Configurable Software Model Checking CPAchecker

Dirk Beyer

LMU Ludwig-Maximilians-Universität München
SoSy-Lab Software Systems
Software Verification

C Program

```c
int main() {
    int a = foo();
    int b = bar(a);
    assert(a == b);
}
```

Verification Tool

TRUE
i.e., specification is satisfied

FALSE
i.e., bug found

General method:
Create an overapproximation of the program states /
compute program invariants
CPAchecker History

- 2002: BLAST with lazy abstraction refinement
- 2003: Multi-threading support
- 2005: Memory safety, predicated lattices
- 2006: Lazy shape analysis
- Maintenance and extensions became extremely difficult because of design choices that were not easy to revert
- 2007: Configurable program analysis, CPAchecker was started as complete reimplementation from scratch
CPAchecker History (2)

- 2009: Large-block encoding
- 2010: Adjustable-block encoding
- 2012: Conditional model checking, PredAbs vs. Impact
- 2013: Explicit-state MC, BDDs, precision reuse
- ...
Iterative fixpoint (forward) post computation
Iterative fixpoint (forward) post computation
Iterative fixpoint (forward) post computation
Iterative fixpoint (forward) post computation
Software Model Checking

\[ \text{Reached}, \ \text{Frontier} := \{ e_0 \} \]

\textbf{while} \ Frontier \neq \emptyset \ \textbf{do}

\hspace{1em} \text{remove} \ e \ \text{from} \ \text{Frontier} \\
\hspace{1em} \textbf{for all} \ e' \in \text{post}(e) \ \textbf{do}

\hspace{2em} \textbf{if} \ \neg \text{stop}(e', \ \text{Reached}) \ \textbf{then}

\hspace{3em} \text{add} \ e' \ \text{to} \ \text{Reached}, \ \text{Frontier} \\

\textbf{return} \ \text{Reached}
Software Verification by Data-Flow Analysis

Fixpoint computation on the CFG
Software Verification by Data-Flow Analysis

Fixpoint computation on the CFG
Fixpoint computation on the CFG
Software Verification by Data-Flow Analysis

Fixpoint computation on the CFG
Fixpoint computation on the CFG
Software Verification by Data-Flow Analysis

Fixpoint computation on the CFG
Software Verification by Data-Flow Analysis

Fixpoint computation on the CFG
Software Verification by Data-Flow Analysis

Fixpoint computation on the CFG
Fixpoint computation on the CFG
\[ \text{Reached, Frontier} := \{ e_0 \} \]

\textbf{while} Frontier \neq \emptyset \textbf{ do}

\hspace{1em} \text{remove } e \text{ from Frontier}

\hspace{1em} \textbf{for all } e' \in \text{post}(e) \textbf{ do}

\hspace{2em} \text{if } e'' \neq e'' \text{ then}

\hspace{3em} \text{replace } e''\text{ in Reached, Frontier by } e''\text{ new}

\hspace{1em} \text{if } \neg \text{stop}(e', \text{Reached}) \text{ then}

\hspace{2em} \text{add } e' \text{ to Reached, Frontier}

\textbf{return} \quad \text{Reached}
Reached, Frontier := \{e_0\}

while Frontier \neq \emptyset do
    remove e from Frontier
    for all e' \in \text{post}(e) do
        for all e'' \in Reached do
            e''_{new} := \text{merge}(e', e'')
            if e''_{new} \neq e'' then
                replace e'' in Reached, Frontier by e''_{new}
            if \neg \text{stop}(e', Reached) then
                add e' to Reached, Frontier
        return Reached
Configurable Program Analysis

▶ Better combination of abstractions
→ Configurable Program Analysis  [Beyer/Henzinger/Theoduloz CAV’07]

Unified framework that enables intermediate algorithms
Dynamic Precision Adjustment

Lazy abstraction refinement: [Henzinger/Jhala/Majumdar/Sutre POPL’02]

- Different predicates per location and per path
- Incremental analysis instead of restart from scratch after refinement
Dynamic Precision Adjustment

Better fine tuning of the precision of abstractions
→ Adjustable Precision
[Beyer/Henzinger/Theoduloz ASE’08]

Unified framework enables:

▶ switch on and off different analysis, and can
▶ adjust each analysis separately

● Not only refine, also abstract!
Adjustable Block-Encoding

- Handle loop-free blocks of statements at once
- Abstract only between blocks (less abstractions, less refinements)

[Beyer/Cimatti/Griggio/Keremoglu/Sebastiani FMCAD’09]
[Beyer/Keremoglu/Wendler FMCAD’10]
CPA – Summary

- Unification of several approaches → reduced to their essential properties
- Allow experimentation with new configurations that we could never think of
- Flexible implementation CPAchecker
Framework for Software Verification — current status

- Written in Java
- Open Source: Apache 2.0 License
- ~80 contributors so far from 15 universities/institutions
- 430,000 lines of code
  (275,000 without blank lines and comments)
- Started 2007

https://cpachecker.sosy-lab.org
CPAchecker: Features

- Input language C (experimental: Java)
- Web frontend available: https://cpachecker.appspot.com
- Counterexample output with graphs
- Benchmarking infrastructure available (with large cluster of machines)
- Cross-platform: Linux, Mac, Windows
Among world’s best software verifiers: https://sv-comp.sosy-lab.org/2018/results/
Continuous success in competition since 2012 (52 medals: 16x gold, 18x silver, 18x bronze)
Awarded Gödel medal by Kurt Gödel Society
Used for Linux driver verification with dozens of real bugs found and fixed in Linux
Included Concepts:
- CEGAR
- Interpolation
- Adjustable-block encoding
- Conditional model checking
- Verification witnesses

Further available analyses:
- \textsc{impact} algorithm
- Bounded model checking
- k-Induction
- Property-directed reachability
Completely modular, and thus flexible and easily extensible
Every abstract domain is implemented as a "Configurable Program Analysis" (CPA)
E.g., predicate abstraction, explicit-value analysis, intervals, octagon, BDDs, memory graphs, and more
Algorithms are central and implemented only once
Separation of concerns
Combined with Composite pattern
CPAAlgorithm is the core algorithm for reachability analysis / fixpoint iteration.

Other algorithms can be added if desired, e.g.,
  - CEGAR
  - Double-checking counterexamples
  - Sequential combination of analyses
CPAchecker: Architecture

- Source Code
- Spec
- Parser & CFA Builder
- CPA Algorithm
  - Spec CPA
  - Location CPA
  - Callstack CPA
  - Predicate CPA
- CEGAR Algorithm
- k-induction Algorithm
- Results
Try CPAchecker

▶ Online at Google AppEngine:
   https://cpachecker.appspot.com/

▶ Download for Linux/Windows:
   https://cpachecker.sosy-lab.org
   
   ▶ Run scripts/cpa.sh | scripts\cpa.bat
   ▶ -predicateAnalysis <FILE>
   ▶ Windows/Mac need to disable bitprecise analysis:
     -predicateAnalysis-linear
     -setprop solver.solver=smtinterpol
     -setprop analysis.checkCounterexamples=false

▶ Look at output/CPALog.txt for problems

▶ Open .dot files with dotty / xdot (www.graphviz.org/)

▶ Open graphical report in browser: output/*.html
Model Checkers check only what you specified

**CPAchecker’s default:**
- Label ERROR
- Calling function `_assert_fail()`
- `assert(pred)` needs to be pre-processed

**SV-COMP:**
- Calling function `_VERIFIER_error()`
- `-spec sv-comp-reachability`
Want to implement your own analysis?

- Easy, just write a CPA in Java
- Implementations for 10 interfaces needed
- But for 8, we have default implementations
  - Minimal configuration:
    - abstract state and
    - abstract post operator
The CPA framework is flexible:

- Many components are provided as CPAs:
  - Location / program counter tracking
  - Callstack tracking
  - Specification input (as automata)
  - Pointer-aliasing information

- CPAs can be combined,
  so your analysis doesn’t need to care about these things