Benchmarking and Preserving Tools for Formal Methods

Dirk Beyer

2025-09-05, at Boise State University

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

SoSy-Lab Software Systems

Part 1: Reliable Benchmarking

Dirk Beyer, *Stefan Löwe*, and *Philipp Wendler*.
Reliable Benchmarking:
Requirements and Solutions. [1]
STTT 2019

Motivation — Example SV-COMP 2025

- Competition on Software Verification (SV-COMP) [2]
- Largest competition in area of formal methods (consider also SAT-COMP and SMT-COMP)
- 62 verifiers, 18 witness validators
- 33 353 verification tasks
- 942 284 verification runs, 2 312 days of CPU time
- 21.8 million validation runs, 2 573 days of CPU time

Motivation — Example CPAchecker

- Regression tests for development
- 50 tool configurations with each on avg. 4 000 runs
- In total more than 150 million runs in 8 years
- We use BenchCloud [3] with BenchExec [1]
- Together with research and teaching experiments:
  About 1 million executions per week

Evaluation of Research Result

- Result “Theorem”
  Evaluation “Proof”
- Result “Algorithm”
  Evaluation “Algorithm Analysis, properties, Big-O”
- Result “Heuristics for Complex Problems”
  Evaluation “Performance Experiments”

Comparative Evaluation

- Old: Done by competitors
- New: Done by independent competitions

Background: Requirements

Repeatability

- everything documented (machine, version of tool and OS, parameters)
- deterministic tool
- **reliable benchmarking** (here)

Reproducibility

- everything above
- **availability of tool** (FM-Tools), **benchmark set** (SV-COMP), configuration, **environment** (FM-Weck)
- published and archived, appropriate license

Replicability

(not discussed here)

Benchmarking is Important

- Evaluation of new approaches
- Evaluation of tools
- Competitions
- Tool development (testing, optimizations)

Reliable, reproducible, and accurate results needed!

Benchmarking is Hard

- Influence of I/O
- Networking
- Distributed tools
- User input

Benchmarking is Hard

- ~~Influence of I/O~~
- ~~Networking~~
- ~~Distributed tools~~
- ~~User input~~

Not relevant for
most verification tools → Easy?

Benchmarking is Hard

- ~~Influence of I/O~~
- ~~Networking~~
- ~~Distributed tools~~
- ~~User input~~

Not relevant for
most verification tools

- Different hardware architectures
- Heterogeneity of tools
- Parallel benchmarks

Relevant!

Goals

- Reproducibility
  - Avoid non-deterministic effects and interferences
  - Provide defined set of resources
- Accurate results
- For verification tools (and similar)
- On Linux

Checklist

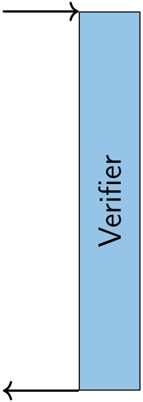
1. Measure and Limit Resources Accurately
   - Time
   - Memory
2. Terminate Processes Reliably
3. Assign Cores Deliberately
4. Respect Non-Uniform Memory Access
5. Avoid Swapping
6. Isolate Individual Runs
   - Communication
   - File system

Measure and Limit Resources Accurately

- Wall time and CPU time
- Define memory consumption
  - Size of address space? Too large
  - Size of heap? Too low
  - Size of resident set (RSS)?
- Measure peak consumption
- Always define memory limit for reproducibility
- Include sub-processes

Measuring CPU time with "`time`"

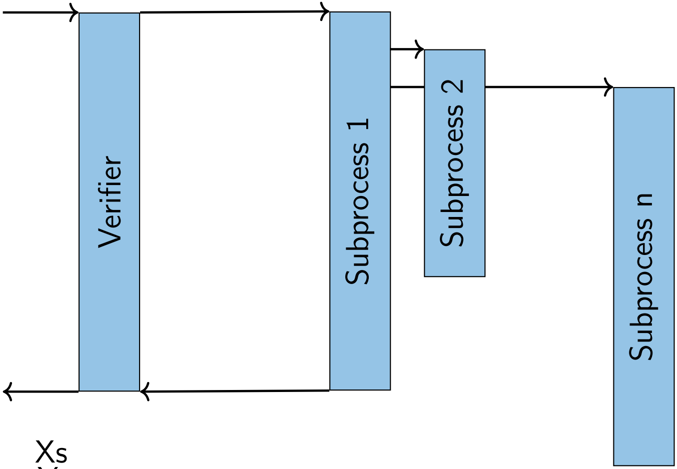
```
~$ time verifier
```



```
real    Xs
user    Ys
sys     Zs
```

Measuring CPU time with “time”

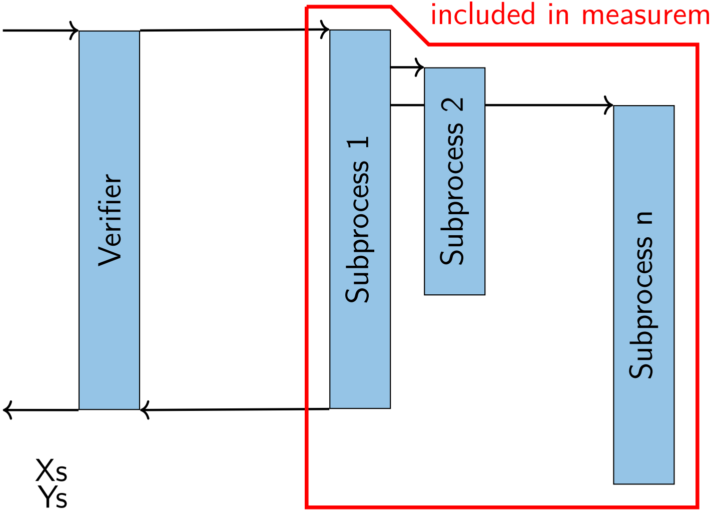
```
~$ time verifier
```



```
real    Xs
user    Ys
sys     Zs
```

Measuring CPU time with "`time`"

```
~$ time verifier
```

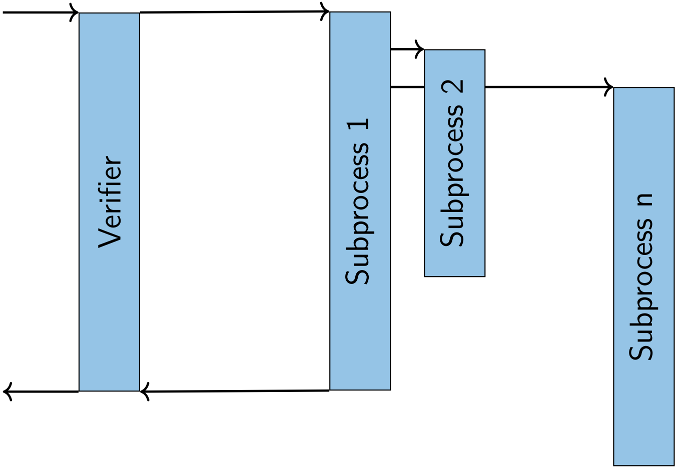


CPU time may not be included in measurement

```
real    Xs
user    Ys
sys     Zs
```

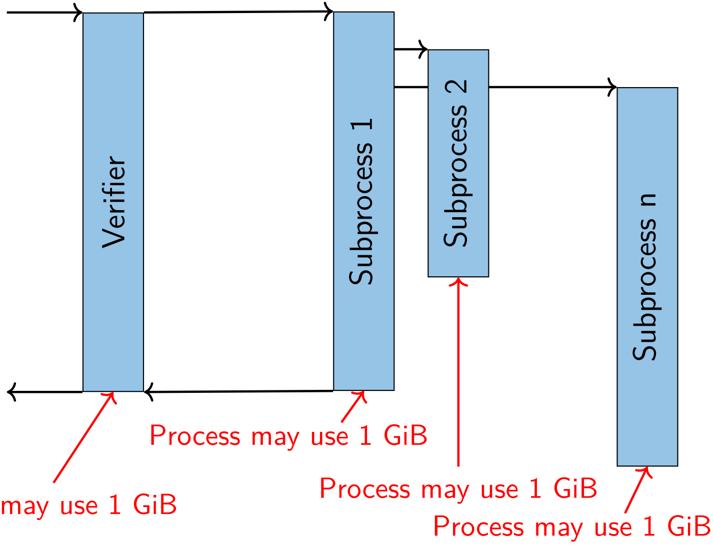
Limiting memory with “ulimit”

```
~$ ulimit -v 1048576 # 1 GiB
~$ verifier
```



Limiting memory with “ulimit”

```
~$ ulimit -v 1048576 # 1 GiB
~$ verifier
```



Verifier

Subprocess 1

Subprocess 2

Subprocess n

Process may use 1 GiB

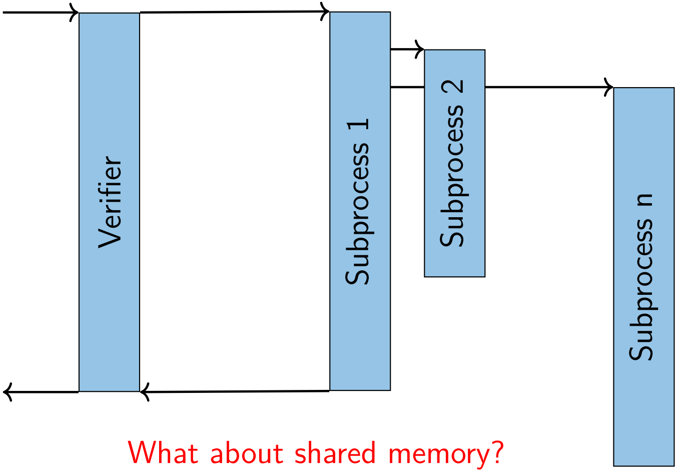
Process may use 1 GiB

Process may use 1 GiB

Process may use 1 GiB

Limiting memory with "ulimit"

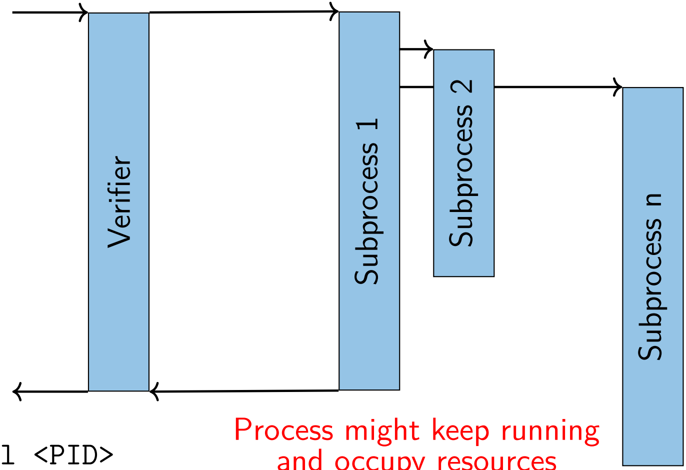
```
~$ ulimit -v 1048576 # 1 GiB
~$ verifier
```



What about shared memory?

Terminate Processes Reliably

```
~$ verifier
```



```
~$ kill <PID>
```

Process might keep running
and occupy resources

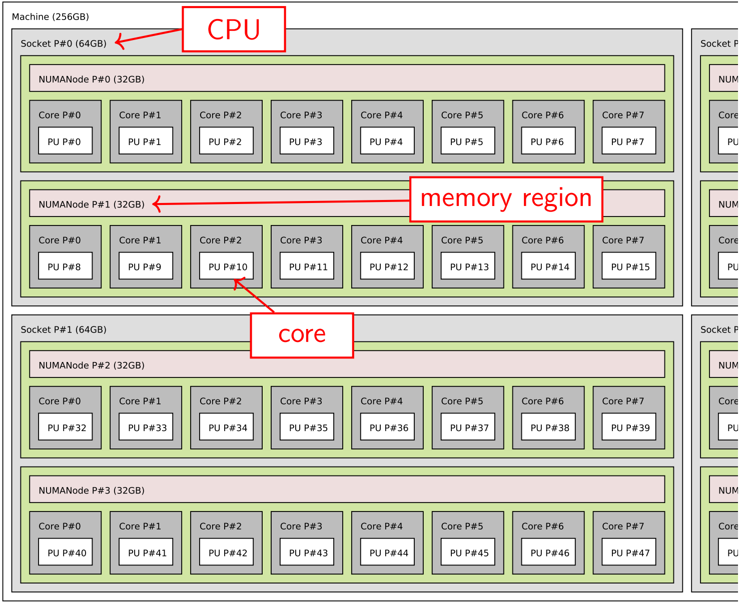
Assign Cores Deliberately

- Hyper Threading:
  Multiple threads sharing execution units
- Shared caches

Respect Non-Uniform Memory Access (NUMA)

- Memory regions have different performance depending on current CPU core
- Hierarchical NUMA makes things worse

Type `lstopo` on your machine (Ubuntu: package `hwloc`)



Machine (256GB)

Socket P#0 (64GB)

NUMANode P#0 (32GB)

Core P#0 PU P#0 | Core P#1 PU P#1 | Core P#2 PU P#2 | Core P#3 PU P#3 | Core P#4 PU P#4 | Core P#5 PU P#5 | Core P#6 PU P#6 | Core P#7 PU P#7

NUMANode P#1 (32GB)

Core P#0 PU P#8 | Core P#1 PU P#9 | Core P#2 PU P#10 | Core P#3 PU P#11 | Core P#4 PU P#12 | Core P#5 PU P#13 | Core P#6 PU P#14 | Core P#7 PU P#15

Socket P#1 (64GB)

NUMANode P#2 (32GB)

Core P#0 PU P#32 | Core P#1 PU P#33 | Core P#2 PU P#34 | Core P#3 PU P#35 | Core P#4 PU P#36 | Core P#5 PU P#37 | Core P#6 PU P#38 | Core P#7 PU P#39

NUMANode P#3 (32GB)

Core P#0 PU P#40 | Core P#1 PU P#41 | Core P#2 PU P#42 | Core P#3 PU P#43 | Core P#4 PU P#44 | Core P#5 PU P#45 | Core P#6 PU P#46 | Core P#7 PU P#47

Socket P

NUM

Core

PU

NUM

Core

PU

Socket P

NUM

Core

PU

NUM

Core

PU

CPU

memory region

core

Isolate Individual Runs

- Excerpt of start script taken from some verifier in SV-COMP:

  # . . . (tool started here)

  ```
  killall z3 2> /dev/null
  killall minisat 2> /dev/null
  killall yices 2> /dev/null
  ```

- Thanks for thinking of cleanup

Isolate Individual Runs

- Excerpt of start script taken from some verifier in SV-COMP:

  # . . . (tool started here)

  ```
  killall z3 2> /dev/null
  killall minisat 2> /dev/null
  killall yices 2> /dev/null
  ```

- Thanks for thinking of cleanup
- But what if there are parallel runs?

Isolate Individual Runs

- Temp files with constant names like `/tmp/mytool.tmp` collide
- State stored in places like `~/.mytool` hinders reproducibility
  - Sometimes even auto-generated
- Restrict changes to file system as far as possible

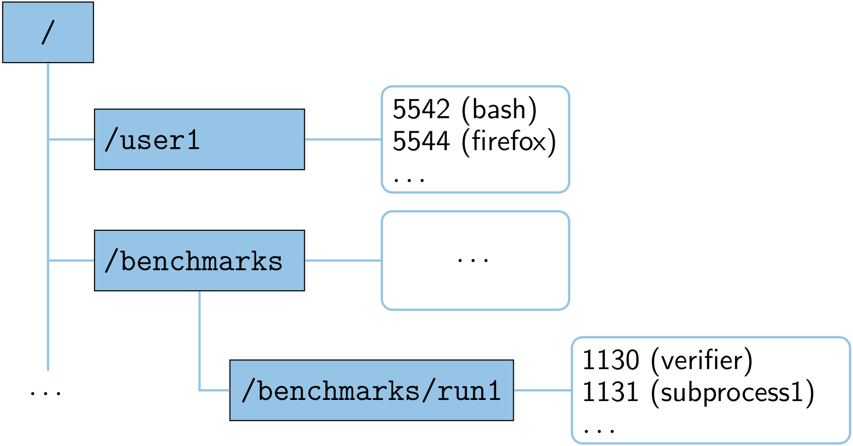
Cgroups

- Linux kernel “control groups”
- Reliable tracking of spawned processes
- Resource limits and measurements per cgroup
  - CPU time
  - Memory
  - I/O etc.

Solution on Linux
for race-free handling of multiple processes!

Cgroups

- Hierarchical tree of sets of processes



/

/user1

5542 (bash)
5544 (firefox)
...

/benchmarks

...

/benchmarks/run1

1130 (verifier)
1131 (subprocess1)
...

...

Namespaces

- Light-weight virtualization
- Only one kernel running, no additional layers
- Change how processes see the system
- Identifiers like PIDs, paths, etc. can have different meanings in each namespace
  - PID 42 can be a different process in each namespace
  - Directory / can be a different directory in each namespace
  - ...
- Can be used to build application containers without possibility to escape
- Usable without root access

Overlay File System

- Protect file system from changes made by subject tool
- Allow subject tool to write to specific folders
- Collect what is written to a folder (into the layer)
- Easy clean-up after execution of the subject tool

Overlay FS — Possible Directory Access Modes

	Read	**Write temp**	**Write persistent**
	existing content	content	content
hidden	✗	✓	✗
read only	✓	✗	✗
overlay	✓	✓	✗
full access	✓	✗	✓

Benchmarking Containers

- Encapsulate groups of processes
- Limited resources (memory, cores)
- Total resource consumption measurable
- All other processes hidden
  and no communication with them
- Disabled network access
- Adjusted file-system layout
  - Private `/tmp`
  - Writes redirected to
    temporary RAM disk

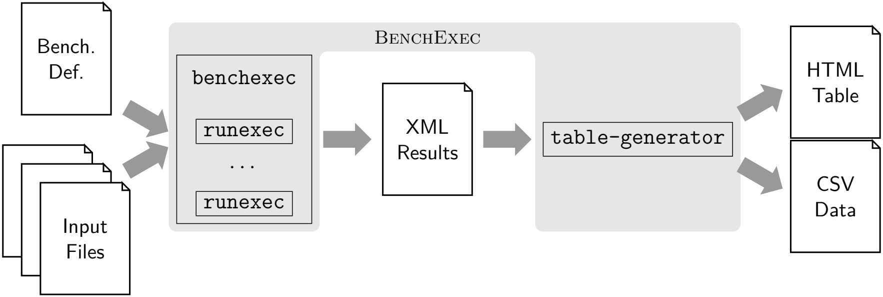
BenchExec

- A Framework for Reliable Benchmarking and Resource Measurement
- Provides benchmarking containers based on cgroups, namespaces, overlay FS
- Allocates hardware resources appropriately
- Low system requirements (modern Linux kernel and cgroups access)

BenchExec

- Open source: Apache 2.0 License
- Written in Python 3
- https://github.com/sosy-lab/benchexec
- Used in International Competition on Software Verification (SV-COMP) and by StarExec
- Originally developed for software-verification, but applicable to arbitrary tools

BenchExec Architecture



`runexec`
: Benchmarks a single run of a tool (in container)

`benchexec`
: Benchmarks multiple runs

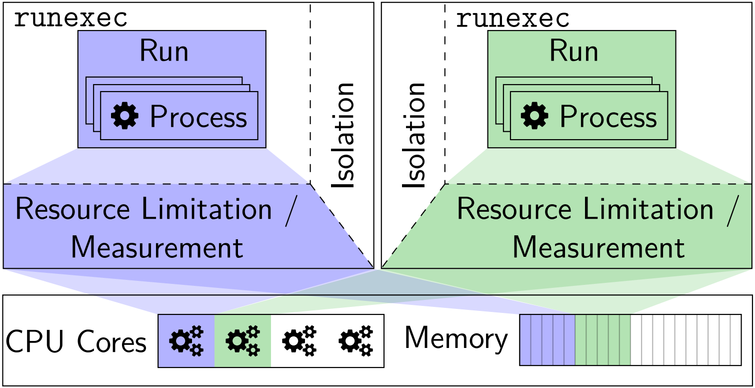
`table-generator`
: Generates CSV and interactive HTML tables

BenchExec: `runexec`

- Benchmarks a single run of a tool
- Measures and limits resources using cgroups
- Runnable as stand-alone tool and as Python module
- Easy integration into other benchmarking frameworks and infrastructure
- Example:

```
runexec --timelimit 100 --memlimit 16000000000
        --cores 0-7,16-23 --memoryNodes 0
        --<TOOL_CMD>
```

BenchExec: `runexec`



BenchExec: `benchexec`

- Benchmarks multiple runs
  (e.g., a set of configurations against a set of files)
- Allocates hardware resources
- Can check whether tool result is as expected
  for given input file and property

BenchExec: `table-generator`

- Aggregates results
- Extracts statistic values from tool output
- Generates CSV and interactive HTML tables (with plots)
- Computes result differences and regression counts

BenchExec Configuration

- Tool command line
- Expected result
- Resource limits
  - CPU time, wall time
  - Memory
- Container setup
  - Network access
  - File-system layout
- Where to put result files

Please Read More

Dirk Beyer, *Stefan Löwe*, and *Philipp Wendler*.
Reliable Benchmarking:
Requirements and Solutions. [1]
STTT 2019

- More details
- Study of hardware influence on benchmarking results
- Suggestions how to present results
  (result aggregation, rounding, plots, etc.)

Conclusion — Part 1: BenchExec

Be careful when benchmarking!

Don't use `time`, `ulimit`, etc.
Always use cgroups and namespaces!

BenchExec
https://github.com/sosy-lab/benchexec

Part 2: Preserving Tools

Dirk Beyer.
Find, Use, and Conserve Tools for Formal Methods. [4]
Proc. Podelski 65th 2024

Dirk Beyer, and *Henrik Wachowitz*.
FM-Weck: Containerized Execution of Formal-Methods Tools. [5]
FM 2024

Vision

- All tools for formal methods work together to solve hard verification problems and make our world safer and more secure.
- Model checkers and theorem provers can be integrated into the software-development process as seamless as unit testing today.
- Model checkers, theorem provers, SMT solvers, and testers use common interfaces for interaction and composition.

Some Steps Towards the Vision

- **Find**: Which tools for software verification exist?
- ... for test-case generation?
- ... for SMT solving?
- ... for hardware verification?
- **Reuse**: How to get executables?
- Where to find documentation?
- Am I allowed to use it?
- How to use them?
- **Conserve**: Which operating system, libraries, environment?

Requirements for Solution

- Support documentation and reuse
- Easy to query and generate knowledge base
- Long-term availability/executability of tools
- Must come with tool support
- Approach must be compatible with competitions

Solution [4]

One central repository:
https://gitlab.com/sosy-lab/benchmarking/fm-tools
which gives information about:

- Location of the tool (via DOI, just like other literature)
- License
- Contact (via ORCID)
- Project web site
- Options
- Requirements (certain Docker container / VM)
- Limits

Maintained by formal-methods community

Example: Entry for CPAchecker

```
id: cpachecker
name: CPAchecker
description: |
  CPAchecker is a configurable framework for
  software verification that is based on
  configurable program analysis and ...
input_languages:
  - C
project_url: https://cpachecker.sosy-lab.org
repository_url:
   https://gitlab.com/sosy-lab/software/cpachecker
spdx_license_identifier: Apache-2.0
benchexec_toolinfo_module:
   benchexec.tools.cpachecker
fmtools_format_version: "2.0"
fmtools_entry_maintainers:
  - dbeyer
  - ricffb
  - PhilippWendler
```

Example: CPAchecker's Contacts

```yaml
maintainers:
  - orcid: 0000-0003-4832-7662
    name: Dirk Beyer
    institution: LMU Munich
    country: Germany
    url: https://www.sosy-lab.org/people/dbeyer/
  - orcid: 0000-0002-5139-341X
    name: Philipp Wendler
    institution: LMU Munich
    country: Germany
    url: https://www.sosy-lab.org/people/wendler/
```

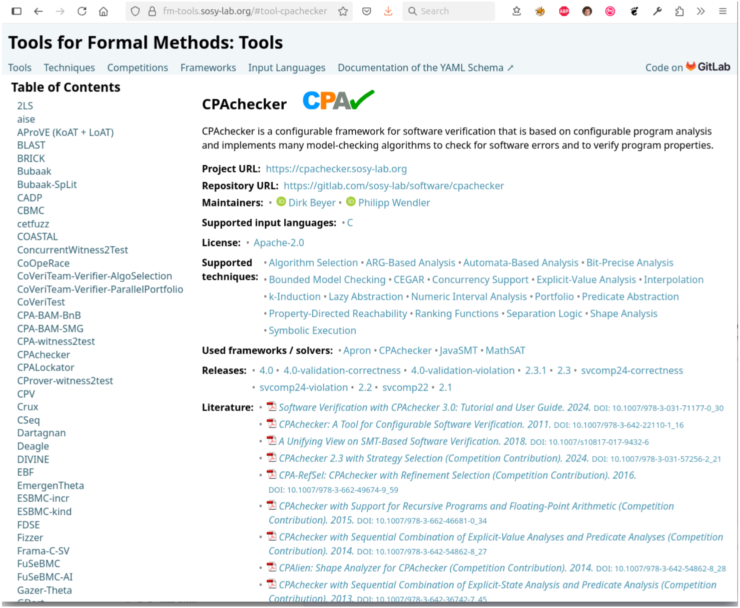
Example: CPAchecker's Versions

```
versions:
  - version: "4.0"
    doi: 10.5281/zenodo.14203369
    benchexec_toolinfo_options: ["--svcomp25",
       "--heap", "10000M", "--benchmark",
       "--timelimit", "900␣s"]
    required_ubuntu_packages:
      - openjdk-17-jdk-headless
    base_container_images:
      - docker.io/ubuntu:22.04
  - version: "4.0-validation-correctness"
    doi: 10.5281/zenodo.14203369
    benchexec_toolinfo_options: ["--witness",
       "${witness}",
       "--correctness-witness-validation",
       "--heap", "5000m", "--benchmark", ...]
    required_ubuntu_packages:
      - openjdk-17-jdk-headless
    base_container_images:
      - docker.io/ubuntu:22.04
```

Example: CPAchecker's Documentation

```
literature:
  - doi: 10.1007/978-3-031-71177-0_30
    title: "Software␣Verification␣with␣CPAchecker␣
        3.0:␣Tutorial␣and␣User␣Guide"
    year: 2024
  - doi: 10.1007/978-3-642-22110-1_16
    title: "CPAchecker:␣A␣Tool␣for␣Configurable␣
        Software␣Verification"
    year: 2011
  - doi: 10.1007/s10817-017-9432-6
    title: "A␣Unifying␣View␣on␣SMT-Based␣Software␣
        Verification"
    year: 2018
```

Example: CPAchecker's Web-Page Entry



Tools for Formal Methods: Tools

Tools Techniques Competitions Frameworks Input Languages Documentation of the YAML Schema ↗ Code on GitLab

Table of Contents

CPAchecker CPA✓

CPAchecker is a configurable framework for software verification that is based on configurable program analysis and implements many model-checking algorithms to check for software errors and to verify program properties.

Project URL: https://cpachecker.sosy-lab.org

Repository URL: https://gitlab.com/sosy-lab/software/cpachecker

Maintainers: • Dirk Beyer • Philipp Wendler

Supported input languages: • C

License: • Apache-2.0

Supported techniques: • Algorithm Selection • ARG-Based Analysis • Automata-Based Analysis • Bit-Precise Analysis • Bounded Model Checking • CEGAR • Concurrency Support • Explicit-Value Analysis • Interpolation • k-Induction • Lazy Abstraction • Numeric Interval Analysis • Portfolio • Predicate Abstraction • Property-Directed Reachability • Ranking Functions • Separation Logic • Shape Analysis • Symbolic Execution

Used frameworks / solvers: • Apron • CPAchecker • JavaSMT • MathSAT

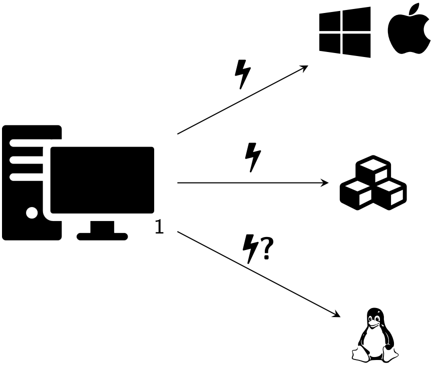
Releases: • 4.0 • 4.0-validation-correctness • 4.0-validation-violation • 2.3.1 • 2.3 • svcomp24-correctness • svcomp24-violation • 2.2 • svcomp22 • 2.1

Literature:
- *Software Verification with CPAchecker 3.0: Tutorial and User Guide. 2024.* DOI: 10.1007/978-3-031-71177-0_30
- *CPAchecker: A Tool for Configurable Software Verification. 2011.* DOI: 10.1007/978-3-642-22110-1_16
- *A Unifying View on SMT-Based Software Verification. 2018.* DOI: 10.1007/s10817-017-9432-6
- *CPAchecker 2.3 with Strategy Selection (Competition Contribution). 2024.* DOI: 10.1007/978-3-031-57256-2_21
- *CPA-RefSel: CPAchecker with Refinement Selection (Competition Contribution). 2016.* DOI: 10.1007/978-3-662-49674-9_59
- *CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic (Competition Contribution). 2015.* DOI: 10.1007/978-3-662-46681-0_34
- *CPAchecker with Sequential Combination of Explicit-Value Analyses and Predicate Analyses (Competition Contribution). 2014.* DOI: 10.1007/978-3-642-54862-8_27
- *CPAlien: Shape Analyzer for CPAchecker (Competition Contribution). 2014.* DOI: 10.1007/978-3-642-54862-8_28
- *CPAchecker with Sequential Combination of Explicit-State Analysis and Predicate Analysis (Competition Contribution). 2013.* DOI: 10.1007/978-3-642-36742-7_45

FM-Tools is FAIR

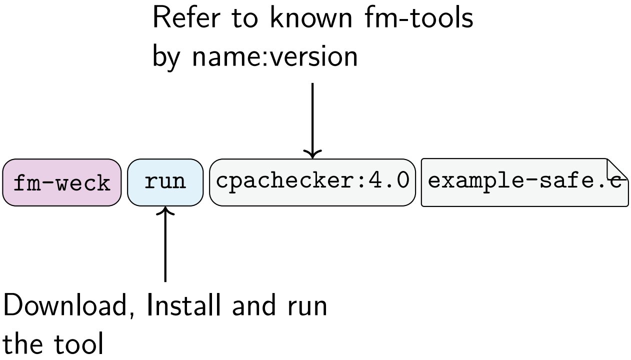
- **F**indable:
  overview is available on internet,
  generated knowledge base
- **A**ccessible:
  data retrievable via Git, format is YAML
- **I**nteroperable:
  Format is defined in schema,
  archives identified by DOIs, researchers by ORCIDs
- **R**eusable:
  Data are CC-BY, each tool comes with a license,
  format of tool archive standardized

What about the Environment?



[1] Image: Flaticon.com

FM-Weck: Run Tools in Conserved Environment [5, Proc. FM 2024]



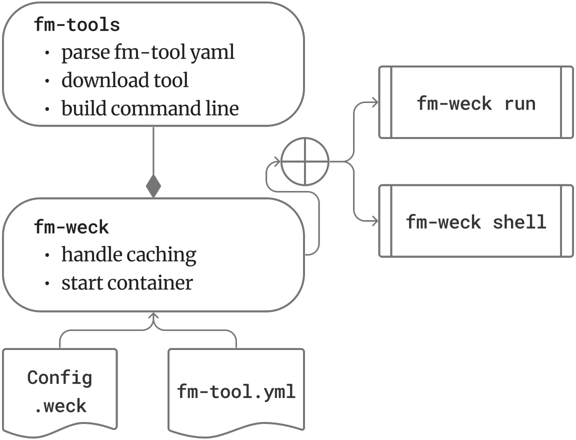
Refer to known fm-tools by name:version

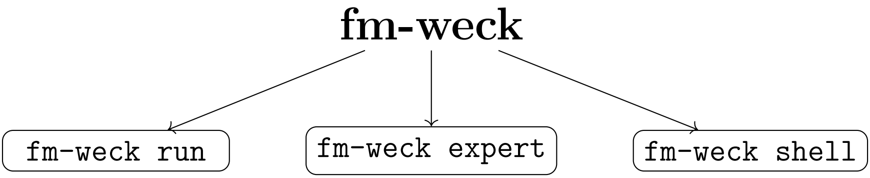
```
fm-weck run cpachecker:4.0 example-safe.c
```

Download, Install and run the tool

- No knowledge of the tools CLI needed
- Tool runs in a container (no dependencies on host system)

FM-Weck: Architecture





fm-weck

`fm-weck run`

`fm-weck expert`

`fm-weck shell`

- Download and execute tool in container
- No knowledge of tool needed

- Download and execute tool in container
- Expert knowledge about tool required

- Spin up interactive shell in tool environment

Conclusion — Part 2: FM-Tools and FM-Weck

FM-Tools collects and stores essential information to:

- Generate a knowledge base about formal-methods tools [4]
  https://fm-tools.sosy-lab.org
- Conserve tool versions and their required environment (with help by Zenodo and Podman/Docker)
- Run a tool in conserved environment via FM-Weck [5]
- Please add your tool

https://fm-tools.sosy-lab.org

References I

[1] Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). doi:10.1007/s10009-017-0469-y

[2] Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS (3). pp. 151–186. LNCS 15698, Springer (2025). doi:10.1007/978-3-031-90660-2_9

[3] Beyer, D., Chien, P.C., Jankola, M.: BenchCloud: A platform for scalable performance benchmarking. In: Proc. ASE. pp. 2386–2389. ACM (2024). doi:10.1145/3691620.3695358

[4] Beyer, D.: Find, use, and conserve tools for formal methods. In: Proc. Festschrift Podelski 65th Birthday. Springer (2024). `https://www.sosy-lab.org/research/pub/2024-Podelski65.Find_Use_and_Conserve_Tools_for_Formal_Methods.pdf`

[5] Beyer, D., Wachowitz, H.: FM-Weck: Containerized execution of formal-methods tools. In: Proc. FM. pp. 39–47. LNCS 14934, Springer (2024). doi:10.1007/978-3-031-71177-0_3