
int x;
if (x == 0)
  func1();
if (x != 0)
  func2();
return 1;
```

Fig. 2. Example program.

Before the above optimizations could be applied, we had to bridge the gap between the semantics of C assumed by LLVM (built for compiler construction) and the verification benchmarks. Consider, for example, the program in Figure 2. After LLVM optimizations, it is reduced to the empty program: `return 1;`. LLVM replaces undefined values by constants that result in the simplest possible program. In our example, the conditions of both `if`-statements are assigned to `0`, even though they contradict each other. On the other hand, verification benchmarks such as [5] assume that without an explicit initialization, the value of `x` is non-deterministic. To account for such semantic differences, a UFO-specific LLVM transformation is scheduled before optimizations are run. It initializes each variable with a call to an external function `nondet()`, forcing LLVM not to make assumptions about its value.

Cutpoint Graph and Weak Topological Ordering. A *cutpoint graph* (CG) is a "summarized" control-flow graph (CFG), where each node represents a cutpoint (loop head) and each edge represents a loop-free path through the CFG. Computed using the technique presented in [12], the CG is used as the main representation of the program P^o. Using it allows us to perform abstract post operations on loop-free segments, utilizing the SMT solver (e.g., in the case of predicate abstraction) for enumerating a potentially exponential number of paths. A *weak topological ordering* (WTO) [7] is an ordering of the nodes of the CG that enables exploring it with a *recursive iteration strategy*: starting with the inner-most loops and ending with the outer-most ones.

2.2 Analysis Phase

The analysis phase, which receives the CG and the WTO of P^o from the preprocessing phase, is comprised of the following components:

ARG Constructor. The *ARG Constructor* is the main driver of the analysis. It maintains an *abstract reachability graph* (ARG) [1] of the CG – an unrolling of the CG, annotated with formulas representing over-approximations of reachable states at each cutpoint. ARGs can be seen as DAG representations of *abstract reachability trees* (ARTs) used in lazy abstraction [15, 6]. When the algorithm terminates without finding a counterexample, the annotated ARG represents a certificate of correctness in the form of a safe inductive invariant I for P^o. To compute annotations for the ARG, the ARG constructor uses three *parameterized components*: (1) the abstract post, to annotate the ARG as it is being expanded; (2) the refiner, to compute annotations that eliminate spurious counterexamples; and (3) the expansion strategy, to decide where to restart expanding the ARG after refinement.
Abstract Post. The abstract post component takes a CG edge and a formula ϕ_{pre} describing a set of states, and returns a formula ϕ_{post} over-approximating the states reachable from ϕ_{pre} after executing the CG edge. UFO includes two common implementations of abstract post – Boolean and Cartesian predicate abstractions [3].
Refiner. The refiner receives the current ARG with potential paths to an error location (i.e., the error location is not annotated with *false*). Its goal is either to find a new annotation for the ARG s.t. the error location is annotated with *false*, or to report a counterexample. UFO includes an interpolation-based implementation of the refiner.
Expansion Strategy. After the refinement, the ARG constructor needs to decide where to restart expanding the ARG. The *expansion strategy* specifies this parameter. UFO includes an eager strategy and a lazy strategy, both of which are described in the following section.

SMT Solver Interface. Components of the analysis phase use an SMT solver in a variety of ways: (1) The ARG constructor uses it to check that the annotations of the ARG form a safe inductive invariant; (2) abstract post, e.g., using predicate abstraction, encode post computations as SMT queries, and (3) the refiner can use it to find counterexamples and to compute interpolants. All these uses are handled through a general interface to two SMT solvers: MathSAT5[1] (used for interpolation) and Z3 [16] (used for quantifier elimination).

3 Instantiations of Ufo

We have implemented the three instantiations of the algorithm of [1] in Ufo: (1) an interpolation-based UD instantiation, (2) a predicate abstraction-based OD instantiation, and (3) a combined OD/UD instantiation that uses predicate abstraction to augment the interpolation-based analysis. In [1], we showed that the combined instantiation can outperform both the UD and the OD instantiations. All of these instantiations use a novel interpolation-based refinement strategy where all paths in the ARG are encoded as a single SMT solver formula, delegating path enumeration to the SMT solver instead of enumerating them explicitly as done by Impact [15] and Yogi [17].

We now show how these instantiations are produced by defining the three Ufo parameters: abstract post, refiner, and expansion strategy.

UD Instantiation. In the UD case, the abstract post always returns *true*, the weakest possible over-approximation. The annotations returned by the refiner are used as for the ARG; therefore, they can be seen as a guess of the safe inductive invariant I. If the guess does not hold, i.e., it is not inductive, then the *lazy* expansion strategy starts expanding the ARG from the inner-most loop where the guess fails [1]. The ARG is then extended and a new guess for I is made by the refiner.

OD Instantiation. In the OD case, the abstract post is based on either Boolean or Cartesian predicate abstraction. The annotations returned by the refiner are used to update the set of predicates but not to guess invariants (and thus annotate the ARG) as in the UD case. The expansion strategy used is *eager*: expansion is restarted from the root of the ARG, i.e., in each iteration Ufo computes an abstract post fixpoint from the initial states $\phi_{\mathcal{I}}$, but with a larger set of predicates from the one used in the previous iteration.

Combined UD/OD Instantiation. In the combined UD/OD case, Ufo uses Boolean or Cartesian predicate abstraction [3] to improve guesses of I found through interpolants. In each iteration, Ufo starts with a guess I, that does not hold, from the previous iteration. A new set of states I_p, where $I \Rightarrow I_p$, is computed by applying an abstract fixpoint computation, based on predicate abstraction, starting from the set of states I and using the transition relation T. Technically, this is performed by expanding the ARG where the guess I fails (as in the UD case). If I_p is not a safe inductive invariant, a new guess is computed using interpolants, and the process is restarted. The trade-off in this case is between the potential for computing invariants in fewer refinements (guesses) using predicate abstraction and the potentially high cost of predicate abstraction computations.

[1] http://mathsat.fbk.eu

4 Conclusion

In this paper, we have described UFO, a framework and a tool for software verification of sequential C programs. As we have shown, by varying the parameters of UFO, it can be instantiated into tools employing varying verification techniques, including an interpolation-based tool, a predicate abstraction-based one, and a tool that combines the two techniques.

UFO's architecture and the fact that is built on top of LLVM provide verification algorithm designers with a flexible and extensible platform to experiment with a wide variety of verification algorithms.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: "From Under-approximations to Over-approximations and Back". In: Proc. of TACAS'12 (2012)
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: "Whale: An Interpolation-based Algorithm for Inter-procedural Verification". In: Proc. of VMCAI'12 (2012)
3. Ball, T., Podelski, A., Rajamani, S.: "Boolean and Cartesian Abstraction for Model Checking C Programs". In: Proc. of TACAS'01. vol. 2031, pp. 268–283 (2001)
4. Ball, T., Rajamani, S.: "The SLAM Toolkit". In: Proc. of CAV'01. LNCS, vol. 2102, pp. 260–264 (2001)
5. Beyer, D.: "Competition On Software Verification" (2012), `http://sv-comp.sosy-lab.org/`
6. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: "The Software Model Checker BLAST". STTT 9(5-6), 505–525 (2007)
7. Bourdoncle, F.A.: "Efficient Chaotic Iteration Strategies with Widenings". In: Proc. of FMPA'93. pp. 128–141. LNCS (1993)
8. Cadar, C., Dunbar, D., Engler, D.: "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: Proc. of OSDI'08. pp. 209–224 (2008)
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: "Counterexample-Guided Abstraction Refinement". In: Proc. of CAV'00. LNCS, vol. 1855, pp. 154–169 (2000)
10. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: "SATABS: SAT-based Predicate Abstraction for ANSI-C". In: Proc. of TACAS'05. LNCS, vol. 3440, pp. 570–574 (2005)
11. Graf, S., Saïdi, H.: "Construction of Abstract State Graphs with PVS". In: Proc. of CAV'97. vol. 1254, pp. 72–83 (1997)
12. Gurfinkel, A., Chaki, S., Sapra, S.: "Efficient Predicate Abstraction of Program Summaries". In: Proc. of NFM'11. LNCS, vol. 6617, pp. 131–145 (2011)
13. Kroening, D., Weissenbacher, G.: "Interpolation-Based Software Verification with Wolverine". In: Proc. of CAV'11. LNCS, vol. 6806, pp. 573–578 (2011)
14. Lattner, C., Adve, V.: "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: CGO'04 (2004)
15. McMillan, K.L.: "Lazy Abstraction with Interpolants". In: Proc. of CAV'06. LNCS, vol. 4144, pp. 123–136 (2006)
16. de Moura, L., Bjørner, N.: "Z3: An Efficient SMT Solver". In: Proc. of TACAS'08. LNCS, vol. 4963, pp. 337–340 (2008)
17. Nori, A., Rajamani, S., Tetali, S., Thakur, A.: "The YOGI Project: Software Property Checking via Static Analysis and Testing". In: Proc. of TACAS'09. LNCS, vol. 5505, pp. 178–181 (2009)