Backward Bounded Model Checking in CPAchecker

Bas Laarakker

Google Summer of Code '23

2023-09-11 @ CPAchecker Workshop

About Me

- Currently MSc Logic @ ILLC, University of Amsterdam
  - Interested in mathematical logic and theoretical CS
- Participant in GSoC '23 @ SoSy Lab, LMU Munich
  - Summer program for contributing to open-source software
  - Supervised by Nian-Ze Lee

Key Points

- Improved support for any backward analysis in CPAchecker

Key Points

- Improved support for any backward analysis in CPAchecker
- Implemented backward BMC algorithm in CPAchecker
  - Handling pointer aliasing improves performance
  - Backward BMC can complement regular BMC

Key Points

- Improved support for any backward analysis in CPAchecker
- Implemented backward BMC algorithm in CPAchecker
  - Handling pointer aliasing improves performance
  - Backward BMC can complement regular BMC
- Empirical evidence for proof that backward BMC and (plain) k-induction are equivalent [2]

Key Points

- Improved support for any backward analysis in CPAchecker
- Implemented backward BMC algorithm in CPAchecker
  - Handling pointer aliasing improves performance
  - Backward BMC can complement regular BMC
- Empirical evidence for proof that backward BMC and (plain) k-induction are equivalent [2]
- Want to identify limiting factors of backward BMC performance
  - Backward formula construction and SMT solving

Backward Bounded Model Checking

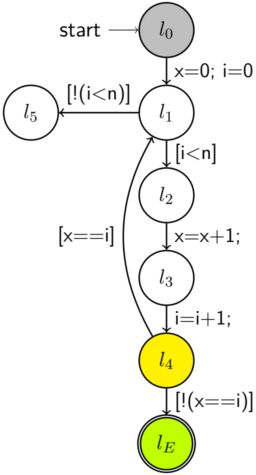
Backward Analysis

- We study the *error location reachability problem*
  - Forward analysis unrolls a program starting from the *main entry* to the *error locations*
  - Backward analysis unrolls a program starting from the *error locations* to the *main entry*

Backward Bounded Model Checking

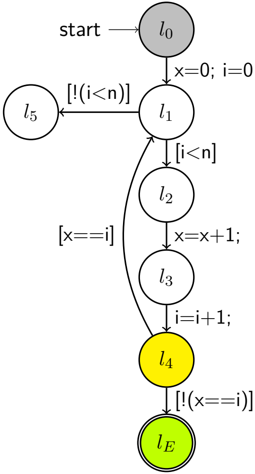
- Based on paper *Backward Symbolic Execution with Loop Folding*, M. Chalupa and J. Strejček [2]
- They claim that BBMC is equivalent to *k*-induction by showing:
  - For a CFA A, let P be the set of all satisfiable paths from the error location. Then both algorithms, when executed on CFA A,
    - return FALSE if P contains a path to the entry location;
    - return TRUE if P is finite and contains no path to the entry location;
    - do not terminate if P is infinite and contains no path to the entry location.

Example: *k*-Induction



Base Case	Step Case
$(l_0, -1)$	$(l_4, 0)$
$\downarrow$	$\downarrow$
$(l_1, -1)$	$(l_1, 0)$
$\downarrow$	$\downarrow$
$(l_2, -1)$	$(l_2, 0)$
$\downarrow$	$\downarrow$
$(l_3, -1)$	$(l_3, 0)$
$\downarrow$	$\downarrow$
$(l_4, 0)$	$(l_4, 1)$
$\downarrow$	$\downarrow$
$(l_E, 0)$	$(l_E, 1)$

Example: BBMC



$$(l_E, \top)$$
$$\uparrow$$
$$(l_4, x \neq i)$$
$$\uparrow$$
$$(l_3, x \neq i + 1)$$
$$\uparrow$$
$$(l_2, x + 1 \neq i + 1)$$
$$\uparrow$$
$$(l_1, i < n \wedge x + 1 \neq i + 1) \longleftarrow (l_0, \bot)$$
$$\uparrow$$
$$(l_4, \bot)$$

Backward Analysis in CPAchecker

Backward Analysis in CPAchecker

- Some support for backward analysis exists:
  - Backward PredicateCPA (partial support)
    - Backward formula construction
  - Backward CallstackCPA
  - Backward LocationCPA

Backward Analysis in CPAchecker

- Some support for backward analysis exists:
  - Backward PredicateCPA (partial support)
    - Backward formula construction
  - Backward CallstackCPA
  - Backward LocationCPA
- Missing support for backward analysis:
  - Exporting witness from backward analysis
  - Backward LoopBoundCPA
  - Handling of pointer aliasing by PredicateCPA

Contributions to CPAchecker

- Implemented support for backward analysis for WitnessExporter
  - MR1

Contributions to CPAchecker

- Implemented support for backward analysis for WitnessExporter
  - MR1
- Implemented support for LoopBoundCPA and the BBMC algorithm in the `BackwardBMCAlgorithm` class
  - MR2

Contributions to CPAchecker

- Implemented support for backward analysis for WitnessExporter
  - MR1
- Implemented support for LoopBoundCPA and the BBMC algorithm in the `BackwardBMCAlgorithm` class
  - MR2
- Implemented support for handling pointer aliasing for backward analysis (WIP)
  - MR3
  - Some language features are not yet supported:
    - `union`
    - Pointers to `struct`

Experimental Setup

- On revision 44369 of CPAchecker
  - Branch backward-bmc-algorithm
- Algorithms:
  - BBMC (pointer aliasing disabled)
  - BBMC+PA (pointer aliasing enabled)
  - BMC
  - Plain *k*-induction
- 5652 *ReachSafety* programs of SV-COMP '23 [1]
  - Removed programs with unsupported features for backward pointer aliasing
    - *ProductLines*
    - *LinuxDrivers*

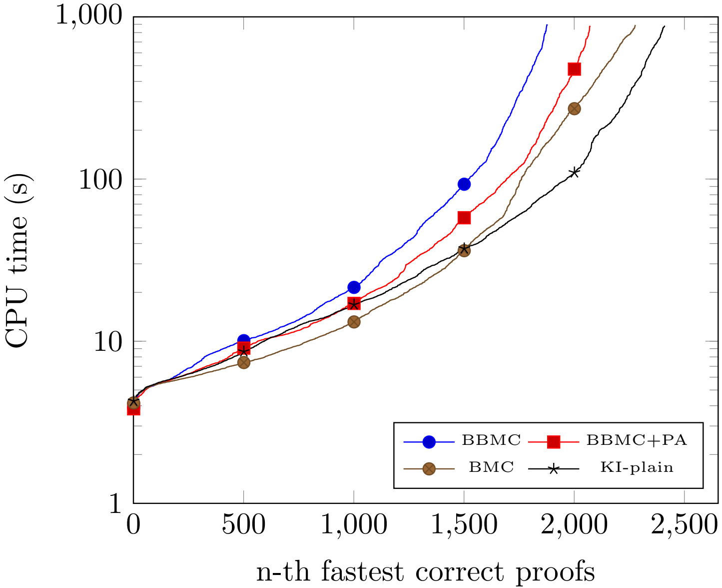
Results

Algorithm	**BBMC**	**BBMC+PA**
Correct True	727	959
Correct False	1152	1113
Incorrect True	10	47
Incorrect False	383	7

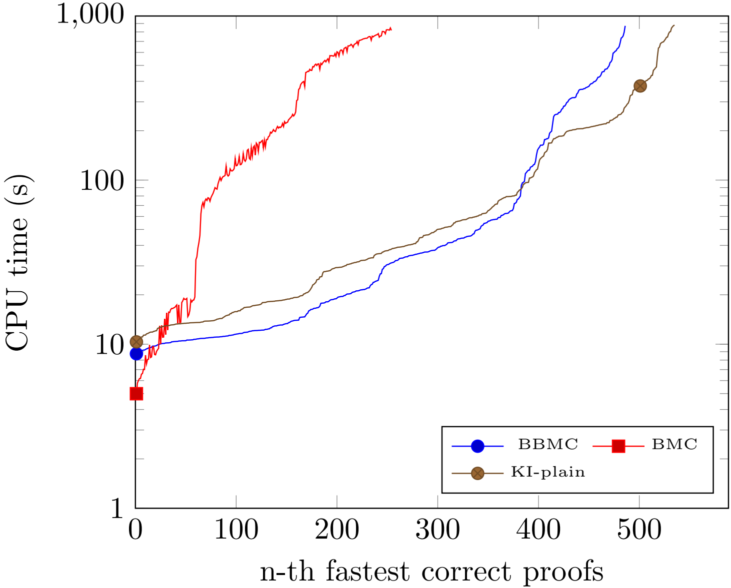
Table: Results of BBMC on 5652 *ReachSafety* programs

- Adding support for backward pointer aliasing solves most false alarms and improves performance by 11%
- However, we get more false proofs
  - These programs have unsupported language features

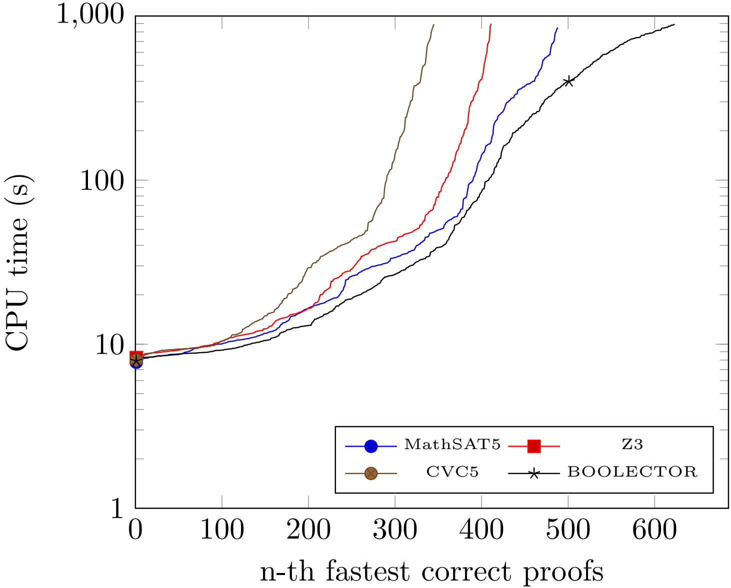
Quantile Plot: All Tasks



Quantile Plot: ECA Tasks



BBMC on ECA Solver Comparison



Observations

- BBMC *complements* BMC on certain program sets
- BBMC and *k*-induction achieve their results at the same loop unrolling bound
- Their results differ only when one algorithm times out before reaching the “required” bound to solve the program
  - Suggests that BBMC and *k*-induction are equivalent on *ReachSafety-ECA*
- On certain programs, BBMC takes a lot of time compared to *k*-induction
  - Likely due to SMT solving phase
  - Solvers make a difference

Conclusion

- Implemented BBMC and further support for backward analysis
- BBMC performs quite well, especially on certain subsets
  - BBMC and k-induction seem to be equivalent, at least on *ReachSafety-ECA*
  - BBMC can benefit from faster performance
- Future work:
  - Identify bottlenecks of BBMC performance (backward formula construction?)
  - Evaluate BBMC vs k-induction further

Technical Challenges

- JavaScript error when generating report
  - `TypeError: simplifiedGraphMap is undefined`
- Improve support for backward pointer aliasing
  - `union`
  - Pointers to `struct`

```c
struct Point { int x; };
// Expected result: FALSE
int test1() {
  struct Point p;
  struct Point *pt = &p;
  pt->x = 2;
  p.x = pt->x - 1;
  if (p.x == 1) {
    reach_error();
    return -1;
  }
  return 0;
}
```

```c
union Data { int a; int b; };
// Expected result: FALSE
int test2() {
  union Data d;
  d.a = 6;
  d.b = d.a + 1;
  if (d.b == 7) {
    reach_error();
    return -1;
  }
  return 0;
}
```

References I

Dirk Beyer.
Competition on software verification and witness validation: Sv-comp 2023.
In Sriram Sankaranarayanan and Natasha Sharygina, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 495–522, Cham, 2023. Springer Nature Switzerland.

Marek Chalupa and Jan Strejček.
Backward symbolic execution with loop folding.
In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, Static Analysis, pages 49–76, Cham, 2021. Springer International Publishing.