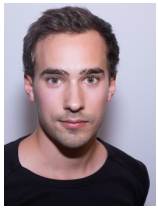
Real-World Software Verification with CPAchecker

Thomas Lemberger

LMU Munich, Germany

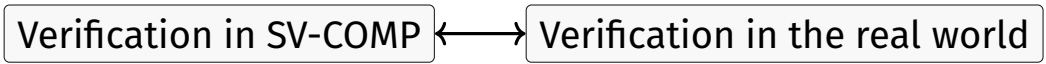


Real-World Software Verification with CPAchecker

Thomas Lemberger

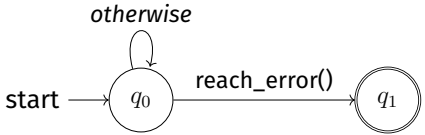
LMU Munich, Germany

Verification in SV-COMP

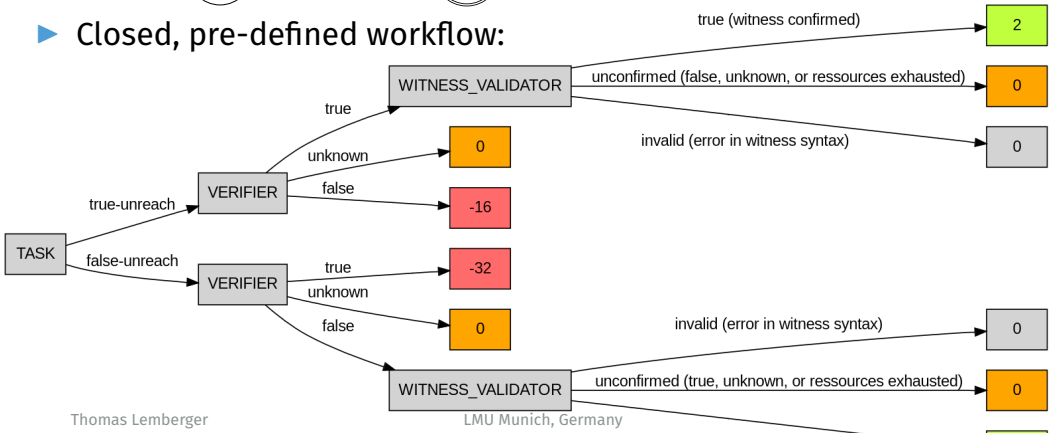

Verification in SV-COMP
Verification in the real world

Verification in SV-COMP

- Fixed set of simple properties



- Closed, pre-defined workflow:



Verification in the Real World

- We have custom properties.
- We want to continue analysis beyond the first result (despite false positives).
- We have to understand the verification result.
- (Which configuration/abstract domain to choose?)

- We use CPAchecker.[1]
  `https://gitlab.com/sosy-lab/software/cpachecker/`
- We focus on C programs.
- Disclaimer: Not *everything* works well.
  - Basic multi-file verification
  - Not full C standard supported (missing, e.g.: `longjmp`, `atexit`, variadic method arguments).

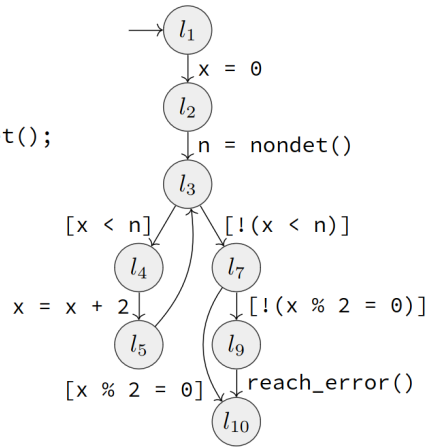
[1] D. Beyer and M. E. Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Proc. CAV*. LNCS 6806. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16

CPAchecker

- CPAchecker uses control-flow automaton (CFA) as intermediate represenation of programs
- CPAchecker uses configurable program analyses (CPAs) to define abstraction of program state space

Control-flow Automaton

```
1  int main(void) {
2    unsigned int x = 0;
3    unsigned short n = nondet();
4    while (x < n) {
5      x += 2;
6    }
7    if (x % 2 == 0) {}
8    else
9      reach_error();
10 }
```



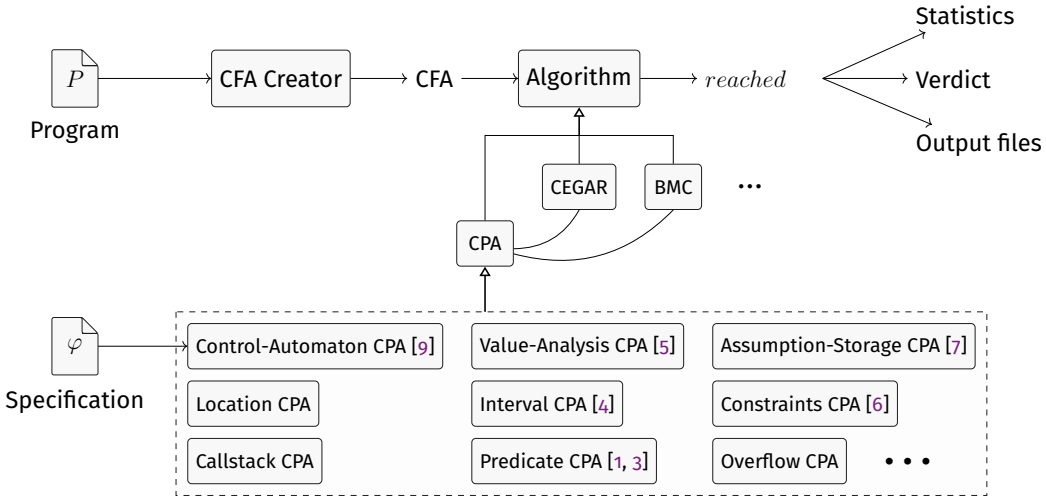
Configurable Program Analysis

- CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \text{merge}, \text{stop}, \text{prec})$
- D: Abstract domain
- $\pi \in \Pi$: Precision
- $\rightsquigarrow$: Transfer relation
- $\text{merge}(e, e')$: Merge operator
- $\text{stop}(e, R)$: Stop operator
- $\text{prec}(e, \pi)$: Precision-adjustment operator

Composition of CPAs

- Combined tracking of various program information through Composite CPA.
- Composite CPA combines CPA with abstract elements E_1 and CPA with abstract elements E_2.
- Special operator: $\downarrow : E_1 \times E_2 \to E_1$
  $\downarrow(\{x = 0\}, \{\text{assume } x == 2\}) = \{x = 2\}$

CPAchecker


Statistics
P
CFA Creator
CFA
Algorithm
$reached$
Verdict
Program
Output files
CEGAR
BMC
...
CPA
φ
Control-Automaton CPA [9]
Value-Analysis CPA [5]
Assumption-Storage CPA [7]
Specification
Location CPA
Interval CPA [4]
Constraints CPA [6]
Callstack CPA
Predicate CPA [1, 3]
Overflow CPA
• • •

Custom program properties

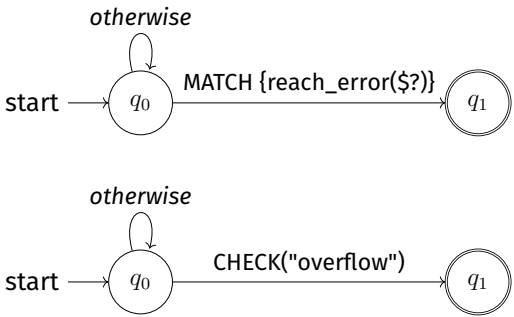
- SV-COMP: `reach_error();`
- Other default: `assert.h` assertions

  `assert(x > 0);`

  → `if (x > 0) ; else __assert_fail("x > 0", ...);`
- Projects do not always use `assert.h`.

```
// From coreutils cksum.c :
unsigned int length;
// ...
if (length + bytes_read < length) {
  errno = EOVERFLOW;
  return false;
}
```

Specs in CPAchecker: Control-Automaton CPA



- Multiple automata can be composed

Defining custom program properties

- Based on the BLAST query language[2].

```
CONTROL AUTOMATON OverflowOrAssert

INITIAL STATE Init;
STATE USEFIRST Init:
 CHECK("overflow")
  -> ERROR("no-overflow: integer overflow in $location");
 MATCH {__assert_fail($1, $2, $3, $4)}
  -> ERROR("assertion in $location: Condition $1 failed in $2, line $3");

END AUTOMATON
```

[2] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. "The BLAST Query Language for Software Verification". In: *Proc. SAS*. LNCS 3148. Springer, 2004, pp. 2–18. DOI: 10.1007/978-3-540-27864-1_2

Defining custom program properties

CPAchecker query language:
https://gitlab.com/sosy-lab/software/cpachecker/-/blob/trunk/doc/SpecificationAutomata.md

- Special states: ERROR, STOP, BREAK
- Transition triggers:

```
MATCH { x = $1 }
MATCH { f($1, $2) }
CHECK(OverflowCPA, "overflow")
CHECK("overflow")
CHECK(SMGCPA, "has-invalid-frees")
CHECK(IS_TARGET_STATE)
```

Defining custom program properties

CPAchecker query language:
https://gitlab.com/sosy-lab/software/cpachecker/-/blob/trunk/doc/SpecificationAutomata.md

- Special states: ERROR, STOP, BREAK
- Transition triggers (`MATCH`, `CHECK`)
- Transition actions:

```
DO automatonVar = $1
DO counter = counter + 1
DO vars[$1] = true
MODIFY(ValueAnalysis, "setvalue(x==0)")
```

Defining custom program properties

CPAchecker query language:
https://gitlab.com/sosy-lab/software/cpachecker/-/blob/trunk/doc/SpecificationAutomata.md

- Special states: ERROR, STOP, BREAK
- Transition triggers (`MATCH`, `CHECK`)
- Transition actions (`DO`, `MODIFY`)
- Transition assumptions:
  `ASSUME {x = 0; n = 1}`

Ignore false positives

```
EVAL(location, "lineno") != 10044 && CHECK("overflow")
  -> ERROR("no-overflow: integer overflow in $location");
```

- Issue: Source-code changes
- TODO: Improve this.

Model external functions

```
extern int sum(int a, int b);
// …
int x = sum(0, 100);
```

```
MATCH { $1 = sum($?) } -> ASSUME { $1 == 99999; } GOTO Init;

MATCH { $1 = sum($2, $3) } -> ASSUME { $1 == $2 + $3; } GOTO Init;
```

Define custom program properties

Only fail if an overflow occurs in an expression that is stored in a variable that is used afterwards.

```
unsigned int bytes_read;
// ...
bytes_read = bytes_read - 1; // may underflow
```

```
STATE USEFIRST Init:
    MATCH { $1 = $? } && CHECK("overflow")
        -> overflown[$1] = true GOTO Init;
    CHECK("use", $1) && overflown[$1] == true
        -> ERROR("Variable $1 contains overflown value and is used");
```

Define custom program properties

Only fail if an overflow occurs in an expression that is stored in a variable that is used afterwards.

```
unsigned int bytes_read;
// ...
bytes_read = bytes_read - 1; // may underflow
```

```
STATE USEFIRST Init:
    MATCH { $1 = $? } && CHECK("overflow")
        -> overflown[$1] = true GOTO Init;
    CHECK("use", $1) && overflown[$1] == true
        -> ERROR("Variable $1 contains overflown value and is used");
```

Define custom program properties

- We can not use pattern matching in CHECK.
- But we can implement many things through communication with CPAs:

```
STATE USEFIRST Init:
    CHECK(IS_TARGET_STATE) -> ERROR;
    MATCH { $1 = $? } && CHECK("overflow")
        -> MODIFY(TaintAnalysis, "taint($1)") GOTO Init;
```

- TODO: Taint analysis

What did the verifier do?

- The promised land: verification-result witnesses [9]
- Violation: abstract error path
- Proof: invariants
- Issue: Often stripped of information that verifier computed

```
main N12:
(assert (let ((.def_87 (= |main::x| (_ bv0 32)))).def_87))
```

But correctness witness without any invariant.

⇒ We turn to verifier-specific files

CPAchecker-specific files

Statistics.txt, CFA.dot, ARG.svg, ARG.dot, predmap.txt, abstractions.txt, invariants.txt, Counterexample.*, ...

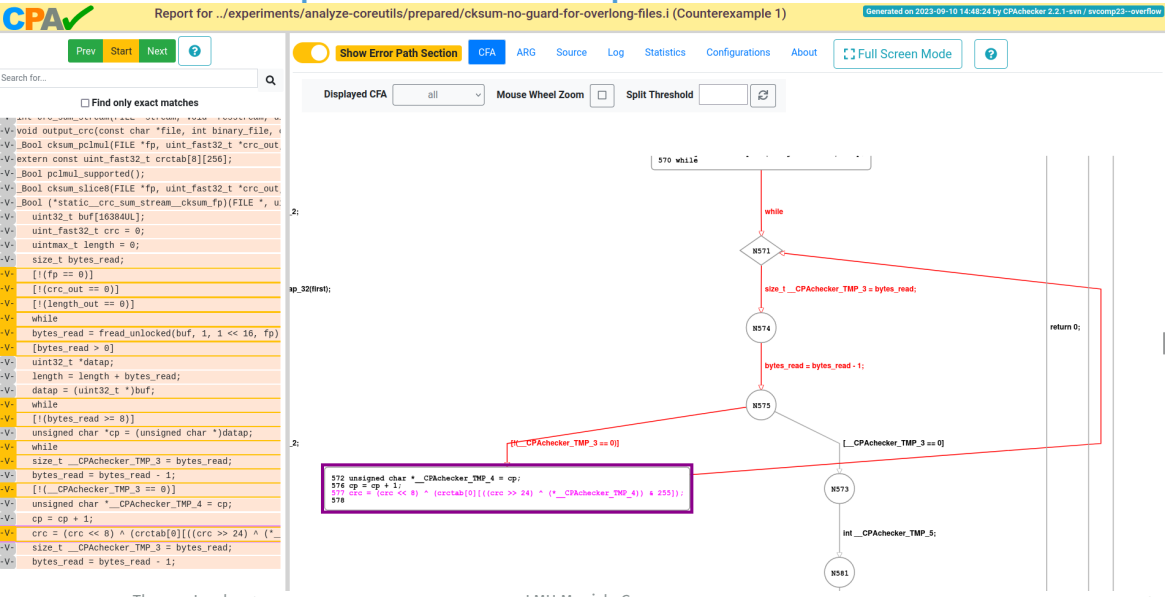
Conclusion

- CPAchecker provides a nice framework (specification + analyses) to implement real-world analyses
- Powerful specification language
- Existing practical issues: multi-file analysis, non-sequential control-flow, understanding verification results

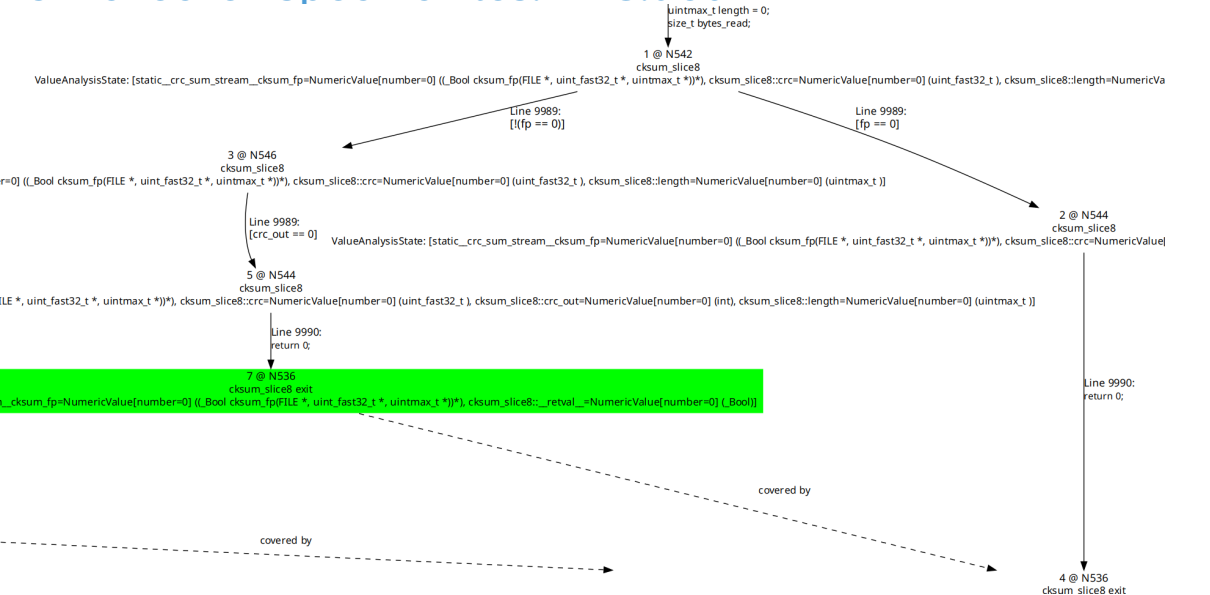
- Lots of SV-COMP medals:
  `https://cpachecker.sosy-lab.org/achieve.php`
- More than 20 active contributors.
- We are always happy to help!

Appendix

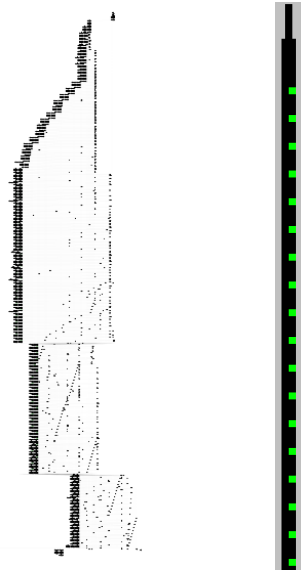
CPAchecker-specific files: Report.html



CPAchecker-specific files: ARG.dot

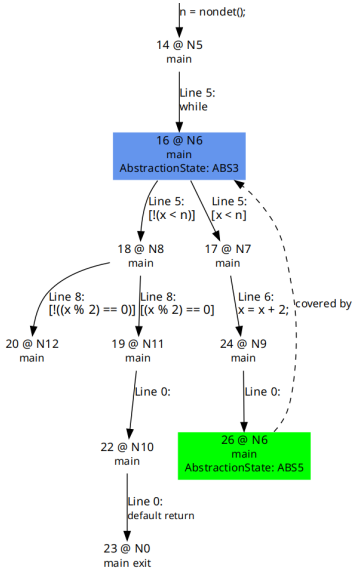


CPAchecker-specific files: ARG.svg



- Abstract view of ARG
- Width in line represents number of ARG nodes with that depth
- Green node: stopped
- Red node: found target
- Yellow node: in waitlist

abstractions.txt




```
(set-info :source |printed by MathSAT|)
(declare-fun |main::x| () (_ BitVec 32))

0 (3) @1:
(assert true)

3 (5) @6:
(assert (let ((.def_83 (bvurem |main::x| (_ bv2
    32)))(let ((.def_86 (bvult |main::x| .def
    _83)))(let ((.def_105 (not .def_86)))(let ((.
    def_91 (= .def_83 (_ bv0 32)))(let ((.def_88
     (bvurem .def_83 (_ bv2 32)))(let ((.def_89
    (= .def_83 .def_88)))(let ((.def_104 (and .
    def_89 .def_91)))(let ((.def_106 (and .def
    _104 .def_105)))(let ((.def_84 (bvult .def_83
     (_ bv2 32)))(let ((.def_107 (and .def_84 .
    def_106))).def_107))))))))))
```

Counterexample.1.assignment.txt

- Model of the counterexample

```
int main(void) {
    unsigned int x = 0;                 main::n@3: 6
    unsigned short n = nondet();        main::x@2: 0
    while (x < n) {                     main::x@3: 2
        x += 2;                         main::x@4: 4
    }                                   main::x@5: 6
    if (x > 5) reach_error();           nondet!2@: 6
}
```

Counterexample.1.core.txt

- Reduced counterexample path with assignments

```
line 4: N2 -{unsigned int x = 0;}-> N3
       x == 0U;
line 5: N4 -{n = nondet();}-> N5
       x == 0U;
line 6: N5 -{while}-> N6
line 6: N6 -{[x < n]}-> N7
       x == 0U;
line 7: N7 -{x = x + 2;}-> N9
       x == 2U;
none:  N9 -{}-> N6
line 6: N6 -{[x < n]}-> N7
       x == 2U;
```

```
line 7: N7 -{x = x + 2;}-> N9
       x == 4U;
none:  N9 -{}-> N6
line 6: N6 -{[x < n]}-> N7
       x == 4U;
line 7: N7 -{x = x + 2;}-> N9
       x == 6U;
none:  N9 -{}-> N6
line 6: N6 -{[!(x < n)]}-> N8
       x == 6U;
line 9: N8 -{[x > 5]}-> N11
       x == 6U;
line 9: N11 -{reach_error();}-> N12
```

References I

[1] D. Beyer, M. Dangl, and P. Wendler. “A Unifying View on SMT-Based Software Verification”. In: *J. Autom. Reasoning* 60.3 (2018), pp. 299–335. ISSN: 1573-0670. DOI: 10.1007/s10817-017-9432-6.

[2] D. Beyer and M. E. Keremoglu. “CPAchecker: A Tool for Configurable Software Verification”. In: *Proc. CAV*. LNCS 6806. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16.

[3] D. Beyer, M. E. Keremoglu, and P. Wendler. “Predicate Abstraction with Adjustable-Block Encoding”. In: *Proc. FMCAD*. https://www.sosy-lab.org/research/pub/2010-FMCAD.Predicate_Abstraction_with_Adjustable-Block_Encoding.pdf. FMCAD, 2010, pp. 189–197.

[4] D. Beyer, P.-C. Chien, and N.-Z. Lee. “CPA-DF: A Tool for Configurable Interval Analysis to Boost Program Verification”. In: *Proc. ASE*. 2023.

References II

[5] D. Beyer and S. Löwe. “Explicit-State Software Model Checking Based on CEGAR and Interpolation”. In: *Proc. FASE*. LNCS 7793. Springer, 2013, pp. 146–162. DOI: 10.1007/978-3-642-37057-1_11.

[6] D. Beyer and T. Lemberger. “Symbolic Execution with CEGAR”. In: *Proc. ISoLA*. LNCS 9952. Springer, 2016, pp. 195–211. DOI: 10.1007/978-3-319-47166-2_14.

[7] D. Beyer et al. “Conditional Model Checking: A Technique to Pass Information between Verifiers”. In: *Proc. FSE*. ACM, 2012. DOI: 10.1145/2393596.2393664.

[8] D. Beyer et al. “The BLAST Query Language for Software Verification”. In: *Proc. SAS*. LNCS 3148. Springer, 2004, pp. 2–18. DOI: 10.1007/978-3-540-27864-1_2.

References III

[9] D. Beyer et al. “Verification Witnesses”. In: *ACM Trans. Softw. Eng. Methodol.* 31.4 (2022), 57:1–57:69. DOI: 10.1145/3477579.