Call-Graph-Guided Verification

Lauren Pick (lpick@princeton.edu)
Princeton UniversIty

Modular Verification

Goal: Infer procedure summaries that are…

- sufficient for verification
- efficiently computable
- sufficiently abstract and relevant to reason about the whole program

Challenges

- How should procedures be explored and in what environments (if any) should a procedure call be considered?
- How do we ensure procedure relevance *and* scalability?
- How would mutual recursion be handled?

Call Graph and Bounded Environments

Exploring Procedures and Environments

- Consider each procedure in an *environment* to learn over- and under-approximate summaries
- Environments represent possible counterexample paths
- Choose a finite path through the call graph:
  - Final call is the *target procedure* to consider
  - The rest of the procedures in the path make up the *environment*

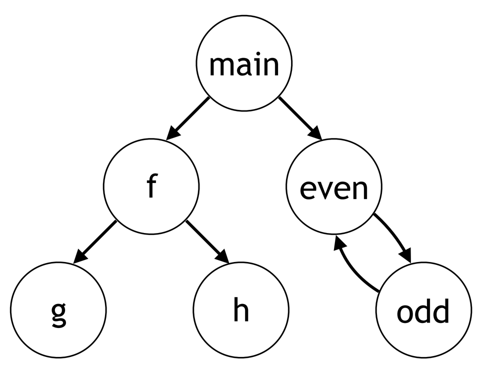
Scalability: Use Bounded Environments

- Longer call paths lead to larger queries and poor scalability
- Achieve scalability by *approximating* the environments
- A *b-bounded environment* captures at most the body of *b* procedures above the target procedure in the call graph path

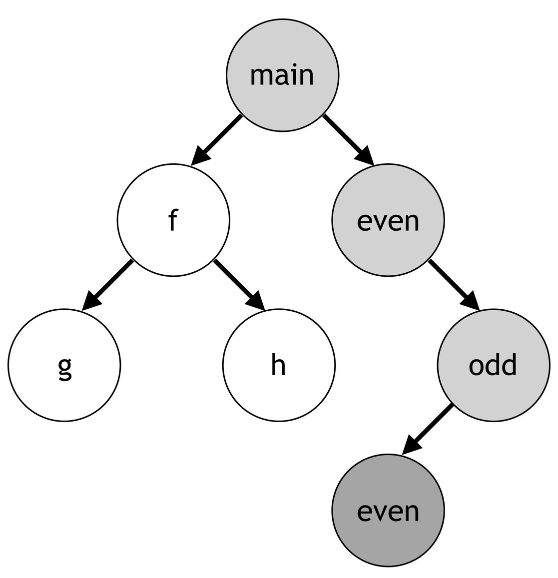
```
void main() {
  assert not(even(f() - 1));
}

int f() {
  return h(g());
}

int g() {
  return 2*havoc() + 1;
}

bool h(x) {
  return x + 1;
}
```

```
bool even(x) {
  assume (x >= 0);
  if (x==0) return true;
  else return odd(x - 1);
}

bool odd(x) {
  assume (x >= 1);
  if (x == 1) return true;
  return even(x - 1);
}
```

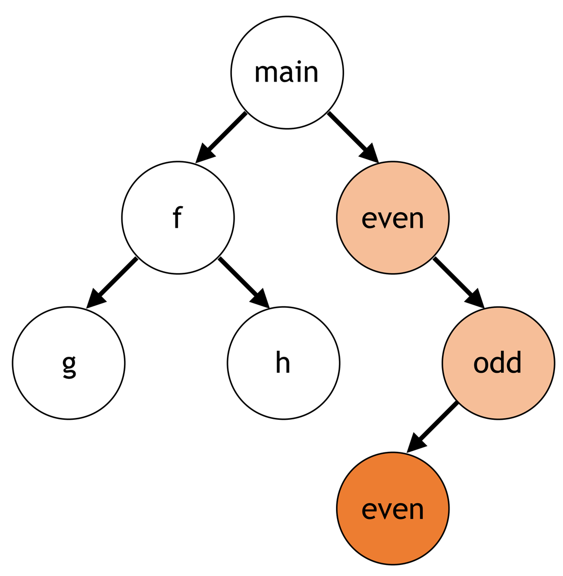
Example Program



Example Program Call



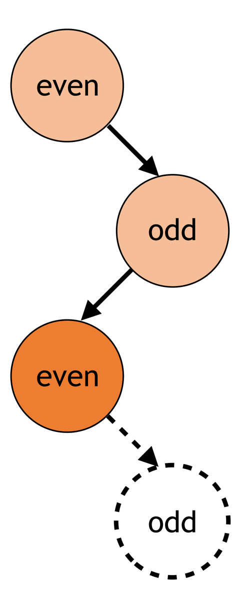
Full Environment (lighter) and Target Procedure



2-Bounded Environment (lighter) and Target Procedure

Mutual Recursion

- Handle recursion by performing *explicit induction*
- Handle mutual recursion by performing induction *under assumptions*
- Choice of assumptions: assume the negation of the (bounded) environment of procedures above the target
- Learn *implications among procedure summaries* — we call these Environment-Call (EC) lemmas



assume: odd.out ⇔ (1 + odd.in) mod 2 = 0

prove: even.out ⇔ even.in mod 2 = 0

Learning Procedure Summaries

Update Target's Under-Approximate Summary

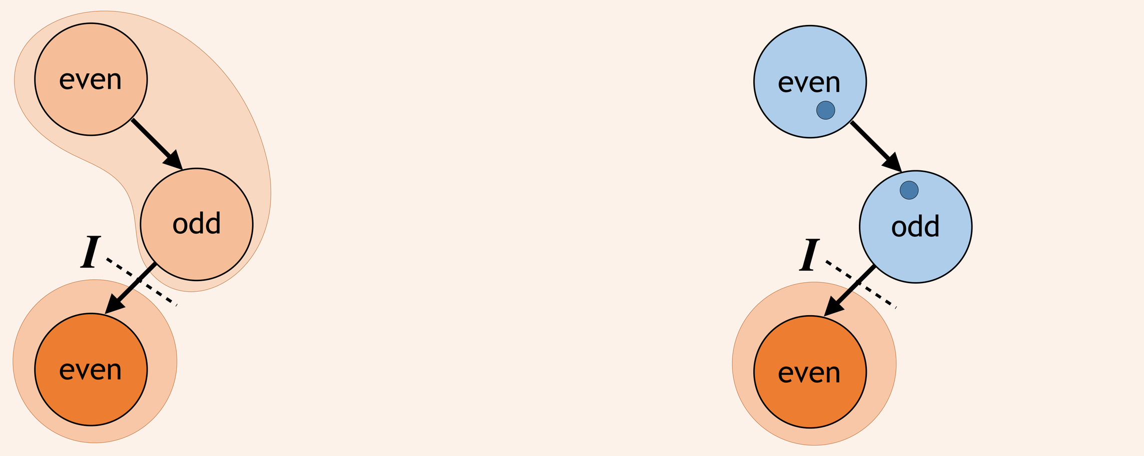
- Perform SMT check for under-approximation of procedure body and over-/under-approximation of environment
- If satisfiable, learn potential counterexample behaviors in target

Over-approximate Environment
Under-Approximate Procedure

Under-approximate Environment
Under-Approximate Procedure

Update Target's Over-Approximate Summary

- Perform SMT check for over-approximation of procedure body and over-/under-approximation of environment
- If unsatisfiable
  - find **interpolant** $\boldsymbol{I}$ that separates the target and environment
  - learn interpolant as over-approximation
- **With induction** to handle recursion
- **With assumptions** to handle mutual recursion



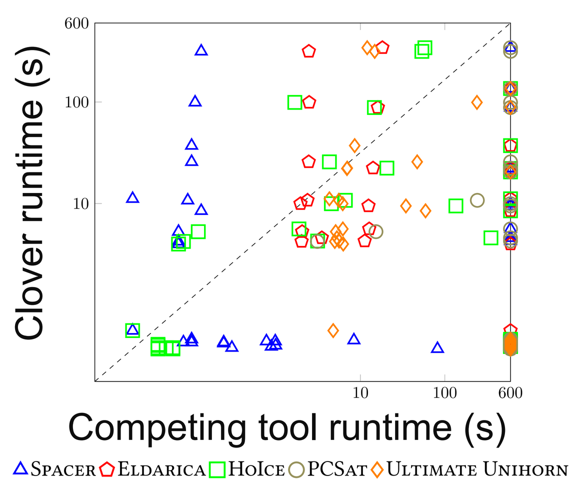
Over-approximate Environment
Over-Approximate Procedure

Under-approximate Environment
Over-Approximate Procedure

Experimental Results

	Clover	Spacer	Eldarica	HoIce	PCSat	Ultimate Unihorn
CHC-Comp	81	93	94	92	81	76
Mutual Recursion	24	12	0	0	0	0
Montgomery	16	6	14	14	3	13
s2n	6	5	0	2	N/A	6
Combination	2	0	0	0	0	0
Arrays	35	20	N/A	N/A	N/A	N/A

Number of examples in each set of benchmarks (shown on left) solved by each tool (shown on top)



Comparison of runtimes on the second set of benchmarks

- Clover: our prototype implementation
- Evaluated on three sets of benchmarks:
  1. 101 CHC-Comp 2019 benchmarks
  2. Benchmarks containing mutual recursion, programs based on Montgomery encoding and s2n, and combinations of these
  3. Benchmarks containing unbounded arrays
- Compared against other tools [1,2,3,4,5]
- Results demonstrate Clover is very effective for the latter two benchmark sets while remaining competitive with other tools on the first

References

[1] A. Champion, N. Kobayashi, and R. Sato, "Hoice: An ICE-based non-linear horn clause solver," in *APLAS*, ser. Lecture Notes in Computer Science, vol. 11275. Springer, 2018, pp. 146–156.

[2] D. Dietsch, M. Heizmann, J. Hoenicke, A. Nutz, and A. Podelski, "Ultimate TreeAutomizer," in *HCVS/PERR*, ser. EPTCS, vol. 296, 2019, pp. 42–47.

[3] H. Hojjat and P. Ru¨mmer, "The ELDARICA horn solver," in *FMCAD*. IEEE, 2018, pp. 1–7.

[4] A.Komuravelli, A.Gurfinkel, and S.Chaki,"SMT-based model checking for recursive programs," *Formal Methods in System Design*, vol. 48, no. 3, pp. 175–205, 2016.

[5] Y. Satake, T. Kashifuku, and H. Unno, "PCSat: Predicate constraint satisfaction," 2019, https://chc-comp.github.io/2019/chc-comp19.pdf.

This material is based upon work supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. #DGE-1656466 and NSF Grant No. 1525936. 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 National Science Foundation.