Software Verification with CPAchecker 3.0: Tutorial and User Guide

The CPAchecker Team

Speakers: Dirk Beyer, Thomas Lemberger, and Philipp Wendler

LMU Munich, Germany

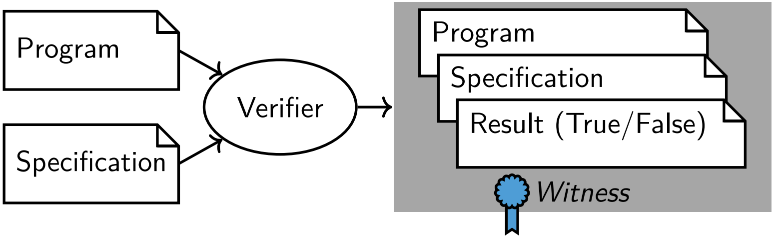
2024-09-10

Outline

- Session 1: 9:00 – 10:30
  - Overview of Concepts and Architecture
  - Installation and Tutorial Examples
  - Overview of the Usage, Inputs, Outputs
- Session 2: 11:00 – 12:30
  - Symbolic Approaches
  - Distributed Summary Synthesis
  - CPA-Daemon
  - Verification Witnesses

Overview of CPAchecker

Software Verification



CPAchecker History

- 2002: BLAST with lazy abstraction refinement [1, 2]
- 2003: Multi-threading support [3]
- 2004: Test-case generation, interpolation, spec. lang. [1, 4]
- 2005: Memory safety, predicated lattices [5, 6]
- 2006: Lazy shape analysis [7]
- Maintenance and extensions became extremely difficult because of design choices that were not easy to revert
- 2007: Configurable program analysis [8, 9],
  CPAchecker was started
  as complete reimplementation from scratch [10]

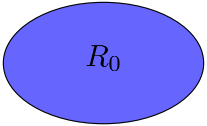
CPAchecker History (2)

- 2009: Large-block encoding [11, FMCAD '09]
- 2010: Adjustable-block encoding [12, FMCAD '10]
- 2012: Conditional model checking [13, FSE '12],
  PredAbs vs. Impact [14, FMCAD '12]
- 2013: Explicit-state MC [15, FASE '13],
  BDDs [16, STTT '14],
  precision reuse [17, FSE '13]
- ...

Software Verification by Model Checking

[18, 19, Clarke/Emerson, Queille/Sifakis 1981]

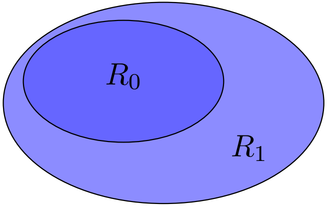
Iterative fixpoint (forward) post computation



Software Verification by Model Checking

[18, 19, Clarke/Emerson, Queille/Sifakis 1981]

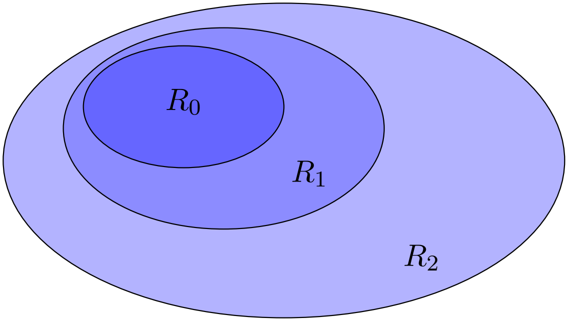
Iterative fixpoint (forward) post computation



Software Verification by Model Checking

[18, 19, Clarke/Emerson, Queille/Sifakis 1981]

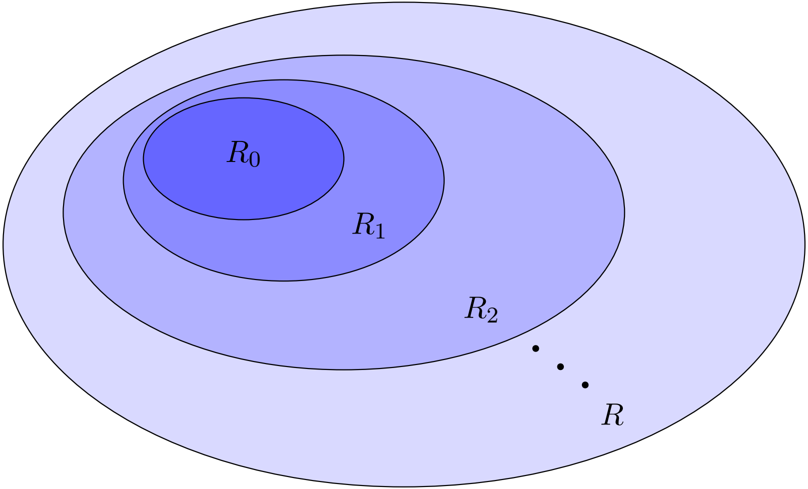
Iterative fixpoint (forward) post computation



Software Verification by Model Checking

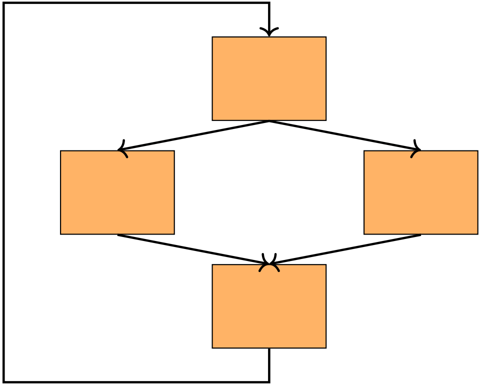
[18, 19, Clarke/Emerson, Queille/Sifakis 1981]

Iterative fixpoint (forward) post computation

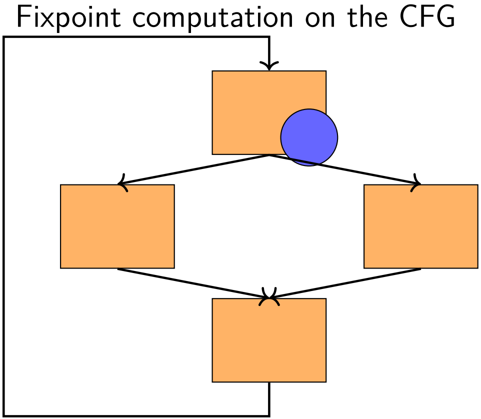


Software Verification by Data-Flow Analysis
[20, Kildall 1973]

Fixpoint computation on the CFG

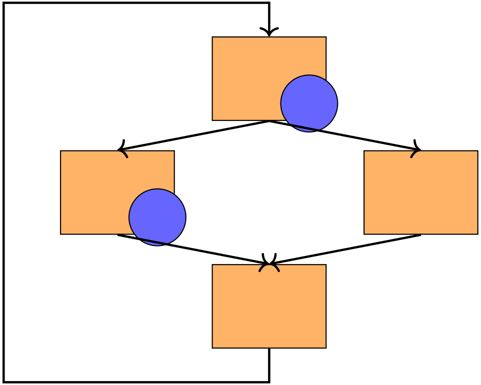


Software Verification by Data-Flow Analysis
[20, Kildall 1973]



Software Verification by Data-Flow Analysis
[20, Kildall 1973]

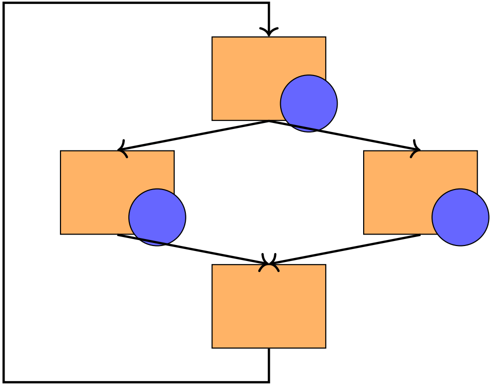
Fixpoint computation on the CFG



Software Verification by Data-Flow Analysis

[20, Kildall 1973]

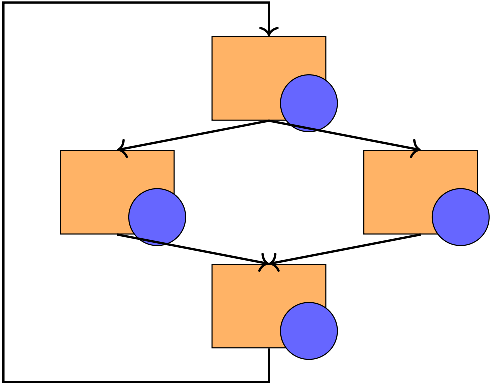
Fixpoint computation on the CFG



Software Verification by Data-Flow Analysis

[20, Kildall 1973]

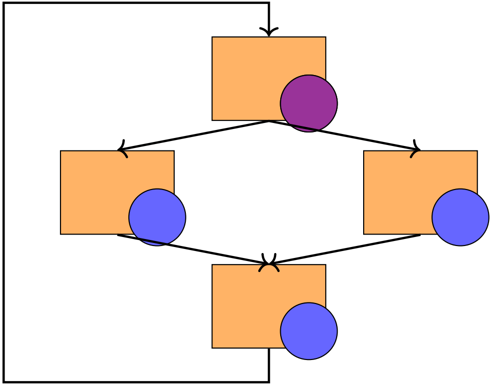
Fixpoint computation on the CFG



Software Verification by Data-Flow Analysis

[20, Kildall 1973]

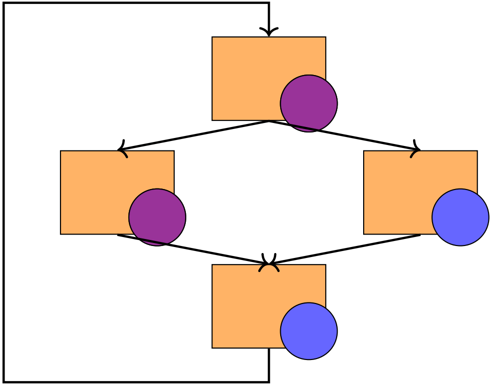
Fixpoint computation on the CFG



Software Verification by Data-Flow Analysis

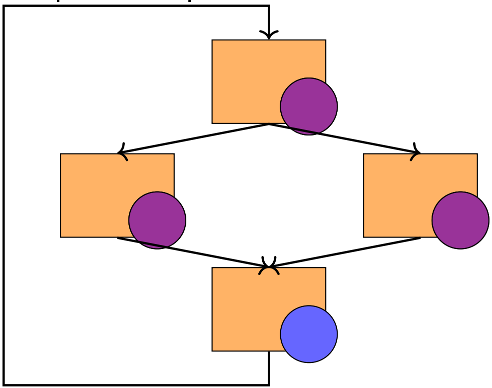
[20, Kildall 1973]

Fixpoint computation on the CFG



Software Verification by Data-Flow Analysis
[20, Kildall 1973]

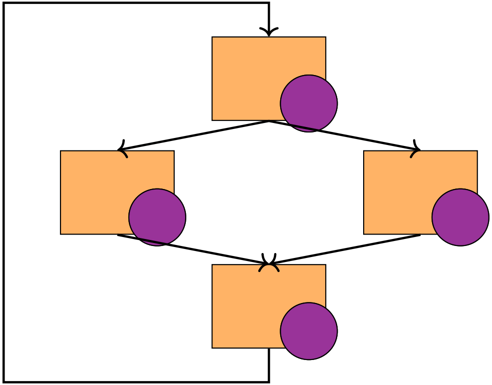
Fixpoint computation on the CFG



Software Verification by Data-Flow Analysis

[20, Kildall 1973]

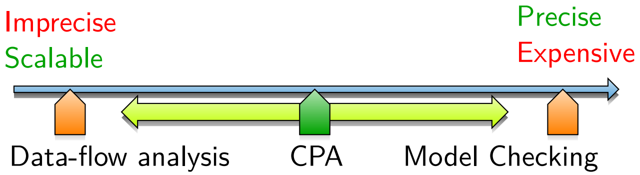
Fixpoint computation on the CFG



Configurable Program Analysis

[8, Beyer/Henzinger/Theoduloz CAV '07]

- Better combination of abstractions
  → Configurable Program Analysis



Unified framework that enables intermediate algorithms

Dynamic Precision Adjustment

Lazy abstraction refinement: [21, Henzinger/Jhala/Majumdar/Sutre POPL '02]

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

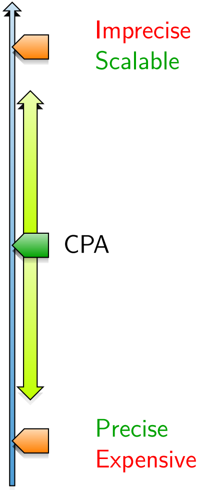
Dynamic Precision Adjustment

Better fine tuning of the precision of abstractions
→ Adjustable Precision
[9, 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**!

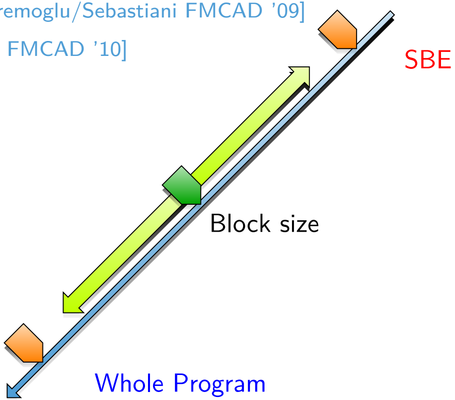


Adjustable Block-Encoding

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

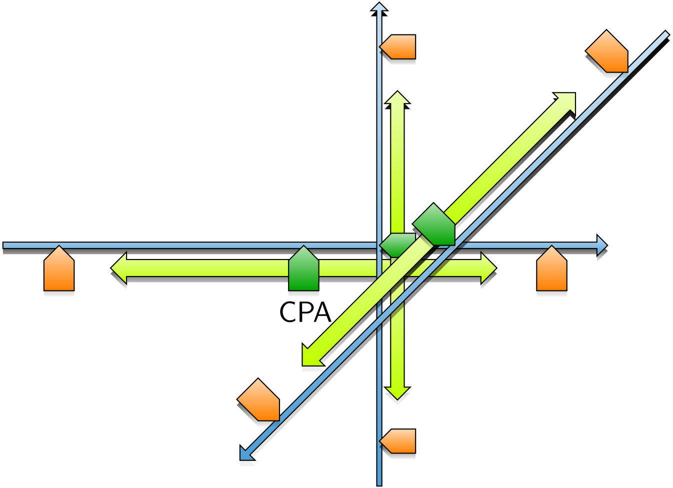
  [11, Beyer/Cimatti/Griggio/Keremoglu/Sebastiani FMCAD '09]

  [12, Beyer/Keremoglu/Wendler FMCAD '10]



CPAchecker

[10, Beyer/Keremoglu CAV '11]



CPA – Summary

- Unification of several approaches
  $\rightarrow$ reduced to their essential properties
- Allow experimentation with new configurations that we could never think of
- Flexible implementation CPACHECKER

CPAchecker

- Framework for Software Verification — current status
  - Written in Java
  - Open Source: Apache 2.0 License
  - ~131 contributors so far from 15 universities/institutions
  - 572.195 lines of code
    (410.470 without blank lines and comments)
  - Started 2007

https://cpachecker.sosy-lab.org

CPAchecker: Features

- Input language C (experimental: Java)
- Web frontend available:
  https://vcloud.sosy-lab.org/cpachecker/webclient/run
- Counterexample output with graphs
- Benchmarking infrastructure available
  (with large cluster of machines)
- Cross-platform: Linux, Mac, Windows

CPAchecker: Achievements

- Among world's best software verifiers:
  https://sv-comp.sosy-lab.org/2021/results/
- Continuous success in competition since 2012
  (66 medals: 19x gold, 22x silver, 25x bronze)
- Awarded Gödel medal
  by Kurt Gödel Society

- Used for Linux driver verification
  with dozens of real bugs found and fixed in Linux [22, 23]

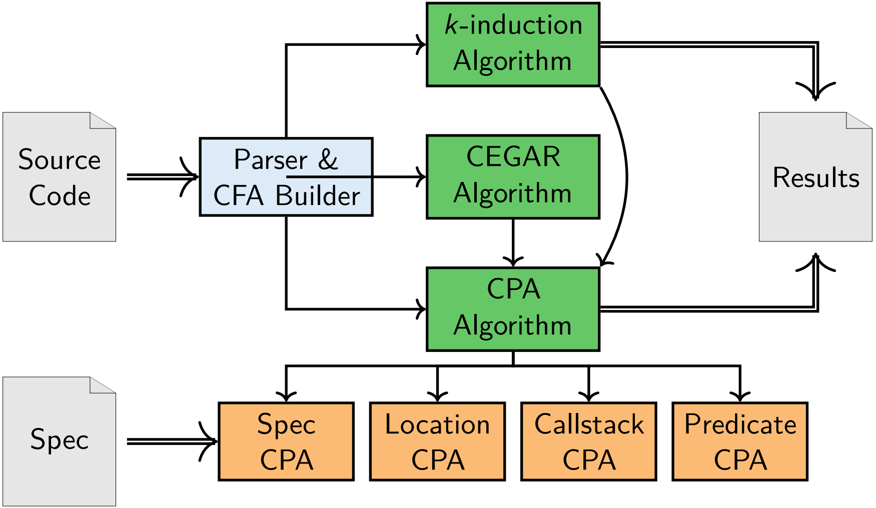
CPAchecker: Concepts

- Included Concepts:
  - CEGAR [24] Interpolation [15, 25]
  - Configurable Program Analysis [8, 9]
  - Adjustable-block encoding [12]
  - Conditional model checking [13]
  - Verification witnesses [26, 27]
  - Various abstract domains: predicates, intervals, BDDs, octagons, explicit values
- Available analyses approaches:
  - Predicate abstraction [11, 12, 9, 17]
  - Impact algorithm [28, 14, 25]
  - Bounded model checking [29, 25]
  - k-Induction [30, 25]
  - IC3/Property-directed reachability [31]
  - Explicit-state model checking [15]
  - Interpolation-based model checking [32]

CPAchecker: Concepts

- 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

CPAchecker: Architecture



Getting Started with CPAchecker

Overview of the First Steps

All the necessary files can be downloaded online OR copied from our USB stick.

- We provide multiple ways on how to get CPAchecker:
  - a pre-compiled version from Zenodo,
  - as a docker image,
  - installation through `.deb` file,
  - as an online service.
- Please download or copy an artifact with all the examples.
- We will show how to run your first analysis.

Installation

We recommend a 64-bit GNU/Linux machine for this tutorial.

Installation via Zenodo (primary)

- This method assumes Java 17+ installed.
- As a first step copy the ZIP file from our local USB stick or download the file from https://doi.org/10.5281/zenodo.12663059
- The file contains the precompiled CPAchecker 3.0 release.

```
cd <path to folder with the .zip file>
unzip CPAchecker-3.0-unix.zip
cd CPAchecker-3.0-unix
```

- When running the examples, the tutorial assumes either adding `bin/` in front of the `cpachecker` or putting `bin/` into the `PATH` environment variable with the following command:

```
export PATH="<path to bin/ folder>:$PATH"
```

Other Ways of Execution

Docker image

```
docker load -i <the .tar.gz file>
docker run -it --rm -v "$(pwd)":/workdir --entrypoint /bin/bash sosylab/cpachecker:3.0
```

`.deb` file

- It requires connection to the internet if the Java package is not installed.

```
sudo apt install ./cpachecker_3.0-1_amd64.deb
```

Website

One can try out the tutorial examples at
https://vcloud.sosy-lab.org/cpachecker/webclient/run/

Example Verification Task example-safe.c

- Download the examples from:
  https://doi.org/10.5281/zenodo.13612338

```
extern unsigned
    __VERIFIER_nondet_uint();
extern void __assert_fail();
int main() {
  unsigned n =
      __VERIFIER_nondet_uint();
  unsigned x =
      __VERIFIER_nondet_uint();
  unsigned y = n - x;
  while(x > y) {
    x--; y++;
    if (x + y != n) {
      __assert_fail();
    }
  }
  return 0;
}
```

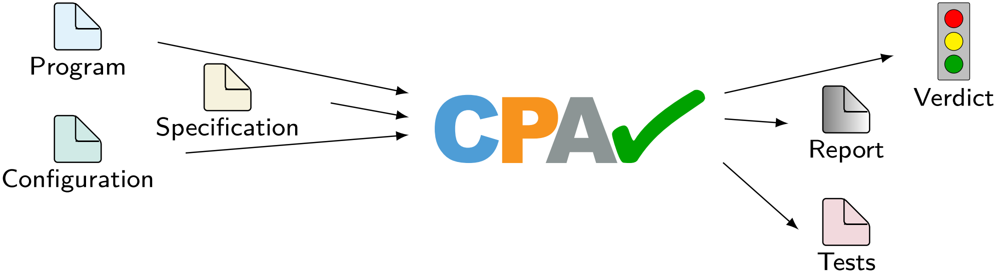
Commands to verify the program:

```
cd <folder of unzipped artifact>
cpachecker ./examples/example-safe.c
```

The expected output:

```
Verification result: TRUE.
No property violation
found by chosen configuration.
More details about the verification
run can be found in the directory "./output".
Graphical representation
included in the file "./output/Report.html".
```

Overview of the Usage, Inputs, Outputs


Program
Specification
Configuration
CPA
Verdict
Report
Tests

Input Program

```
#include<assert.h>




extern int input();



int foo() {
  int a = input();
  int b = input();
  assert(a == b);
}
```

Input Program

```
#include<assert.h>
```
← Must be preprocessed (--preprocess)

```
extern int input();

int foo() {
  int a = input();
  int b = input();
  assert(a == b);
}
```

Input Program

```
extern void __assert_fail (const char *__assertion, const char *__file,
      unsigned int __line, const char *__function)
     __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__noreturn__));

extern int input();

int foo() {
  int a = input();
  int b = input();
  if (a != b) __assert_fail(..);
}
```

Input Program

```
extern void __assert_fail (const char *__assertion, const char *__file,
     unsigned int __line, const char *__function)
    __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__noreturn__));
```

```
extern int input();
```

← CPAchecker assumes undefined functions to be **pure** (with configurable exceptions)

```
int foo() {
  int a = input();
  int b = input();
  if (a != b) __assert_fail(..);
}
```

Input Program

```
extern void __assert_fail (const char *__assertion, const char *__file,
      unsigned int __line, const char *__function)
     __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__noreturn__));

extern int __VERIFIER_nondet_int(); ← New nondet value on each call

int input() {
  return __VERIFIER_nondet_int();    ← Model for input()
}

int foo() {
  int a = input();
  int b = input();
  if (a != b) __assert_fail(..);
}
```

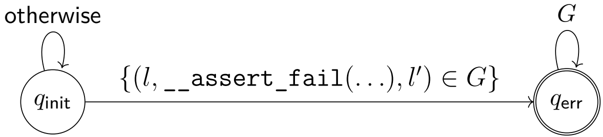
Specification

- `--spec default`
- Support for SV-COMP properties
- Automaton-based specification language (cf. `doc/SpecificationAutomata.md`)

```
OBSERVER AUTOMATON AssertionAutomaton

INITIAL STATE Init;

STATE Init :
  MATCH {__assert_fail($1, $2, $3, $4)}
   -> ERROR("assertion in $location: Condition $1 failed in $2, line $3");
END AUTOMATON
```



Specification

Specification	Description
`ErrorLabel`	Labels named `ERROR` (case insensitive) are never reachable.
`Assertion`	All `assert` statements hold.
`default`	Both `ErrorLabel` and `Assertion` hold.
`overflow`	All operations with a signed-integer type never produce values outside the range representable by the respective type.
`datarace`	Concurrent accesses to the same memory location must be atomic if at least one of them is a write access.
`memorysafety`	All memory deallocations and pointer dereferences are valid and all allocated memory is pointed to or deallocated when the program exits.

More available in `config/specification/`.

CPA✔ Configuration

- What analysis to run? (explicit-value analysis, bounded model checking, . . .)
- Where to start analysis? (function `main`, function `foo`?)
- What machine data model to assume? (32/64bit, x86/ARM?)

Configuration

- Configuration through `key=value` pairs
- Command-line flags: `--option solver.solver=MATHSAT5`
- Input file: `--config valueAnalysis-NoCegar.properties`

```
#include otherFile.properties

# Define tree of CPAs to use
cpa = cpa.arg.ARGCPA
ARGCPA.cpa = cpa.composite.CompositeCPA
CompositeCPA.cpas = cpa.location.LocationCPA, cpa.callstack.CallstackCPA,
  cpa.value.ValueAnalysisCPA, $specification

cpa.value.merge = join
# .. snip ..
```

Configuration

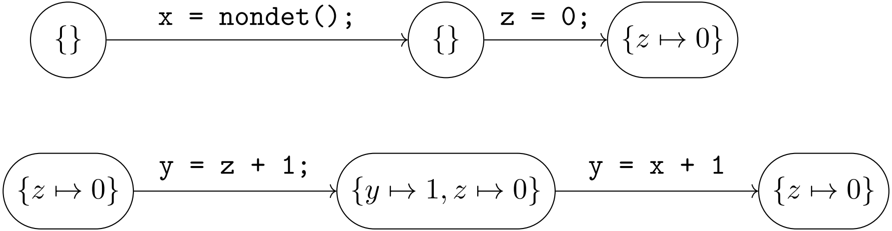
- Predefined analyses in `config/`
- Shorthand: `--config config/bmc.properties`
  $\Leftrightarrow$ `--bmc`
- List of all options: `doc/ConfigurationOptions.txt`

CPA Configuration

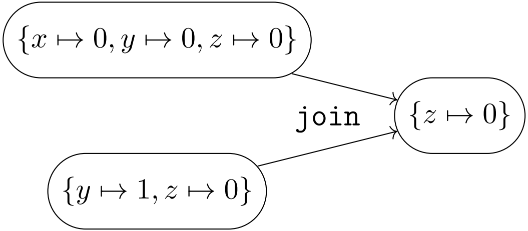
- Analysis entry: `--entry-function foo` (default: main)
- Data model (default: ILP32)
  - `--32`: 32-bit x86 Linux (ILP32)
  - `--64`: 64-bit x86 Linux (LP64)
- Other command-line options:
  - Control timelimit: `--timelimit 30s`
  - Change memory available: `--heap 8000M`
  - `--help`

Program
Specification
Configuration
CPA
Verdict
Report
Tests

Value Analysis CPA



Value Analysis CPA



- Path-insensitive configuration: `--valueAnalysis-NoCegar-join`

Demo

Demo: Path-Insensitive Value Analysis on `example-const.c`

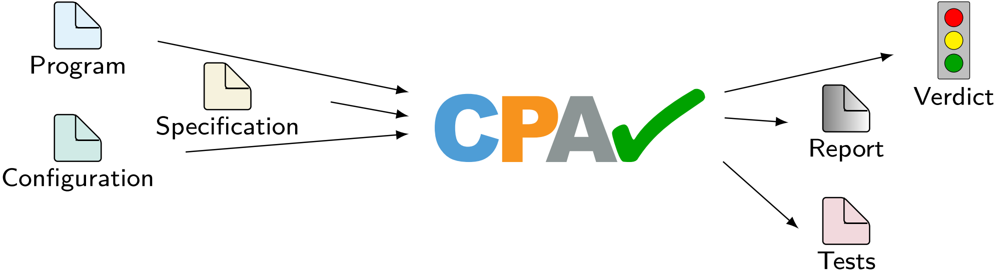
- Example program `example-const.c`
- Command line:
  `cpachecker examples/example-const.c --spec Assertion --valueAnalysis-NoCegar-join`
- Relevant output:
  `output/Report.html`
- In the ‘ARG’ tab:
  Note how the abstract state ‘7 @ N6’ only stores that $z \mapsto 0$.

Demo: Path-Insensitive Value Analysis on `example-safe.c`

- Example program `example-safe.c`
- Command line:
  `cpachecker examples/example-safe.c --spec Assertion --valueAnalysis-NoCegar-join`
- CPAchecker reports UNKNOWN.
- Value analysis finds an error path, but CPAchecker performs a counterexample check on that, finds it infeasible, and removes it.
- Relevant output:
  `output/Report.html`
- In the 'ARG' tab:
  Note how the abstract state '32 @ N8' is *covered by* '3 @ N8'.

Demo: Test Harness for `example-unsafe.c`

- Example program `example-unsafe.c`
- Command line:
  `cpachecker --kInduction examples/example-unsafe.c`
- Relevant output:
  `output/Counterexample.1.harness.c`
  1. `gcc output/Counterexample.1.harness.c examples/example-unsafe.c -o testx`
  2. `./testx` prints violation message


Program
Specification
Configuration
CPA
Verdict
Report
Tests

Questions?

Question session at the end of the tutorial:

Session 2

- Session 1: 9:00 – 10:30
  - Overview of Concepts and Architecture
  - Installation and Tutorial Examples
  - Value Analysis
  - Overview of the Usage, Inputs, Outputs
- Session 2: 11:00 – 12:30
  - Symbolic Approaches
  - Distributed Summary Synthesis
  - CPA-Daemon
  - Verification Witnesses

Predicate Analyses

- Many analyses in CPAchecker [25, 32, 33]
  based on predicates and SMT solvers
  (Predicate Abstraction, k-Induction, Interpolation-Based Model Checking, ISMC, DAR, BMC, Impact)
- Symbolic and precise encoding of program semantics
  (bitvectors, floats, dynamic memory; can be configured if necessary)
- Many SMT solvers supported thanks to JavaSMT [34]:
  MathSAT5, Cvc5, SMTInterpol, Princess, z3
- Default solver is MathSAT5 (academic license)
  due to its strength in bitvectors and interpolation

(Incremental) Bounded Model Checking [35, 25]

- Well known, good for finding bugs
- Unroll loops, check for feasibility, increment loop bound, repeat

Demo: BMC on `example-unsafe.c`

- Example program `example-unsafe.c`
- Command line:
  `cpachecker --bmc-incremental examples/example-unsafe.c`
- Relevant output:
  `expected-outputs/section-4.6-1/output-files/Counterexample.1.html`
- Assert reachable with $n = 3$, $x = 1$, $y = 2$.
- Longer counterexample (with loop unrollings) by changing to
  `unsigned y = x - 5;`

Demo: BMC on `example-safe.c`

- Example program `example-safe.c`
- Command line:
  `cpachecker --bmc-incremental examples/example-safe.c`
- Relevant output:
  `expected-outputs/section-4.6-2/output-files/Report.html`
- Continuous unrolling until timelimit, cf. log and ARG shape

Beyond BMC

- *k*-Induction (`--kInduction`)
  - Tries induction proof, enhanced with auxiliary invariants.
  - Default analysis of CPAchecker
- Interpolation-based extensions of BMC:
  Find fixpoint based on Craig interpolants from SMT solver.
  - Interpolation-based model checking (`--bmc-interpolation`)
  - Interpolation-sequence based model checking (`--bmc-interpolationSequence`)
  - Dual approximated reachability (`--bmc-interpolationDualSequence`)
- Interpolation-based CEGAR approaches:
  - Predicate Abstraction (`--predicateAnalysis`)
  - Impact (`--predicateAnalysis-ImpactRefiner-ABEl`)

Depends on verification task which works best.

Demo: BMC Extensions

- Example program `example-safe.c`
- Command lines:

```
cpachecker --kInduction examples/example-safe.c
cpachecker --bmc-interpolation examples/example-safe.c
cpachecker --bmc-interpolationSequence examples/example-safe.c
cpachecker --bmc-interpolationDualSequence examples/example-safe.c
```

- Rarely useful output except verification verdict

Demo: Predicate Abstraction

- Example program `example-safe.c`
- Command line:
  `cpachecker --predicateAnalysis examples/example-safe.c`
- Relevant output:
  `expected-outputs/section-4.5-1/output-files/`:
  `Report.html`, `witness-2.0.yml`, `invariants.txt`, `predmap.txt`
- Abstract state 25 in ARG covered by abstract state state 16: fixpoint found
- Invariant visible in witness

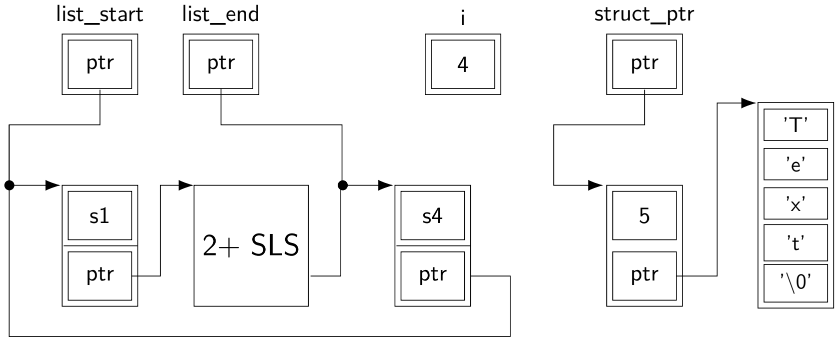
Verification of Memory Safety

- Supported properties defined by SV-COMP (`valid-deref`, `valid-free`, etc.)
- with SV-COMP property file:
  `--spec .../valid-memsafety.prp`
- or manually:
  `--spec memorysafety --smg`
- specialized analysis:
  Symbolic Memory Graphs (SMG) [36]

Symbolic Memory Graphs (SMG)

- Bit-precise heap representation as graph
- Combined with symbolic execution for tracking constraints
- Abstracts linked lists on heap into symbolic representations

Example:



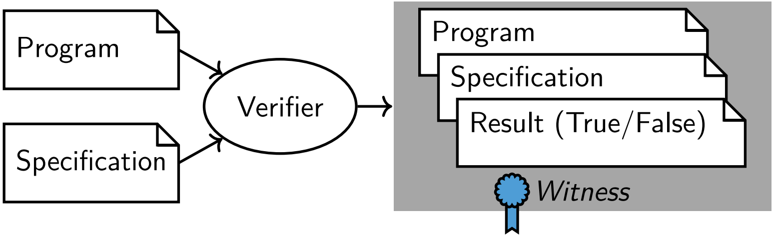
Demo: SMG for Memory Safety

- Example program `example-unsafe-memsafety.c` (needs `--preprocess`)
- Command line:
  ```
  cpachecker --preprocess --smg --spec memorysafety \
  examples/example-unsafe-memsafety.c
  ```
- Relevant output:
  `expected-outputs/section-4.8-1/output-files/Report.html`
- End of path in ARG shows violation in first visit of line 21.
- Fix with `int i = size - 1;` in line 20.

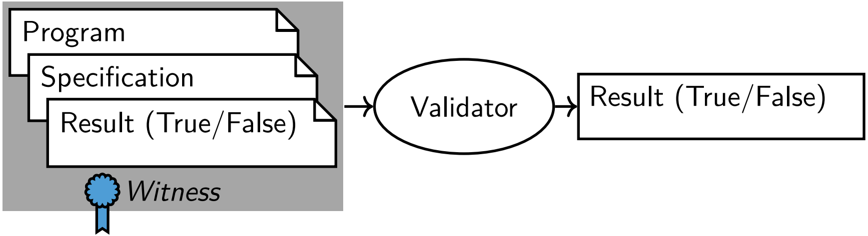
New Features / Recent Developments

Support of Verification Witnesses 2.0

Software Verification



Witness Validation



- Validate untrusted results
- Easier than full verification

Purpose of Witnesses

- Provide insights into the verification process
- Validate verification results
- Exchange information between different tools

Software-Verification Witnesses

Correctness Witnesses:

- Aids in reconstructing the proof
- Contains invariants

Violation Witnesses:

- Aids in replaying a violation
- Represents a set of paths
- At least one is a violation path

Demo Verification Witnesses

Verification (correct example):

```
cpachecker --predicateAnalysis examples/example-safe.c
```

Validation (with correctness witness):

```
cpachecker --witnessValidation --witness output/witness-2.0.yml
examples/example-safe.c
```

Verification (buggy example):

```
cpachecker --predicateAnalysis examples/example-unsafe.c
```

Motivation

- Context: (Automatic) Software Model Checking
- We need low response time.
- Therefore, we need massively parallel approaches.
- Solution: Decomposition into blocks, construct contracts automatically
- Goal: Scalable and Distributed Software Verification

Solution: Distributed Summary Synthesis

Based on [37]:

Dirk Beyer, Matthias Kettl, Thomas Lemberger:

Decomposing Software Verification using Distributed Summary Synthesis

Proc. ACM on Software Engineering, Volume 1, Issue FSE, 2024.
https://doi.org/10.1145/3660766

Overview of Decomposition

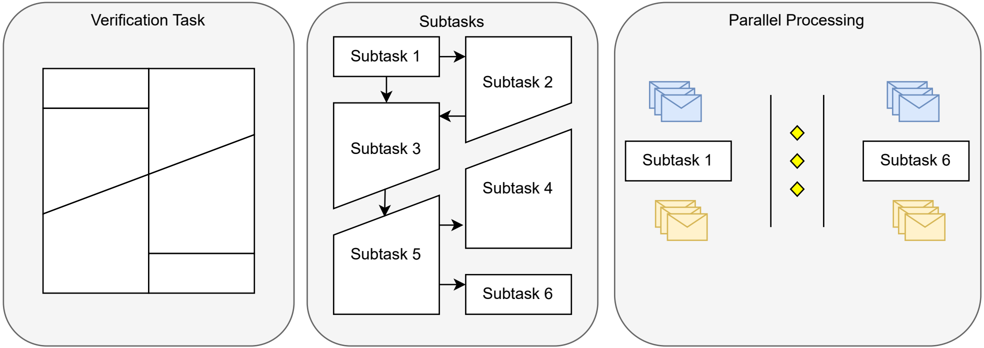


Figure: Overview of the *DSS* approach

Example: Control-Flow Automaton

```
int main() {
  int x = 0;
  int y = 0;
  while (n()) {
    x++;
    y++;
  }
  assert(x == y);
}
```

Figure: Safe program

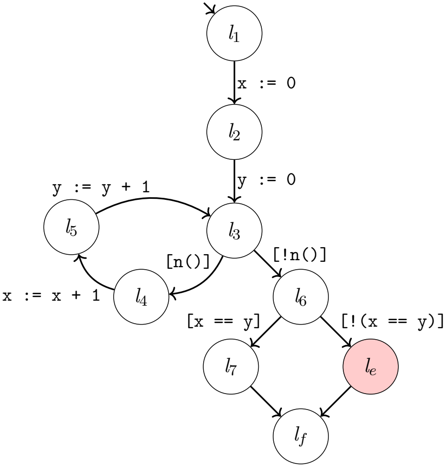


Figure: CFA of program

Decomposition

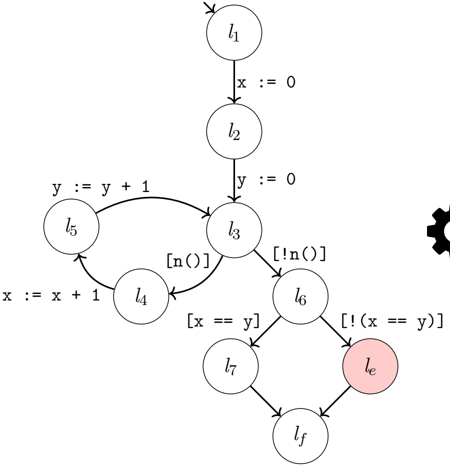
We split a large verification task into multiple smaller subtasks.

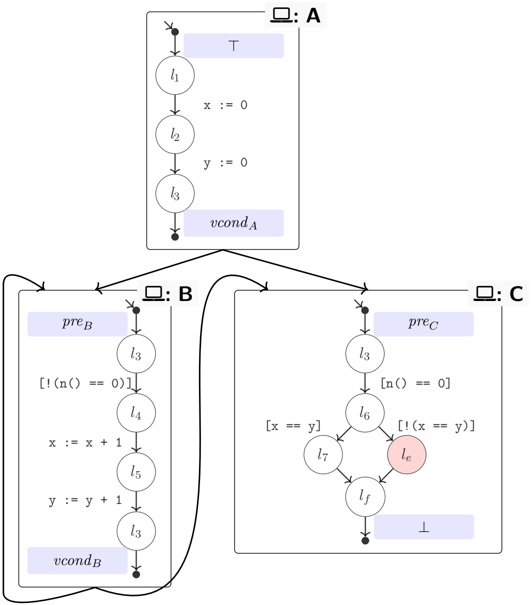
Requirements for eligible decompositions:

- Each block has exactly one entry and one exit location.
- Loops should be reflected as loops in the block graph.
- Blocks should as large as possible.
- Blocks not bound to functions.

Approach: We decompose the CFA similar to large-block encoding [11].

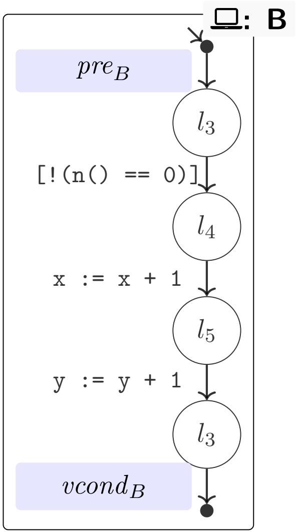
Example: Decomposition



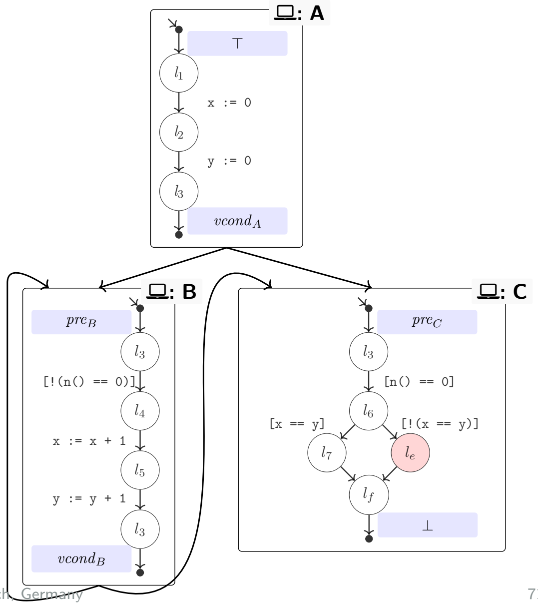


Workers

- Each worker runs independently in an own compute thread/node.
- Preconditions describe good entry states of a block (over-approximating).
- Violation condition needs to be refuted to prove a program safe.
- Preconditions are refined until all violation conditions are refuted or at least one is confirmed.



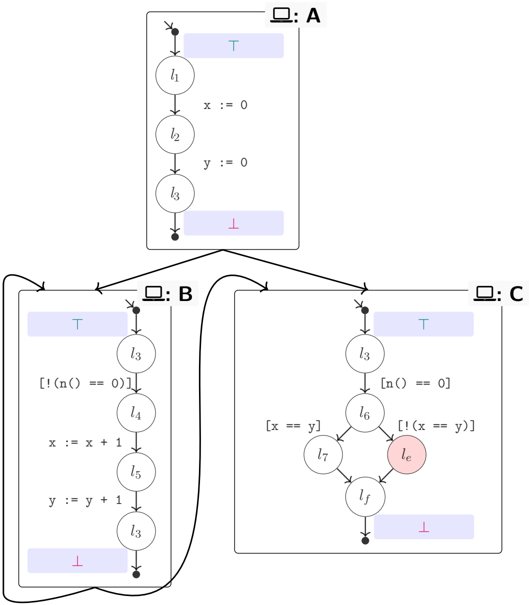
Communication Model



- Workers know their successor and predecessors.
- Workers maintain a list of preconditions, violation conditions, and their subtask.

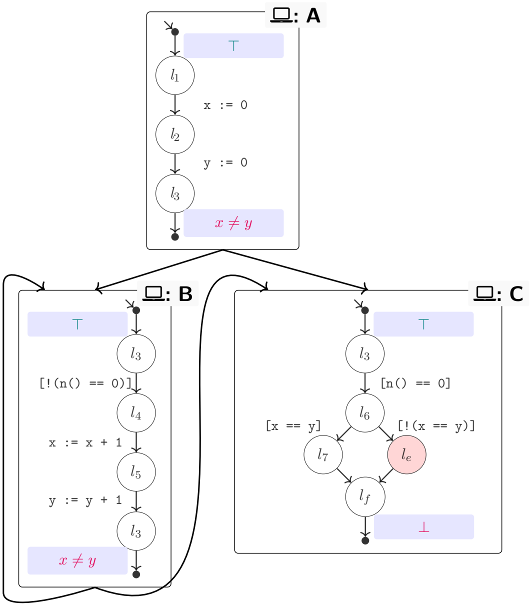
Verification with *DSS* 1

Block	Result
A	$\downarrow$✉$_{B,C}$: $\top$
B	$\downarrow$✉$_{B,C}$: $\top$
C	$\uparrow$✉$_{A,B}$: $x \neq y$



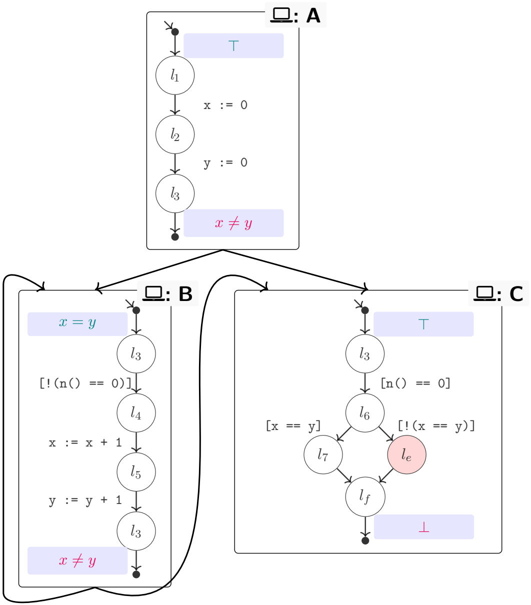
Verification with *DSS* 2

Block	Result
A	$\downarrow$✉$_{B,C}$: $x = y$
B	$\uparrow$✉$_{A,B}$: $x \neq y$
C	*idle*



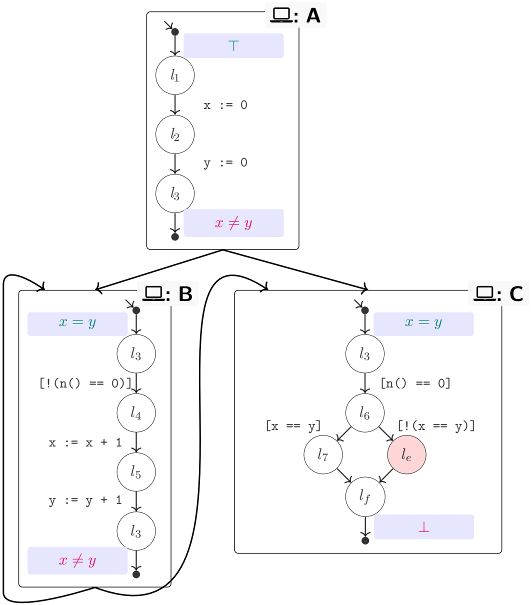
Verification with *DSS* 3

Block	Result
A	$\downarrow$✉$_{B,C}$: $x = y$
B	$\downarrow$✉$_{B,C}$: $x = y$
C	*idle*



Verification with *DSS* 4

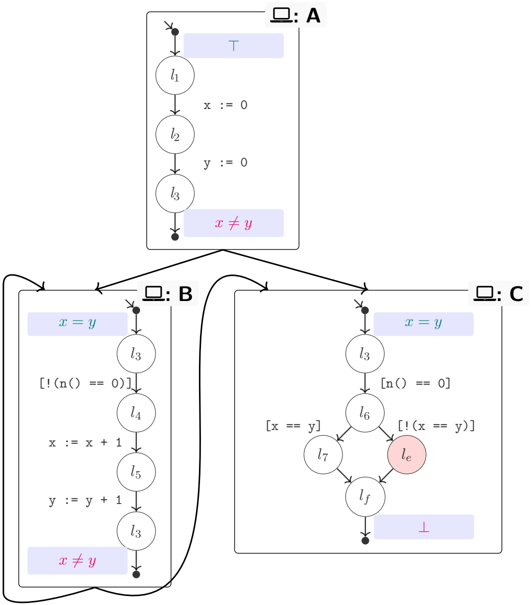
Block	Result
A	*idle*
B	*idle*
C	$\downarrow$✉$_\emptyset$: $\top$



Verification with *DSS* 5

Block	Result
A	*idle*
B	*idle*
C	*idle*

$\Rightarrow$ Fix-point reached, program safe.



Evaluation: Setup

Benchmark Setup:

- We evaluate *DSS* on the subcategory *SoftwareSystems* of the SV-COMP '23 benchmarks.
- We focus on the 2 485 safe verification tasks.
- We use the SV-COMP [38] benchmark setup:
  15 GB RAM and an 8 core Intel Xeon E3-1230 v5 with 3.40 GHz.

Evaluation: Results

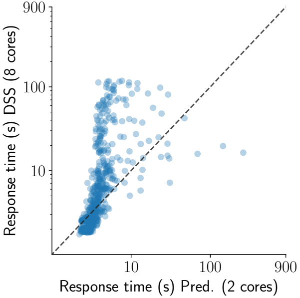


Figure: Response time of predicate abstraction (x-axis) vs. *DSS* (y-axis).

DSS introduces overhead which only pays-off for more complex tasks.
A parallel portfolio combines the best of both worlds.

Evaluation: Distribution of Workload

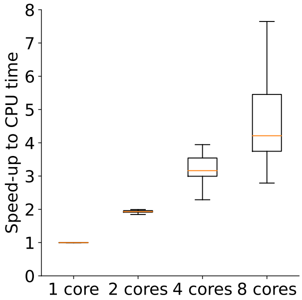


Figure: The ratio of the CPU time and the response time for 1, 2, 4, and 8 cores.

The workload is distributed effectively to multiple processing units.

Evaluation: Outperforming Predicate Analysis

Task	$\text{CPU}_{\mathbb{P}}(s)$	$\text{CPU}_{DSS}(s)$	$\text{RT}_{\mathbb{P}}(s)$	$\textbf{RT}_{DSS}(s)$	# threads
leds–leds-regulator...	44.8	33.2	30.8	7.18	92
rtc–rtc-ds1553.ko-l...	49.0	64.6	30.3	14.0	164
rtc–rtc-stk17ta8.ko...	46.7	67.9	28.9	15.1	162
watchdog–it8712f_w...	86.8	50.3	69.0	15.9	216
ldv-commit-tester/m0...	50.1	103	28.8	21.0	230

DSS introduces overhead which only pays-off for more complex tasks.
A parallel portfolio combines the best of both worlds.

Related Approaches

Existing approaches have limitations that distributed summary synthesis solves, most importantly the potential to scale to many nodes:

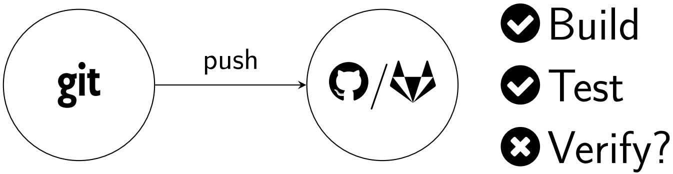
- INFER [39, 40] scales well but reports many false alarms.
  ⇒ *DSS* inherits all properties of the underlying analysis.
- BAM [41] has nested blocks that are not parallelizable.
  ⇒ *DSS* parallelizes as much as possible.
- HIFROG [42] is bound to SMT-based model-checking algorithms.
  ⇒ *DSS* is domain-independent.

Conclusion

- *DSS* can decompose a verification task into independent smaller tasks.
- *DSS* is domain-independent.
- *DSS* effectively distributes the workload to multiple processing units.

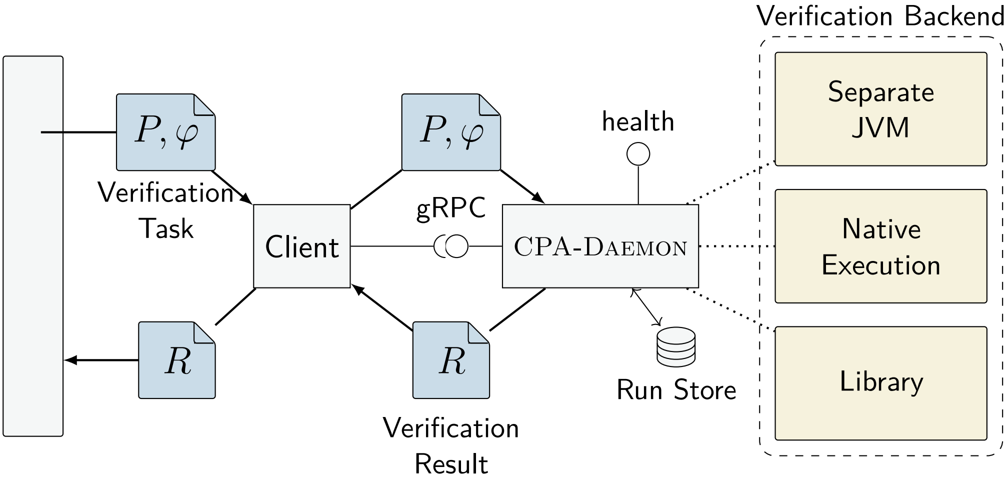
Supplementary webpage

Support CI: CPA-Daemon

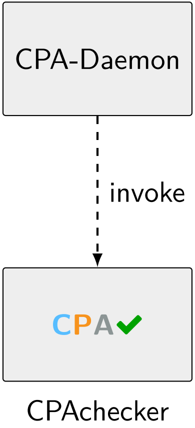


- Challenge: **Machine friendly** interaction.
- Challenge: Fast **response time**.

CPA-Daemon

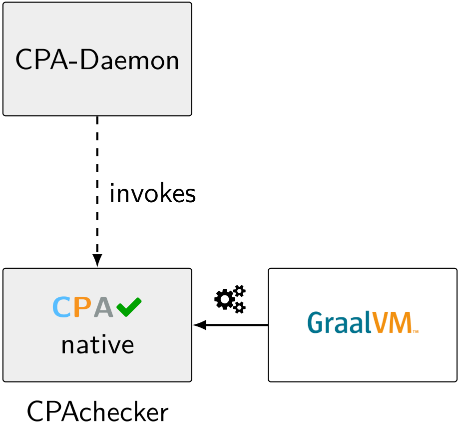

Verification Backend
P, φ
Verification Task
Client
P, φ
gRPC
health
CPA-Daemon
Separate JVM
Native Execution
Library
R
R
Verification Result
Run Store

CPA-Daemon: Separate JVM



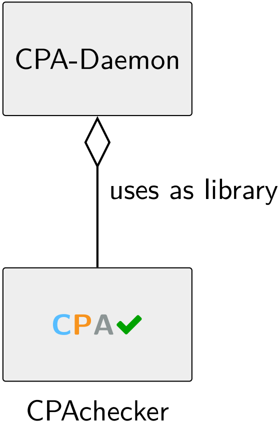
- CPA✔ executed in a fresh JVM
- **Baseline:** Should work just like CPA✔ alone

CPA-Daemon: Native Execution



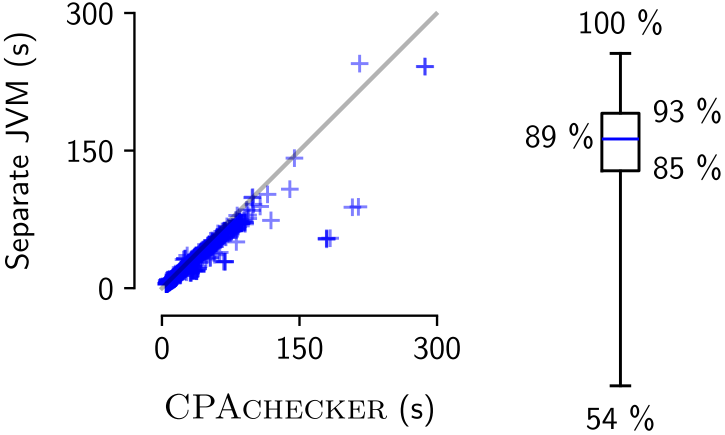
- compile CPA✔ with GraalVM to native binary
- CPA✔ as native ⇒ no JVM need
- ideally: Significant speedup

CPA-Daemon: Library

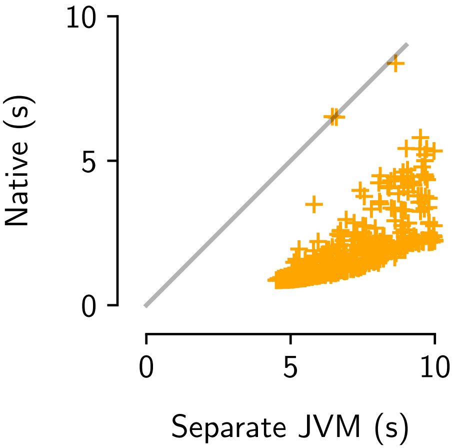


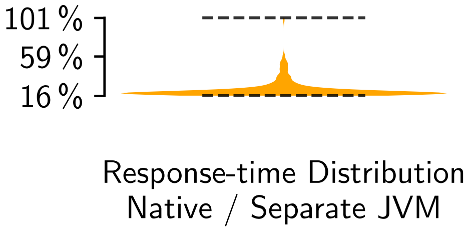
- **CPA-Daemon** is a continuously running service
- CPA✔ executed in the same JVM

Overhead?

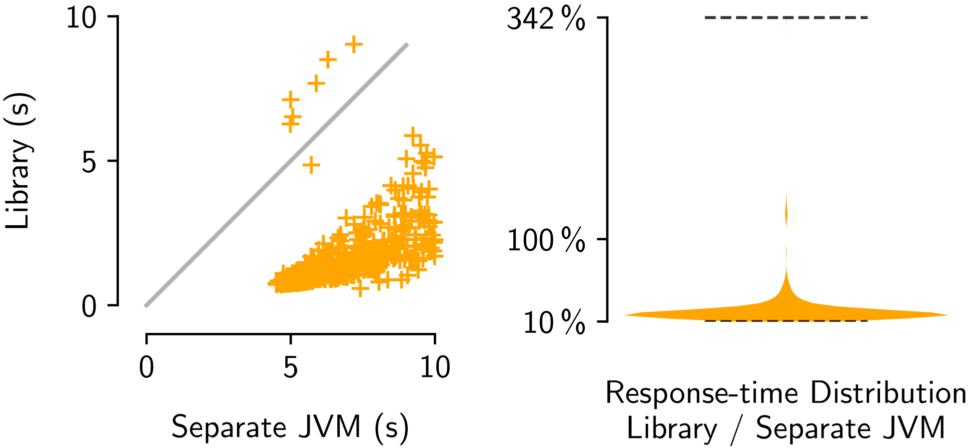


Benefit on Fast-to-solve Tasks

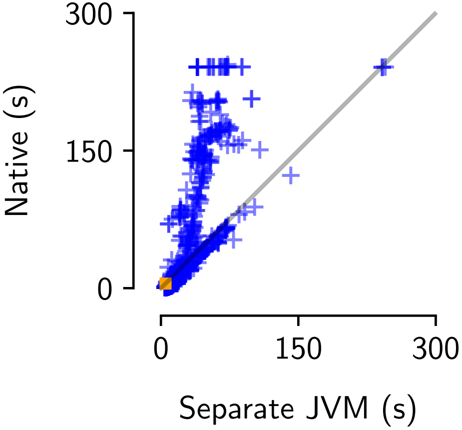


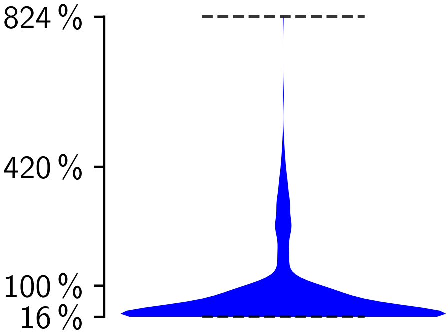


Benefit on Fast-to-solve Tasks



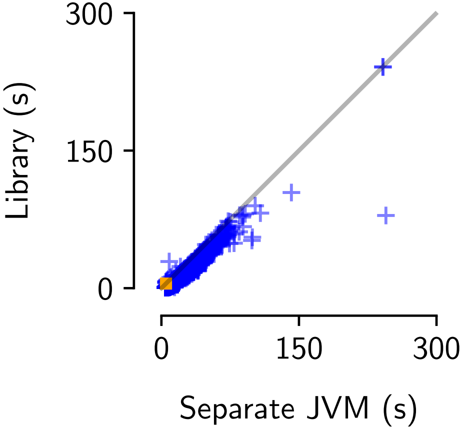
Scalability

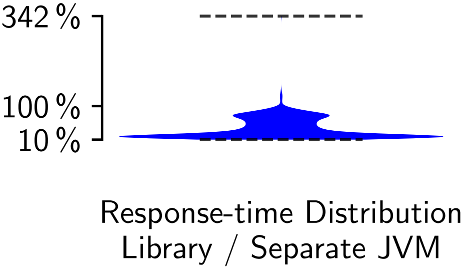




Response-time Distribution
Native / Separate JVM

Scalability





Question Session

Questions:

Feedback

Feedback welcome (same link as for questions):

References I

[1] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. Int. J. Softw. Tools Technol. Transfer **9**(5-6), 505–525 (2007). doi:10.1007/s10009-007-0044-z, https://www.sosy-lab.org/research/pub/2007-STTT.The_Software_Model_Checker_BLAST.pdf

[2] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Proc. SPIN, pp. 235–239. LNCS 2648, Springer (2003). doi:10.1007/3-540-44829-2_17

[3] Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: Proc. CAV. pp. 262–274. LNCS 2725, Springer (2003). doi:10.1007/978-3-540-45069-6_27

References II

[4] Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). doi:10.1109/ICSE.2004.1317455, https://www.sosy-lab.org/research/pub/2004-ICSE.Generating_Tests_from_Counterexamples.pdf

[5] Fischer, J., Jhala, R., Majumdar, R.: Joining data flow with predicates. In: Proc. FSE. pp. 227–236. ACM (2005). doi:10.1145/1081706.1081742

[6] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: Checking memory safety with BLAST. In: Proc. FASE. pp. 2–18. LNCS 3442, Springer (2005). doi:10.1007/978-3-540-31984-9_2, https://www.sosy-lab.org/research/pub/2005-FASE.Checking_Memory_Safety_with_Blast.pdf

References III

[7] Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Proc. CAV. pp. 532–546. LNCS 4144, Springer (2006). doi:10.1007/11817963_48, https://www.sosy-lab.org/research/pub/2006-CAV.Lazy_Shape_Analysis.pdf

[8] Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). doi:10.1007/978-3-540-73368-3_51

[9] Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE. pp. 29–38. IEEE (2008). doi:10.1109/ASE.2008.13

References IV

[10] Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). doi:10.1007/978-3-642-22110-1_16

[11] Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). doi:10.1109/FMCAD.2009.5351147

[12] Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010), https://dl.acm.org/doi/10.5555/1998496.1998532

References V

[13] Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). doi:10.1145/2393596.2393664, https://www.sosy-lab.org/research/pub/2012-FSE.Conditional_Model_Checking.pdf

[14] Beyer, D., Wendler, P.: Algorithms for software model checking: Predicate abstraction vs. IMPACT. In: Proc. FMCAD. pp. 106–113. FMCAD (2012), https://ieeexplore.ieee.org/document/6462562

[15] Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). doi:10.1007/978-3-642-37057-1_11, https://www.sosy-lab.org/research/pub/2013-FASE.Explicit-State_Software_Model_Checking_Based_on_CEGAR_and_Interpolation.pdf

References VI

[16] Beyer, D., Stahlbauer, A.: BDD-based software verification: Applications to event-condition-action systems. Int. J. Softw. Tools Technol. Transfer **16**(5), 507–518 (2014). doi:10.1007/s10009-014-0334-1

[17] Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE. pp. 389–399. ACM (2013). doi:10.1145/2491411.2491429

[18] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Proc. Logic of Programs 1981. pp. 52–71. LNCS 131, Springer (1982). doi:10.1007/BFb0025774

[19] Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Proc. Symposium on Programming. pp. 337–351. LNCS 137, Springer (1982). doi:10.1007/3-540-11494-7_22

References VII

[20] Kildall, G.A.: A unified approach to global program optimization. In: Proc. POPL. pp. 194–206. ACM (1973). doi:10.1145/512927.512945

[21] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL. pp. 58–70. ACM (2002). doi:10.1145/503272.503279

[22] Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). doi:10.1007/978-3-642-11486-1_14

[23] Beyer, D., Petrenko, A.K.: Linux driver verification. In: Proc. ISoLA. pp. 1–6. LNCS 7610, Springer (2012). doi:10.1007/978-3-642-34032-1_1, https://www.sosy-lab.org/research/pub/2012-ISOLA.Linux_Driver_Verification.pdf

References VIII

[24] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). doi:10.1145/876638.876643

[25] Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). doi:10.1007/s10817-017-9432-6

[26] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). doi:10.1145/2786805.2786867

[27] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). doi:10.1145/2950290.2950351

References IX

[28] McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). doi:10.1007/11817963_14

[29] Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). doi:10.1007/978-3-540-24730-2_15

[30] Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). doi:10.1007/978-3-319-21690-4_42

[31] Beyer, D., Dangl, M.: Software verification with PDR: An implementation of the state of the art. In: Proc. TACAS (1). pp. 3–21. LNCS 12078, Springer (2020). doi:10.1007/978-3-030-45190-5_1

References X

[32] Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. arXiv/CoRR **2208**(05046) (July 2022). doi:10.48550/arXiv.2208.05046

[33] Beyer, D., Chien, P.C., Lee, N.Z.: Augmenting interpolation-based model checking with auxiliary invariants. In: Proc. SPIN. Springer (2024)

[34] Baier, D., Beyer, D., Friedberger, K.: JavaSMT 3: Interacting with SMT solvers in Java. In: Proc. CAV. Springer (2021). doi:10.1007/978-3-030-81688-9_9

[35] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). doi:10.1007/3-540-49059-0_14

References XI

[36] Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Proc. SAS. pp. 215–237. LNCS 7935, Springer (2013). doi:10.1007/978-3-642-38856-9_13

[37] Beyer, D., Kettl, M., Lemberger, T.: Decomposing software verification using distributed summary synthesis. Proc. ACM Softw. Eng. **1**(FSE) (2024). doi:10.1145/3660766

[38] Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). doi:10.1007/978-3-031-57256-2_15

References XII

[39] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). doi:10.1007/978-3-319-17524-9_1

[40] Kettl, M., Lemberger, T.: The static analyzer INFER in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 451–456. LNCS 13244, Springer (2022). doi:10.1007/978-3-030-99527-0_30

[41] Beyer, D., Friedberger, K.: Domain-independent interprocedural program analysis using block-abstraction memoization. In: Proc. ESEC/FSE. pp. 50–62. ACM (2020). doi:10.1145/3368089.3409718

References XIII

[42] Alt, L., Asadi, S., Chockler, H., Even-Mendoza, K., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: HiFrog: SMT-based function summarization for software verification. In: Proc. TACAS. pp. 207–213. LNCS 10206 (2017). doi:10.1007/978-3-662-54580-5_12