ESBMC v6.0: Verifying C Programs using *k*-Induction and Invariant Inference (Competition Contribution)

Mikhail R. Gadelha[1]*, Felipe Monteiro[2], Lucas Cordeiro[3], and Denis Nicole[4]

[1]SIDIA Instituto de Ciência e Tecnologia, Brazil, m.gadelha@samsung.com
[2]Federal University of Amazonas, Brazil, felipemonteiro@ufam.edu.br
[3]University of Manchester, UK, lucas.cordeiro@manchester.ac.uk
[4]University of Southampton, UK, dan@ecs.soton.ac.uk

Abstract. ESBMC v6.0 employs a k-induction algorithm to both falsify and prove safety properties in C programs. We have developed a new interval-invariant generator that pre-processes the program, inferring invariants based on intervals and introducing them in the program as assumptions. Our experiments show that ESBMC v6.0 using k-induction can prove up to 7% more programs when the invariant generation is enabled.

1 Overview

The k-induction algorithm is an effective verification technique implemented in various software model checkers with the goal of proving partial correctness over a large number of different programs and properties [1,2,3]. Typical k-induction-based verifiers use iterative deepening and repeatedly unwind the program to produce the verification results; its incremental nature means that it always finds the smallest falsification [2]. In SV-COMP'19, we have implemented a new interval-invariant generator that runs as a pre-processing step in ESBMC [4]. In this implementation, invariants based on intervals are automatically introduced in the program as assumptions and, although the implementation has some limitations in keeping track of the relations between variables (i.e., our abstract domain is non-relational), it significantly strengthens the k-induction algorithm results; in particular, we have observed that the use of invariants increases the number of correct proofs by about 7% over the SV-COMP benchmarks.

2 Verification Approach

ESBMC uses a k-induction algorithm [2] to verify and falsify properties over C programs. Let a given C program P under verification be a finite transition system M, where we define:

- $I(s_n)$ and $T(s_n, s_{n+1})$ as the formulae over program's state variable set s_i constraining the initial states and transition relations of M;

* Jury member

– $\phi(s)$ as the formula encoding states satisfying a required safety property;
– $\psi(s)$ as the formula encoding states satisfying the completeness threshold, i.e. states corresponding to termination. $\psi(s)$ will contain unwindings no deeper than the maximum number of loop-iterations occurring in the program.

Note that, in our notation, termination and error are mutually exclusive: $\phi(s) \wedge \psi(s)$ is by construction unsatisfiable; s is a deadlock state if $T(s, s') \vee \phi(s)$ is unsatisfiable.

In each step k of the k-induction algorithm, three checks are performed: the base case $B(k)$, the forward condition $F(k)$ and the inductive step $S(k)$ [2]. $B(k)$ is the standard *bounded model checking* and it is satisfiable *iff* P has a counterexample of length k or less:

$$B(k) = I(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \neg\phi(s_i). \tag{1}$$

The forward condition checks for termination, i.e. whether the completeness threshold $\psi(s)$ must hold for the current k. If $F(k)$ is unsatisfiable, P has terminated:

$$F(k) = I(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \wedge \neg\psi(s_k). \tag{2}$$

No safety property $\phi(s)$ is checked in $F(k)$ as they were checked for the current k in the base case. Finally, the inductive condition $S(k)$ is unsatisfiable if, whenever $\phi(s)$ holds for k unwindings, it also holds after the next unwinding of P:

$$S(k) = \exists n \in \mathbb{N}^{+}. \bigwedge_{i=n}^{n+k-1} (\phi(s_i) \wedge T^{'}(s_i, s_{i+1})) \wedge \neg\phi(s_{n+k}). \tag{3}$$

Here $T^{'}(s_i, s_{i+1})$ is the transition relation after havocking the loop variables [2].

Through $B(k)$, $F(k)$, and $S(k)$, the k-induction algorithm at a given k is:

$$kind(P, k) = \begin{cases} P \text{ contains a bug,} & \text{if } B(k) \text{ is satisfiable,} \\ P \text{ is correct,} & \text{if } B(k) \vee [F(k) \wedge S(k)] \text{ is unsatisfiable,} \\ kind(P, k+1), & \text{otherwise.} \end{cases} \tag{4}$$

2.1 Invariant Inference based on Interval Analysis

Our major new feature is a new interval invariant generator for integer variables; it computes for every integer variable a lower and an upper bound of possible values. These intervals are injected into the program as assumptions (constraints) to address a limitation of the k-induction: when trying to check $S(k)$, the inductive step may find spurious counterexamples if the $T^{'}(s_i, s_{i+1})$ over-approximation is unconstrained. This is because we havoc the variables that are written in a loop, i.e. all loop variables are assigned non-deterministic values. The effect can be seen in Eq. (3): the inductive step checks if whenever ϕ holds for $k-1$ unwindings, it also holds in the current unwinding of the system. In Eq. (3), the state space is only constrained using the properties in the program; these are (usually) not strong enough to prove program correctness.

Several authors address this problem by generating program invariants to rule out unreachable regions of the state space, either as a pre-processing step where invariants are introduced in the program before verification [3], or during the verification itself [1,5]. Similarly to Rocha et al. [3], we perform a static analysis prior to loop unwinding and (over-)estimate the range that a variable can assume. In contrast to Rocha et al., we do not rely on external tools to infer polyhedral constraints (e.g., $ax + by \leq c$, where a, b, and c are constants and x and y are variables) over C programs. Instead, we implement a "rectangular" invariant generation based on interval analysis (e.g. $a \leq x \leq b$) as a pre-processing step of the verification, i.e., before the program is symbolically executed and the resulting formulae are checked by an SMT solver.

Here we use the abstract-interpretation component from CPROVER [6]. This implements an abstract domain based on expressions over intervals; these constraints associate each variable with an upper and lower bound. The algorithm starts by assuming an unbounded interval for each variable in the program and follows the reachable instructions from the `main` function while updates the intervals, merging them if necessary. When loops are found, an widening operation is applied, in order to accelerate the generation process [7].

Our tool generates new invariants $\varphi(s_n)$ and changes Eq. (3) to use them as assumptions during verification, such that the new inductive step is defined as:

$$S'(k) = \exists n \in \mathbb{N}^+ .\, \varphi(s_n) \wedge \bigwedge_{i=n}^{n+k-1} (\phi(s_i) \wedge T'(s_i, s_{i+1})) \wedge \neg\phi(s_{n+k}). \tag{5}$$

The *k*-induction algorithm of Eq. (4) now uses the inductive step from Eq. (5) to participate in all categories with C programs of SV-COMP'19.

3 Strengths and Weaknesses

We have observed that the use of invariants increases the number of correct proofs in ESBMC by about 7%. This, however, comes at a cost: due to bugs in the invariant generator, the number of incorrect proofs is trebled if these invariants are used. In particular, we do not track intervals of variables changed through pointers and nor if the intervals are defined in terms of other variables. For this we would need a relational analysis that can keep track of relations between variables. As a result, with the interval invariants enabled, ESBMC becomes a (better) *bug-finding* tool rather than one delivering proofs of guaranteed soundness.

In SV-COMP'19, ESBMC correctly claims 3556 benchmarks correct and finds existing errors in 1753. Sadly, it also finds unexpected errors for 14 benchmarks and fails to find the expected errors in another 41, which impacts its overall performance. The failures are mostly concentrated in the `MemSafety` and `ConcurrencySafety` categories and are mainly due to: (1) our non-relational abstract domain, (2) an internal bug in ESBMC (since corrected) which did not track variables going out of scope, and (3) an incomplete modelling of some *pthread* functions. ESBMC's performance has improved greatly since SV-COMP'18 (v4.60): the number of errors detected has increased by 36% and the number of correct-true results increased by 32%. The biggest improvements are reflected in the categories `ReachSafety` and `FalsificationOverall`.

4 Tool Setup and Configuration

In order to run our `esbmc-wrapper.py` script[1], one must set the architecture (*i.e.*, 32 or 64-bit), the competition strategy (*k*-induction, falsification or incremental BMC), the property file path, and the benchmark path, as:

```
esbmc-wrapper.py [-h] [-a {32,64}] [-p PROPERTY_FILE]
                 [-s {kinduction,falsi,incr}]
                 [benchmark]
```

where `-a` sets the architecture, `-p` sets the property file path, and `-s` sets the strategy, in this case, `kinduction` for *k*-induction.

Internally, by choosing the *k*-induction strategy, the following options are set for every property when executing ESBMC-kind: `--no-div-by-zero-check`, which disables the division by zero check (required by SV-COMP); `--k-induction`, which enables the *k*-induction; `--floatbv`, which enables floating-point SMT encoding; `--unlimited-k-steps`, which removes the upper limit of iteration steps in the *k*-induction algorithm; `--witness-output`, which sets the witness output path; `--force-malloc-success`, which sets that all dynamic allocations succeed (also an SV-COMP requirement); and `--interval-analysis`, which enables the invariant generation. In addition, ESBMC-kind sets further options depending on the property that needs to be checked: `--no-pointer-check` and `--no-bounds-check` for reachability verification; `--memory-leak-check` for memory verification; and `--overflow-check` for overflow verification. The Benchexec tool info module is named `esbmc.py` and the benchmark definition file is `esbmc-kind.xml`. For SV-COMP'19, ESBMC-kind uses Boolector v2.4.1 [8] and competes in all categories with C programs.

5 Software Project

The ESBMC source code is available for downloading at `https://github.com/esbmc/esbmc`, while self-contained binaries for ESBMC v6.0 64-bit can be downloaded from `https://github.com/esbmc/esbmc/releases`. ESBMC is publicly available under the terms of the Apache License 2.0. Instructions for building ESBMC from source are given in the file `BUILDING` (including the description of all dependencies). ESBMC is a joint project with the Federal University of Amazonas (Brazil), University of Southampton (UK), University of Manchester (UK), and University of Stellenbosch (South Africa).

[1] `https://gitlab.com/sosy-lab/sv-comp/archives-2019/blob/master/2019/esbmc-kind.zip`

References

1. Beyer, D., Dangl, M., Wendler, P.: Boosting *k*-Induction with Continuously-Refined Invariants. In: CAV, LNCS 9206, pp. 622–640, 2015.
2. Gadelha, M.Y.R., Ismail, H.I., Cordeiro, L.C.: Handling Loops in Bounded Model Checking of C Programs via *k*-Induction. In: STTT, **19**(1), pp. 97–114, 2017.
3. Rocha, W., Rocha, H., Ismail, H., Cordeiro, L.C., Fischer, B.: DepthK: A *k*-Induction Verifier Based On Invariant Inference for C Programs - (Competition Contribution). In: TACAS, LNCS 10206, pp. 360–364, 2017.
4. Gadelha, M. R., Monteiro, F. R., Morse, J., Cordeiro, L. C., Fischer, B. and Nicole, D. A.: ESBMC 5.0: An Industrial-Strength C Model Checker. In: ASE. IEEE/ACM, pp. 888-891, 2018.
5. Malík, V., Martiček, Š., Schrammel, P., Srivas, M., Vojnar, T., Wahlang, J.: 2LS: Memory Safety and Non-Termination. In: TACAS, LNCS 10806, pp. 417–421, 2018.
6. Kroening, D.: CProver Manual. `http://www.cprover.org/cprover-manual/` (2018) [Online; accessed February-2019].
7. Yamaguchi, T., Brain, M., Ryder, C., Imai, Y., Kawamura, Y.: Application of Abstract Interpretation to the Automotive Electronic Control System. In: VMCAI, LNCS 11388, pp. 425–445, 2019.
8. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 System Description. Journal on Satisfiability, Boolean Modeling and Computation. **9**, pp. 53–58, 2015.