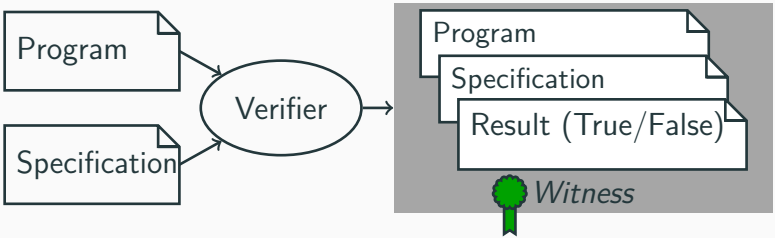
# Towards
# Scalable and Distributed Software Verification

**Dirk Beyer**

LMU Munich, Germany

Huawei Workshop 2024
Grenoble
2024-09-17

# Automatic Software Verification




Mostly context-sensitive, whole-program analysis

# Verification of Software for Pratical Applications

- **Part 1: Distributed Summary Synthesis**
  - ⇒ Modular
  - ⇒ Distributed
  - ⇒ Precise
- **Part 2: Containerized Execution of Verifiers**
  - ⇒ Verifier-agnostic verification
  - ⇒ Zero config
  - ⇒ Container isolation

# Motivation — Part 1

- 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

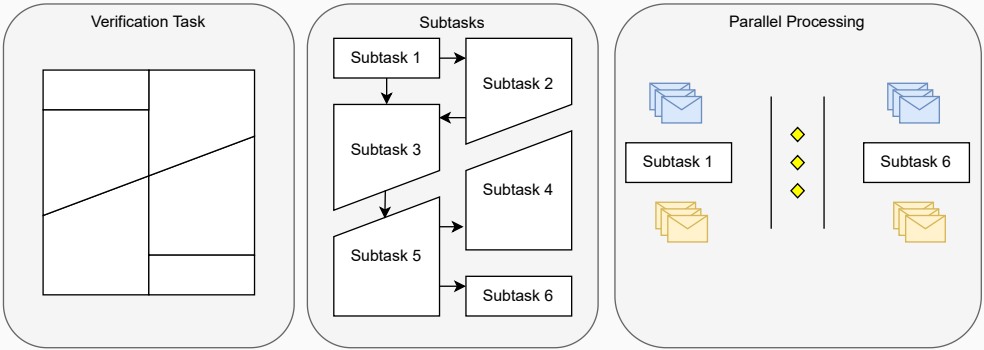
Based on [7]:

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

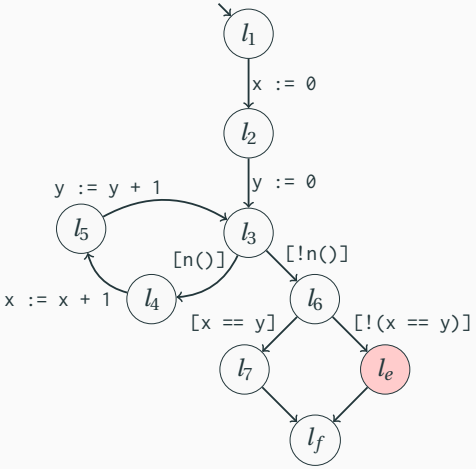



Overview of the *DSS* approach

# Example: Control-Flow Automaton

```
1 int main() {
2   int x = 0;
3   int y = 0;
4   while (n()) {
5     x++;
6     y++;
7   }
8   assert(x == y);
9 }
```

Safe program




CFA of program

# Decomposition

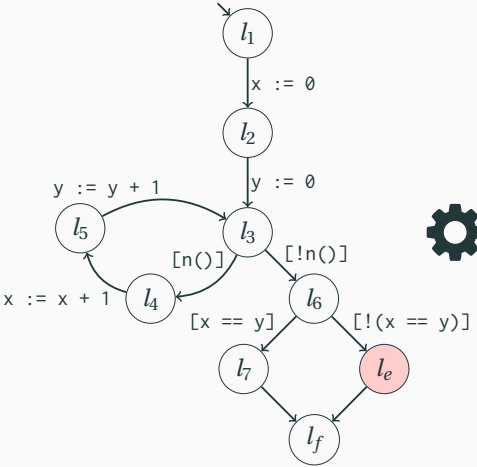
We split a large verification task into multiple smaller subtasks.

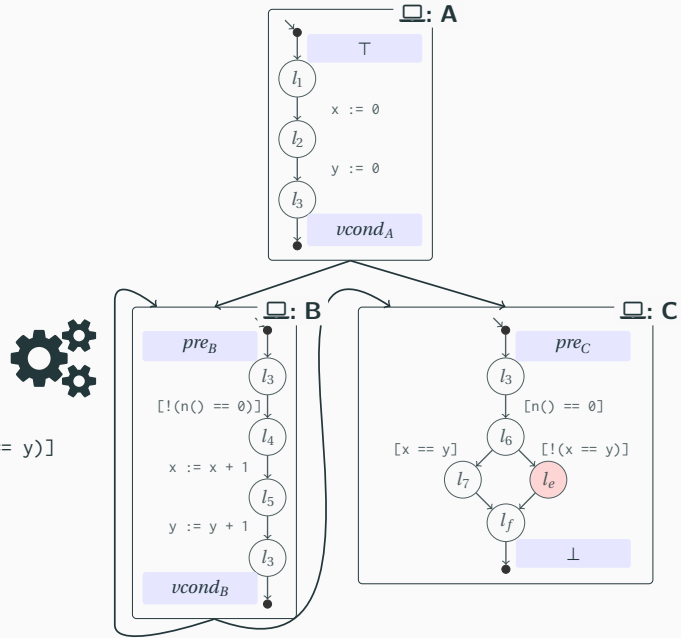
Requirements for eligible decompositions:

- 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 [4].

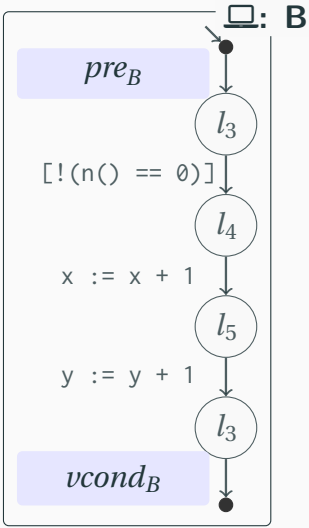
# Example: Decomposition



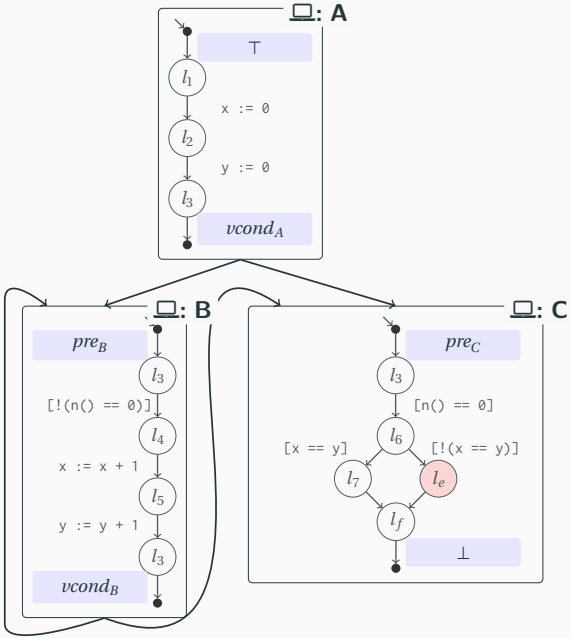
# Workers

- 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.



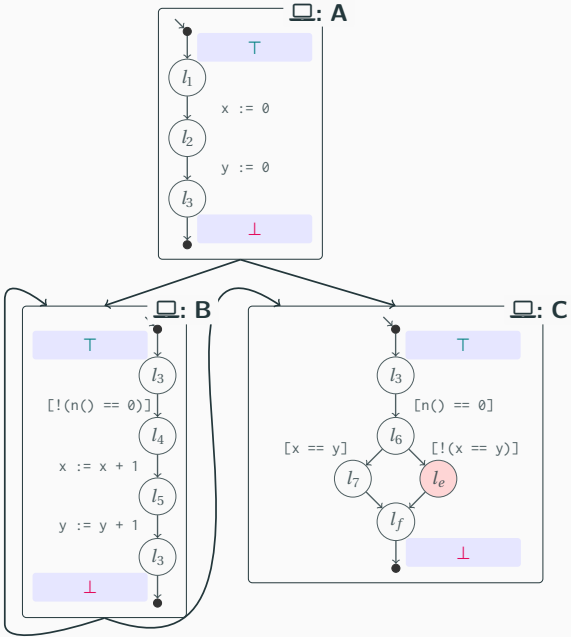
# Communication Model

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



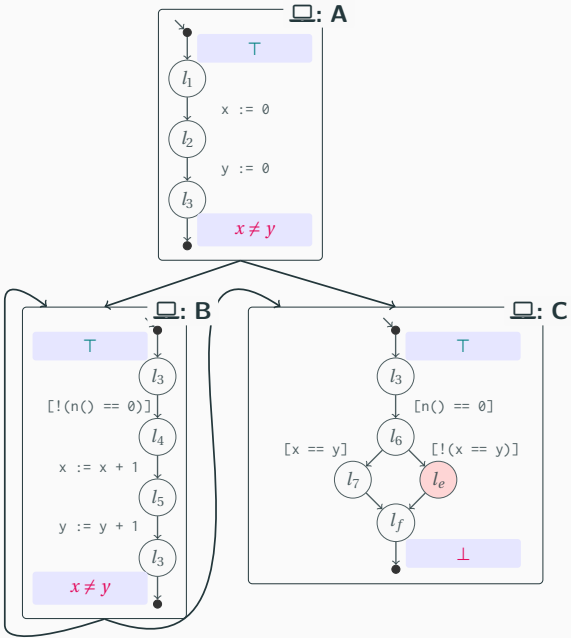
# Verification with *DSS* 1

| Block | Result |
|---|---|
| A | $\downarrow\!\boxtimes_{B,C} : \top$ |
| B | $\downarrow\!\boxtimes_{B,C} : \top$ |
| C | $\uparrow\!\boxtimes_{A,B} : x \neq y$ |



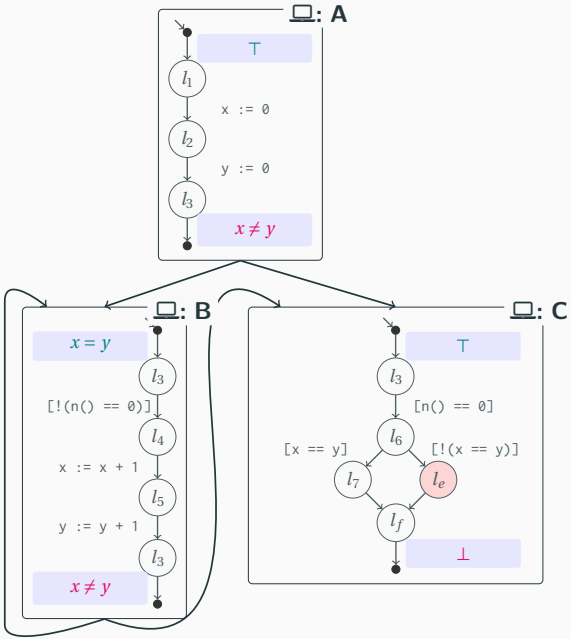
# Verification with *DSS* 2

| Block | Result |
|---|---|
| A | $\downarrow \boxtimes_{B,C} : x = y$ |
| B | $\uparrow \boxtimes_{A,B} : x \neq y$ |
| C | *idle* |



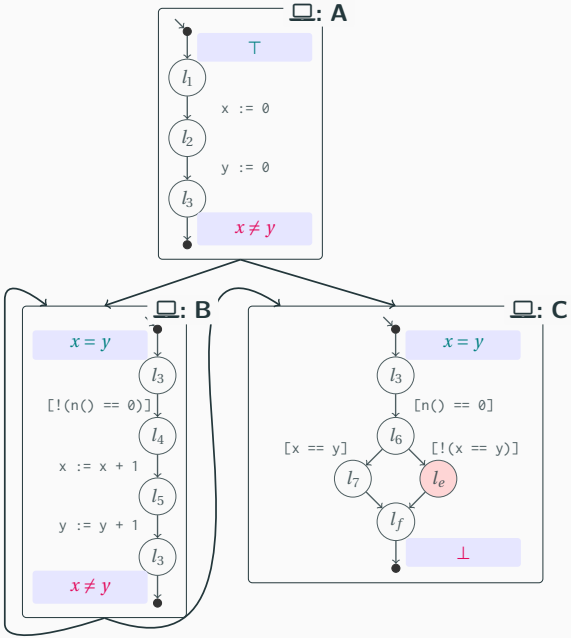
# Verification with *DSS* 3

| Block | Result |
|---|---|
| A | $\downarrow\!\boxtimes_{B,C} : x = y$ |
| B | $\downarrow\!\boxtimes_{B,C} : x = y$ |
| C | *idle* |



# Verification with *DSS* 4

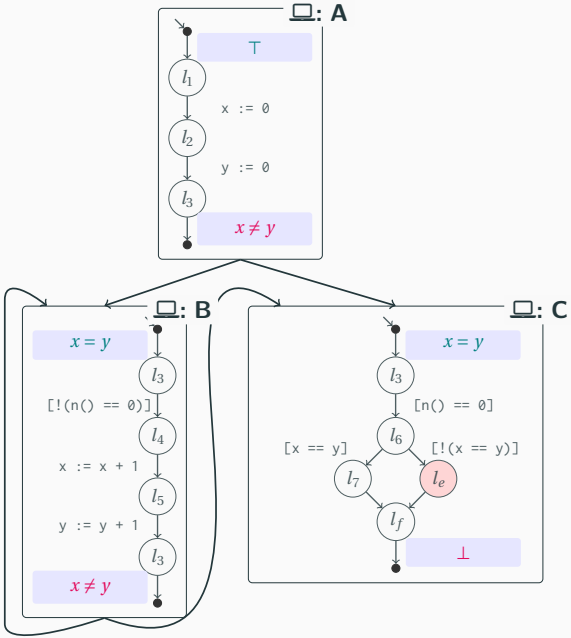
| Block | Result |
|---|---|
| A | *idle* |
| B | *idle* |
| C | $\downarrow\boxtimes_{\emptyset} : \top$ |



# Verification with *DSS* 5

| Block | Result |
|---|---|
| A | *idle* |
| B | *idle* |
| C | *idle* |

⇒ Fix-point reached, program safe.

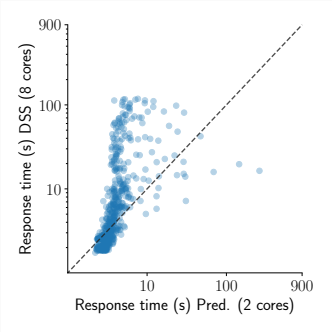


# Evaluation: Setup

Benchmark Setup:

- We implemented DSS in CPAchecker [6].
- We evaluate *DSS* on the subcategory *SoftwareSystems* of the SV-COMP '23 benchmarks.
- We focus on the 2485 safe verification tasks.
- We use the SV-COMP [3] benchmark setup:
  15 GB RAM and an 8 core Intel Xeon E3-1230 v5 with 3.40 GHz.

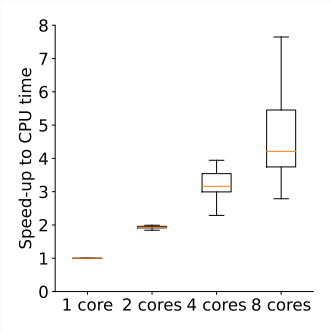
# Evaluation: Results




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




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

The workload is distributed effectively to multiple processing units.

# Evaluation: Outperforming Predicate Analysis

| Task | $\mathrm{CPU}_{\mathbb{P}}(s)$ | $\mathrm{CPU}_{DSS}(s)$ | $\mathrm{RT}_{\mathbb{P}}(s)$ | $\mathbf{RT}_{DSS}(s)$ | # 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 [9, 10] scales well but reports many false alarms.
  ⇒ *DSS* is not restricted to a local context.
- BAM [5] has nested blocks that are not parallelizable.
  ⇒ *DSS* avoids nested blocks and parallelizes as much as possible.
- HIFROG [1] is bound to function summaries and SMT-based algorithms.
  ⇒ *DSS* is domain-independent and can use arbitrary block size.

# Conclusion — Part 1

- *DSS* can decompose a verification task into independent smaller tasks.
- *DSS* is domain-independent and supports various block sizes.
- *DSS* effectively distributes the workload to multiple processing units.
- *DSS* is implemented in CPAchecker [6].



Supplementary webpage

# Part 2

- There are many verifiers for C and Java available (SV-COMP: 76).
- How to make all verifiers from SV-COMP employable in practice?

# Motivation

- **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

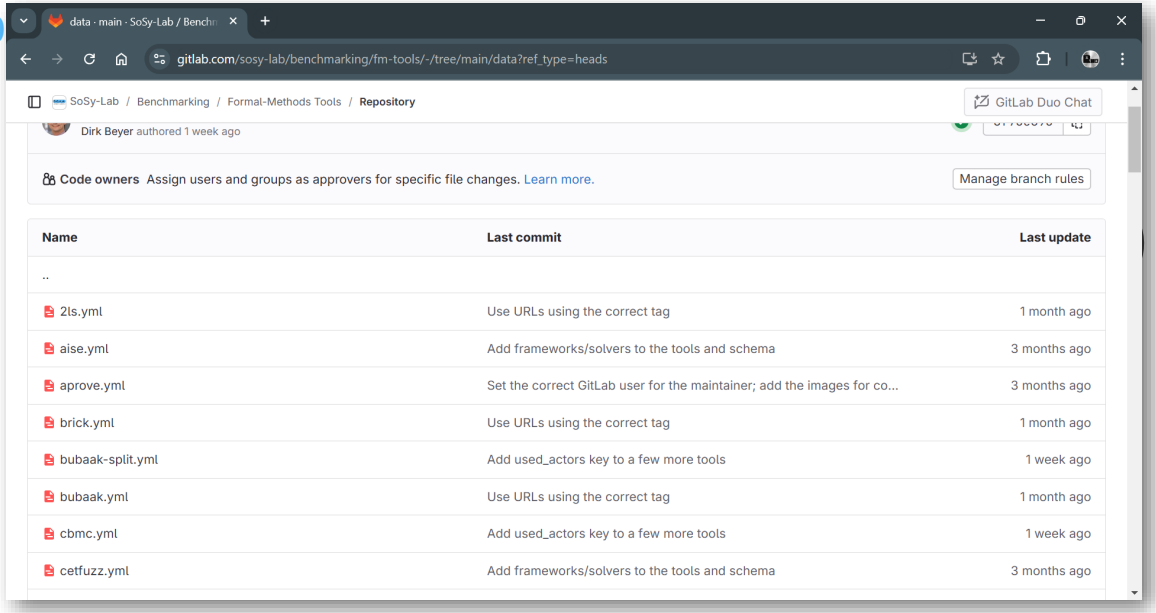
- Support documentation and reuse
- Easy to query and generate knowledge base
- Long-term availability/executability of tools
- Must come with tool support
- Approach must be compatible with competitions

# Solution [2]

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)

Maintained by formal-methods community und tool support by FM-Weck [8]


data - main - SoSy-Lab / Benchm
gitlab.com/sosy-lab/benchmarking/fm-tools/-/tree/main/data?ref_type=heads
SoSy-Lab / Benchmarking / Formal-Methods Tools / Repository
GitLab Duo Chat
Dirk Beyer authored 1 week ago
Code owners Assign users and groups as approvers for specific file changes. Learn more.
Manage branch rules
Name
Last commit
Last update
..
2ls.yml
Use URLs using the correct tag
1 month ago
aise.yml
Add frameworks/solvers to the tools and schema
3 months ago
aprove.yml
Set the correct GitLab user for the maintainer; add the images for co...
3 months ago
brick.yml
Use URLs using the correct tag
1 month ago
bubaak-split.yml
Add used_actors key to a few more tools
1 week ago
bubaak.yml
Use URLs using the correct tag
1 month ago
cbmc.yml
Add used_actors key to a few more tools
1 week ago
cetfuzz.yml
Add frameworks/solvers to the tools and schema
3 months ago

fm-tool YAML

```yaml
name: CPAchecker
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
```

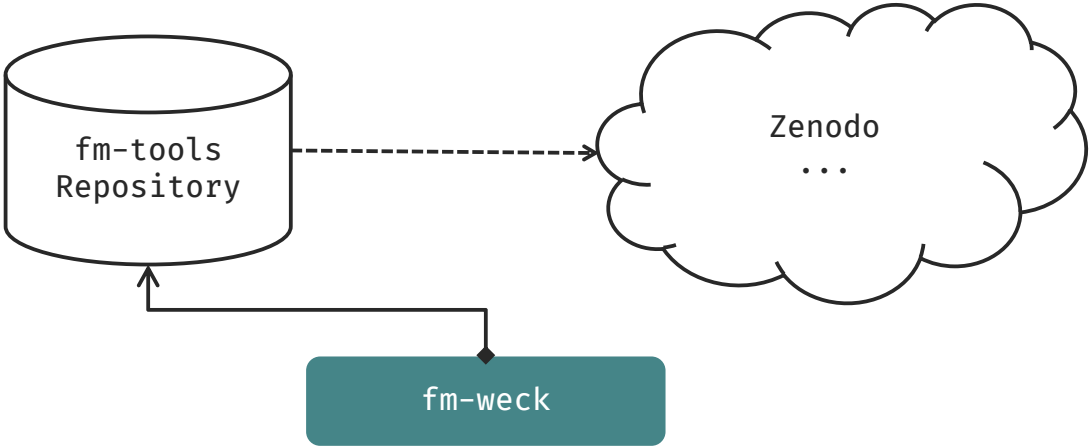
**fm-tool YAML**

```yaml
name: CPAchecker
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
```

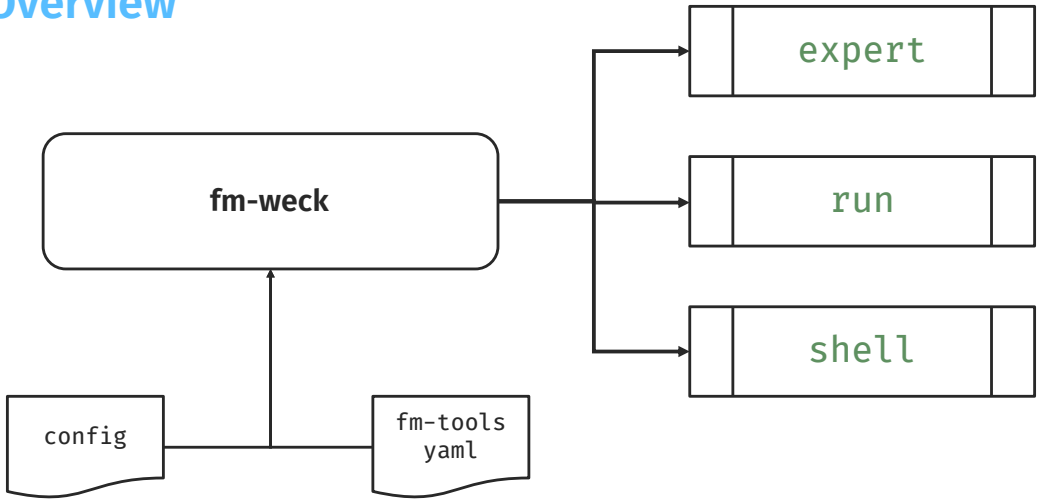
**Anatomy of a version**

```yaml
versions:
 - version: "svcomp24"
   doi: 10.5281/zenodo.10203297
   benchexec_toolinfo_options:
     ["-svcomp24", "-heap", "10000M", "-benchmark",
      "-timelimit", "900 s"]
   contianer_base_image: "ubuntu:22.04"
   container_full_image: "registry.gitlab.com/sosy-
lab/benchmarking/competition-scripts/user:latest"
   required_ubuntu_packages:
     - openjdk-17-jdk-headless
```

# Data Source



# Overview



# Modes

Tool Name & Version

```
fm-weck run cpachecker:2.3.1 \
    -p unreach-call \
    prog.c
```

Common Property by name

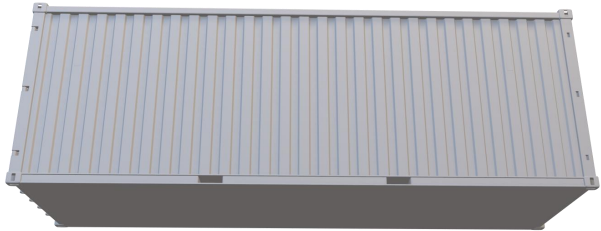
# Modes

Tool Name & Version

```
fm-weck expert cpachecker:2.3.1 \
    --witness witness.yml
    --spec unreach-call.prp
    prog.c
```

Verbatim Arguments to Tool

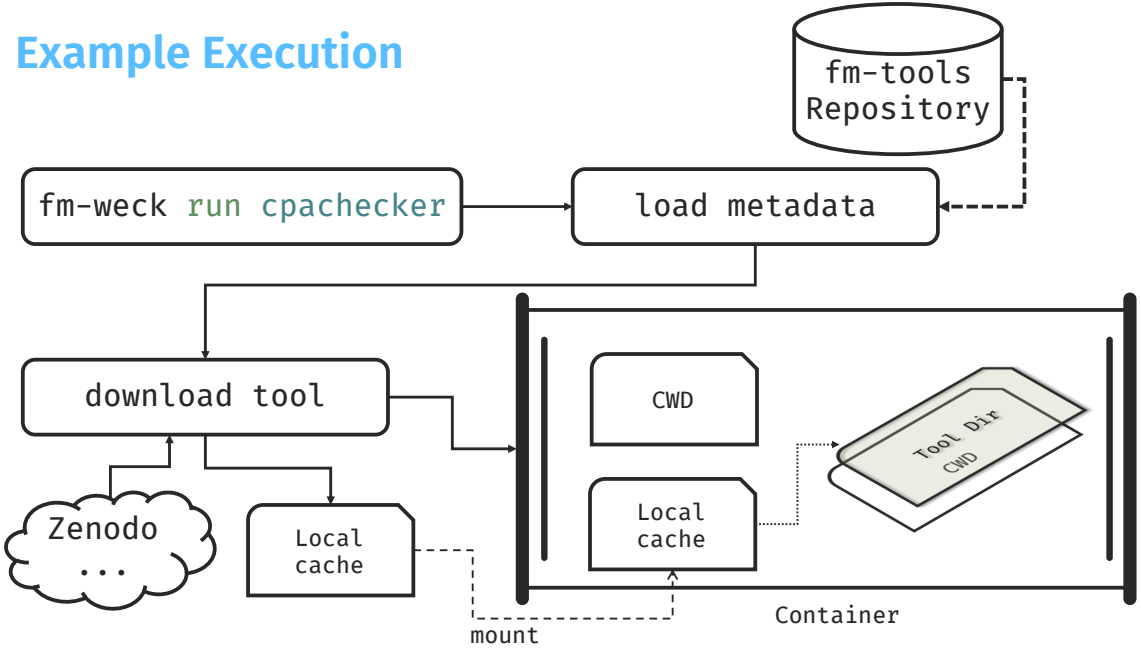
# Modes



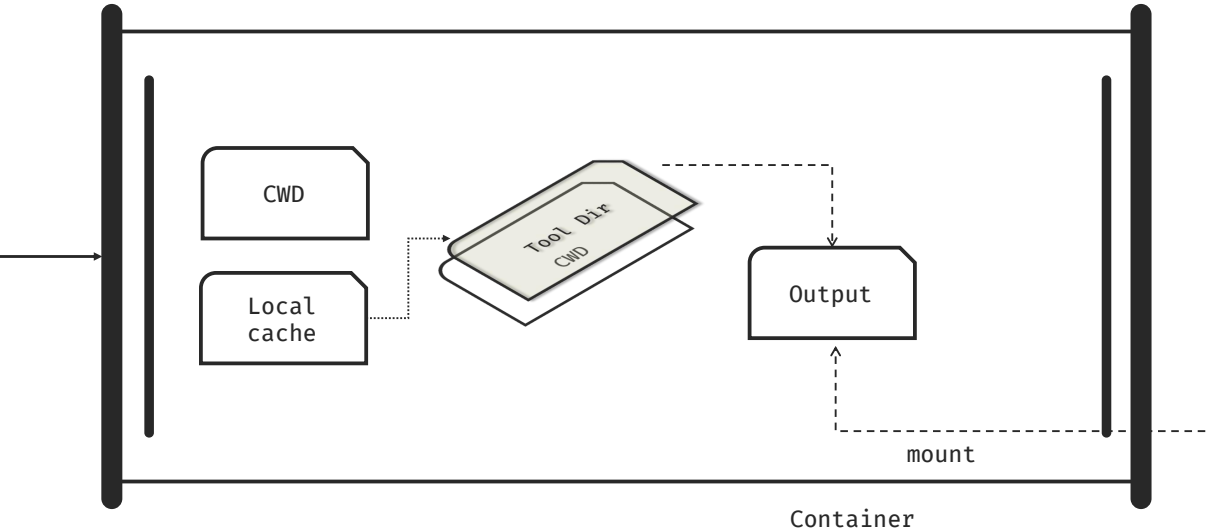
```
fm-weck shell cpachecker:2.3.1
```

Launch interactive shell

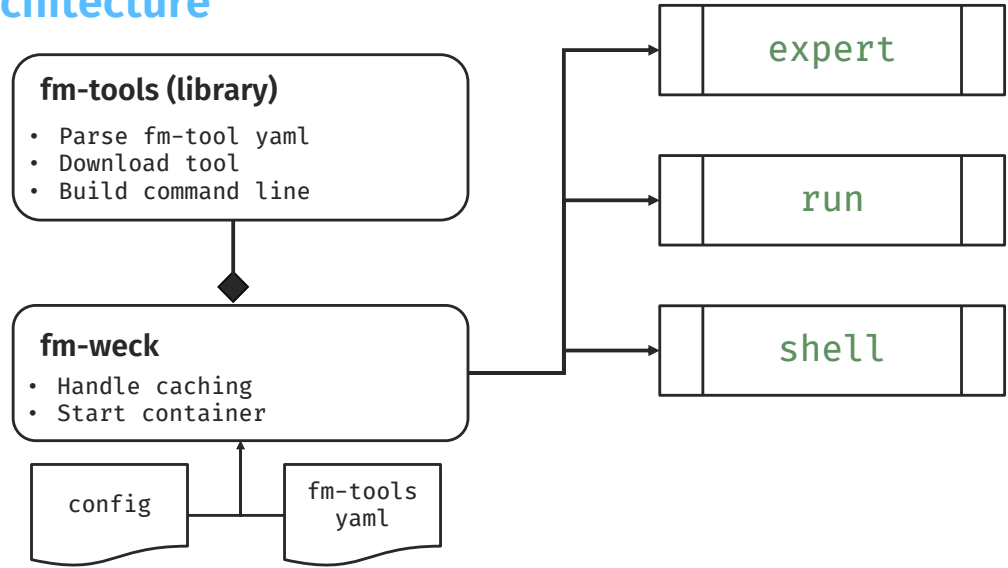
# Example Execution



# Example Execution



# Architecture



on gitlab.com/sosy-lab/software/fm-weck



# fm-weck

`fm-weck expert`

- Download
- Run in Container
- fm-tool cli processed verbatim

`fm-weck run`

- Download
- Run in Container
- fm-tool cli automatically assembled

`fm-weck shell`

- Launch Container
- develop with fm-tools

# FM-Tools and FM-Weck

Dirk Beyer, Henrik Wachowitz:
**FM-Weck: Containerized execution of formal-methods tools**
Proc. FM, 2024. https://doi.org/10.1007/978-3-031-71177-0_3

Dirk Beyer:
**Find, Use, and Conserve Tools for Formal Methods**
To appear, 2024. https://www.sosy-lab.org/research/pub/2024-Podelski65.Find_Use_and_Conserve_Tools_for_Formal_Methods.pdf

# Conclusion

Part 1:

- Distributed Summary Synthesis
- Modular, distributed, analysis-independent
- New verification approach
- Implemented in CPAchecker [6]

Part 2:

- FM-Tools collects and stores essential information [2]
- FM-Weck run a tool in conserved environment [8]
- Generate a knowledge base about formal-methods tools [2]
  https://fm-tools.sosy-lab.org

# References I


[1] 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). https://doi.org/10.1007/978-3-662-54580-5_12

[2] Beyer, D.: Conservation and accessibility of tools for formal methods. In: Proc. Festschrift Podelski 65th Birthday. Springer (2024)

[3] 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). https://doi.org/10.1007/978-3-031-57256-2_15

[4] 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). https://doi.org/10.1109/FMCAD.2009.5351147

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

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

# References II


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

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

[9] 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). https://doi.org/10.1007/978-3-319-17524-9_1

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