Loopfrog — loop summarization for static analysis

Daniel Kroening
Oxford University Comp. Lab., UK
, Natasha Sharygina
University of Lugano, Switzerland
,
Stefano Tonetta
FBK, Trento, Italy
, Aliaksei Tsitovich
University of Lugano, Switzerland
and
Christoph M. Wintersteiger
Computer Systems Institute, ETH Zurich, Switzerland

Loopfrog is a scalable static analyzer for ANSI-C programs, that combines the precision of model checking and the performance of abstract interpretation. In contrast to traditional static analyzers, it does not calculate the abstract fix-point of a program by iterative application of an abstract transformer. Instead, it calculates *symbolic* abstract transformers for program fragments (e.g., loops) using a *loop summarization* algorithm presented in [2]. Loopfrog computes abstract transformers starting from the inner-most loops, which results in linear (in the number of the looping constructs) run-time of the summarization procedure and which is often considerably smaller than the traditional saturation procedure of abstract interpetation. It also provides "leaping" counterexamples to aid in the diagnosis of errors.

An example for a very coarse over-approximation is the following: replace the loop by a piece of code that "havocs" the program state by setting all variables written by the loop to non-deterministic values. A way to obtain better summaries for loops is by strengthening them with loop invariants. Loopfrog does not aim at invariant discovery itself; we draw the loop invariants from a library of abstract domains. The concretization $\gamma(\hat{s})$ of any abstract state $\hat{s}$ corresponds to a predicate over concrete states, and is a candidate for some loop invariant. We heuristically traverse the lattice of abstract states in search of invariants that are preserved by the loop; the set of these abstract states then serves as the summary.

Candidate states $\hat{s}$ are checked as follows: as we start from an innermost loop, the body of the loop is itself loop-free. It is thus straight-forward to build the transition relation of the loop body by transforming the code fragment into a static single assignment (SSA) form[1]. Let ϕ_b denote the resulting expression, which is a precise predicate transformer of the loop body. We then form the conjunction of the concretization of $\hat{s}$ in the pre-state, the loop guard ϕ_g, the loop body ϕ_b, and the negation of the concretization in the post-state (denoted by the prime): $\gamma(\hat{s}) \wedge \phi_g \wedge \phi_b \wedge \neg\gamma(\hat{s}')$. If the loop body has any post-states that do not obey the constraints, the decision procedure will find this formula to be satisfiable. If the formula is unsatisfiable, the constraints are indeed an invariant. We consequently add $\gamma(\hat{s}) \rightarrow (\neg\phi'_g \wedge \gamma(\hat{s}'))$ to the loop summary.

The overall result of a loop summarization is a symbolic expression over pre- and post-states that encodes (in an over-approximating manner) those invariants preserved by the loop that can be expressed by the abstract domain. Our experimental results show that execution times of the decision procedure are usually very small, even if complex abstract domains are used, owing to the relative shortness of the program fragments[2].

[1] This step is performed by the symbolic execution engine of CBMC [1].

[2] The results of experiments and the Loopfrog binary are available at `www.verify.inf.usi.ch/loopfrog`

Once all loops have been summarized, the resulting program is a loop-free over-approximation of the input program. This program is again handed to a symbolic execution engine to check for violations of any assertions. The major advantage of using symbolic execution at this final stage is that LOOPFROG is able to obtain a counterexample trace in case an assertion is violated. Since the loops are over-approximated, the trace contains information only about the entry and exit state of each loop, and therefore is called a *leaping* counterexample.

[1] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS'04*, LNCS. Springer.

[2] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger. Loop summarization using abstract transformers. In *ATVA'08*, LNCS. Springer.