UFO: Verification with Interpolants and Abstract Interpretation (Competition Contribution)

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

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

1 Verification Approach

The algorithms underlying UFO are described in [1–3]. The UFO tool is described in more detail in [4].

UFO marries the power and efficiency of numerical Abstract Interpretation (AI) domains [6] with the generalizing ability of interpolation-based software verification in an abstraction refinement loop. More formally: given a program P, a safety property φ, and some abstract domain $\mathcal{A}$, UFO starts by computing an inductive invariant $\mathcal{I}$ of P in $\mathcal{A}$. If $\mathcal{I} \Rightarrow \varphi$, then we know that P satisfies φ, i.e., P cannot reach any of the error states characterized by $\neg\varphi$. Otherwise, if $\mathcal{I} \not\Rightarrow \varphi$, UFO uses SMT solving to check whether the alarm raised by $\mathcal{I}$ maps to a real bug in the code. To do so, UFO encodes all of the program paths explored by abstract interpretation as a formula, and uses an SMT solver to check its satisfiability. If the formula is satisfiable, an erroneous execution is reported to the user. Otherwise, an interpolation technique guided by the results of AI is used to strengthen $\mathcal{I}$ into $\mathcal{I}'$, where $\mathcal{I}' \Rightarrow \varphi$. If $\mathcal{I}'$ is no longer inductive, abstract interpretation continues from the set of states described by $\mathcal{I}'$. Otherwise, the program is safe.

2 Software Architecture

UFO is implemented in C++ in the LLVM compiler infrastructure [7] as a general verification framework. Its architecture is shown in Fig. 1. In what follows, we describe our instantiation of the framework for the purposes of the competition.
Preprocessing Phase. The first step in this phase is converting a given program into the LLVM intermediate representation. Following that, we perform compiler optimizations and preprocessing in order to simplify the verification process. As a preprocessing step, we initialize uninitialized variables using non-deterministic functions. This is used to bridge the gap between the verification semantics (which assume a non-determinsitic assignment) and compiler semantics, which presets unitialized variables with the goal of optimizing the code. For optimizations, we perform a number of program simplifications such as function inlining, converting the program into the static single assignment (SSA) form by reducing memory operations into SSA registers, removing dead code, etc.

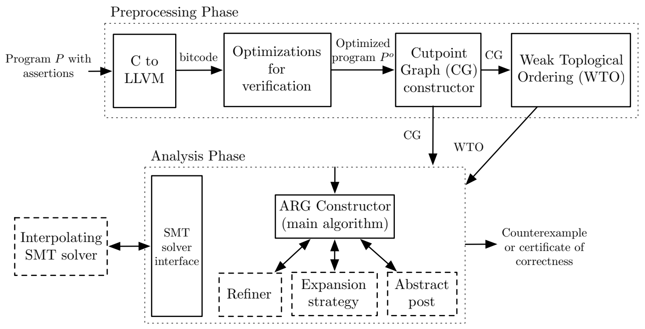
Preprocessing Phase
Program P with assertions
C to LLVM
bitcode
Optimizations for verification
Optimized program P^o
Cutpoint Graph (CG) constructor
CG
Weak Toplogical Ordering (WTO)
CG
WTO
Analysis Phase
Interpolating SMT solver
SMT solver interface
ARG Constructor (main algorithm)
Refiner
Expansion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

Fig. 1: The architecture of UFO [4].

After the optimization step, we represent the program as a Cutpoint Graph (CG), a control-flow graph where each node is a cutpoint in the original program and each edge is a loop-free execution between two cutpoints. Then, a Weak Topological Ordering (WTO) [5] is computed for the CG and used later as the abstract interpretation strategy.

Analysis Phase. The main algorithm constructs an Abtract Reachability Graph (ARG), a labelled unrolling of the program that represents an inductive invariant using a given abstract domain. The ARG contructor is parameterized by the abstract domain used and the refinement strategy:

– *Abstract domains*: The abstract domains we use are Box (intervals), Boxes [6] (intervals with disjunctions), and Cartesian and Boolean predicate abstraction. Our experiments have shown that different domains are useful for different problems and there is no clear winner. Thus, for the purposes of the competition, we instrumented Ufo to run multiple analysis instances with different domains in parallel, reporting the results of the fastest instance.
– *Refinement*: As a refinement strategy, we used AI-guided DAG interpolants from [1]. DAG interpolants annotate a directed acyclic graph of paths using a single call to an SMT solver, delegating the process of path enumeration to the SMT solver. In comparison, other techniques, e.g., Impact [8], unroll the program into a tree, potentially having to refine exponentially many paths in the size of the program. Furthermore, our refinement strategy uses the invariant computed by abstract interpretation in the encoding, often resulting in weaker interpolants and faster SMT solving time.

The Z3 [9] SMT solver is used for satisfiability checking, and MathSAT5[1] for computing interpolants. Due to the efficiency of Z3, we use it to shrink an interpolation query by computing an UNSAT core of a formula before handing it to MathSAT5 for satisfiability checking and interpolation.

The analysis phase results in either `SAFE` – a safe inductive invariant has been computed, `CEX` – a counterexample has been found, or `UNKNOWN` – implying that Ufo failed to produce a conclusive result. Counterexamples are produced as traces over basic blocks in the LLVM intermediate representation of the program.

[1] `mathsat.fbk.eu`

3 Strengths and Weaknesses

UFO has been succesfully applied to `ControlFlowIntegers`, `SystemC`, `DeviceDrivers64`, and `ProductLines`. Currently, UFO uses linear arithmetic to model semantics of sequential C programs, making it imprecise for categories such as `BitVectors` (requiring bit-level precision), `HeapManipulation` (requiring heap tracking), and `Concurrency` (requiring thread handling). Another weakness is UFO's reliance on multiple tools for the front-end: LLVM 2.6, LLVM 2.9, and CIL. This increases the trusted computing base and makes it harder to maintain.

The power of UFO lies in its parameterized nature, allowing instantiations with different abstract domains and providing a general framework for experimenting with verification algorithms.

Tool Setup and Configuration. UFO is available for download from `bitbucket.org/arieg/ufo/wiki/svcomp13.wiki`. The options for running the tool are:

```
./bin/ufo-svcomp-par.py [-m64] --cex=FILE input
```

where `-m64` turns on 64-bit model, `--cex` is the location of the counter-example, and `input` is a C file.

Acknowledgment. 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 FFRDC[2]. We also acknowledge financial support provided by NSERC, NECSIS and Alexander Graham Bell Scholarship.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig Interpretation. In: SAS'12. LNCS, vol. 7460, pp. 300–316 (2012)
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From Under-approximations to Over-approximations and Back. In: TACAS'12. LNCS, vol. 7214, pp. 157–173 (2012)
3. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Whale: An Interpolation-based Algorithm for Inter-procedural Verification. In: VMCAI'12. vol. 7148, pp. 39–55 (2012)
4. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: A Framework for Abstraction- and Interpolation-Based Software Verification. In: CAV'12. LNCS, vol. 7358, pp. 672–678 (2012)
5. Bourdoncle, F.A.: Efficient Chaotic Iteration Strategies with Widenings. In: FMPA'93. pp. 128–141. LNCS (1993)
6. Gurfinkel, A., Chaki, S.: Boxes: A Symbolic Abstract Domain of Boxes. In: SAS'10. LNCS, vol. 6337, pp. 287–303 (2010)
7. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO'04. pp. 75–88 (2004)
8. McMillan, K.L.: Lazy Abstraction with Interpolants. In: CAV'06. LNCS, vol. 4144, pp. 123–136 (2006)
9. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS'08. LNCS, vol. 4963, pp. 337–340 (2008)

[2] NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. This material has been approved for public release and unlimited distribution. (DM-0000076)