LPI: Software Verification with Local Policy Iteration
(Competition Contribution)

Egor George Karpenkov[1,2](✉)

[1] University Grenoble Alpes, VERIMAG, 38000 Grenoble, France
george@metaworld.me
[2] CNRS, VERIMAG, 38000 Grenoble, France

Abstract. LPI is a module for invariant generation embedded inside the CPAchecker framework. It uses a *local policy iteration* approach, which allows it to obtain precise numerical invariants. The approach performs computations in the *template constraints domain* using *maximization modulo* SMT, and terminates with a potentially over-approximating inductive invariant.

Local policy iteration is a sound, but incomplete technique which obtains numerical, conjunctive inductive invariants for the analyzed programs. It can prove programs to be safe by finding a *separating* inductive invariant, but can not find counterexamples to safety. We supply the generated inductive invariant to the k-induction procedure, which terminates with either a counterexample or a proof of safety.

1 Verification Approach

LPI is a module for obtaining numeric inductive invariants on programs, which is based on the *local policy iteration* [6] approach. Local policy iteration finds an inductive invariant in the *template constraints domain* for each of the abstraction points (loop-heads for reducible programs) of the analyzed program. In this abstract domain, a set of *templates* (linear expressions over program variables) is fixed in advance, and the inductive invariant is a vector of upper bounds on the chosen templates. For example, if the selected templates are x and $x + y$, a possible inductive invariant is $x \leq 5 \wedge x + y \leq 6$.

The tool includes a strategy for template synthesis. Templates are extracted from program expressions, and additionally from synthesizing simple linear expressions of a given size (e.g. $\pm x \pm y$ for all program variables x, y alive at the given location). Furthermore, the set of templates is *refined*: the analysis starts with a very coarse domain (upper and lower bound on each variable, emulating the *interval* domain), and if a *separating* inductive invariant is not found

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR".

(an invariant which separates the starting point from the error property), a domain is continuously refined to include more templates by increasing the size of synthesized linear expressions. However, the refinement is unguided and is not based on a target property.

Additionally, the analysis is augmented with a simple congruence module, which tracks parity (even or odd) of all variables and *simple* linear expressions (e.g. $x + y$).

The result of an LPI run is an inductive invariant, which might be an over-approximation of the reachable state space of the program. Thus pure LPI can only be used for verification, and not for finding counterexamples to the safety property. To address this, and to raise the number of programs which can be verified, an invariant produced by LPI is fed to the *k-induction* [1] procedure. For a given value of k, k-induction performs two checks: whether the error state is reachable from the initial one in k steps (forward reachability), and whether the negation of the error property is k-inductive, subject to the strengthening by the invariant produced by LPI. LPI invariant generation (including continuous refinement) runs asynchronously to the k-induction procedure, and they are both continuously refined (number of templates is increasing, and so is the value of k). Counterexamples produced by k-induction are cross-checked with CBMC [5], which either verifies a counterexample or refutes it.

We have used k-induction as it is a natural fit to our invariant generation procedure due to support for continuously refined invariants. LPI improves the precision of pure k-induction, as the inductiveness check may fail due to counterexamples-to-induction which are not reachable in the selected abstract domain.

2 Software Architecture

The verification module is embedded inside CPAchecker [3], an open-source framework for program analysis. CPAchecker implements the Configurable Programming Analysis [2] (CPA) concept: it runs a simple parametrized fixpoint iteration loop, and each analysis is a CPA which parametrizes this iteration. Consequently, LPI is implemented as a single CPA.

The CPA implementation of LPI relies on other CPAs to perform the splitting of the state space, namely *Location*, *Callstack* and *FunctionPointer*.

LPI analysis makes heavy use of optimization modulo SMT, which is done using *νZ* solver [4]. CPAchecker is written in Java and uses the Eclipse CDT parser for dealing with C code.

3 Strengths and Weaknesses

As LPI is expressed in the CPA framework, it benefits from it general strength: the ability to cooperate with other analyses.

The main strength of LPI is finding complex numerical invariants, which can not be found using standard abstract interpretation methods. In the current version of SV-COMP, we have found that many programs can be efficiently analyzed using explicit case enumeration, and complex numerical invariants are usually not required. Thus it limits the applicability of LPI to the SV-COMP dataset. However, on the categories we participate in we have found that LPI obtains reasonable results.

The additional limitation is that the inductive invariant produced by LPI is only sound with respect to mathematical integers and rationals. At the moment, LPI provides no bit-precise analysis, and unsound answers result mainly from integer overflow.

4 Tool Setup and Configuration

LPI code is available for download at http://lpi.metaworld.me/svcomp16.tar.bz2. The only external dependency of LPI is Java 7, all others are either shipped with or downloaded automatically by `ant`. The tool can be run from the main directory using the command `./scripts/cpa.sh -lpi-svcomp16 -disable-java-assertions -heap 10000m -spec property.prp target_program.i`. The parameter `-64` should be inserted before the last argument for 64-bit environment, and `-setprop cpa.predicate.handlePointerAliasing=false` is inserted in case of simple memory model. If a counterexample witness is found, it is written to the file `output/witness.graphml`. LPI can use the same wrapping script for `benchexec` as CPAchecker does. This tool participates in the "Integers and Control Flow" and "Software Systems" categories and opts out from all the others.

5 Software Project and Contributors

The LPI code was written by George Karpenkov. The k-induction module was developed by Matthias Dangl. CPAchecker [3] is mainly developed by the Software Systems Lab at the University of Passau. The code for both is distributed under the Apache 2.0 license.

References

1. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 622–640. Springer, Heidelberg (2015)
2. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
3. 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)

4. Bjørner, N., Phan, A.-D., Fleckenstein, L.: νZ - an optimizing SMT solver. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 194–199. Springer, Heidelberg (2015)
5. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
6. Karpenkov, E.G., Monniaux, D., Wendler, P.: Code analysis with local policy iteration. In: VMCAI (to appear, 2016)