FM-Weck: Containerized Execution of Formal-Methods Tools

Dirk Beyer and Henrik Wachowitz

LMU Munich, Munich, Germany

https://gitlab.com/sosy-lab/software/fm-weck

Abstract. Software is ubiquitous in the digital world, and the correct function of software systems is critical for our society, industry, and infrastructure. While testing and static analysis are long-established techniques in software-development processes, it became widely acknowledged only in the past two decades that *formal methods* are required for giving guarantees of functional correctness. Both academia and industry worked hard to develop tools for formal verification of software during the past two decades, with the result that many software verifiers are available now (for example, 59 freely available verifiers for C and Java programs). However, most software verifiers are challenging to find, install, and use for both external researchers and potential users. FM-Weck changes this: It provides a fully automatic, zero-configuration container-based setup and execution for more than 50 software verifiers for C and Java. Both the setup requirements and execution parameters of every supported verifier are provided by the tool developers themselves as part of the FM-Tools metadata format that was established recently and was already used by the international competitions SV-COMP and Test-Comp. With our solution FM-Weck, anyone gets fast and easy access to state-of-the-art formal verifiers, no expertise required, fully reproducible.

Keywords: Formal Methods · Verification · Model Checking · Testing · FM-Tools · Tool Conservation · Reproducibility · Satisfiability Modulo Theories · Provers

1 Introduction

Reliable, correctly functioning IT systems are fundamental in a digital world. One way to achieve correct systems is to apply formal methods. Tools for formal methods are intricate software systems, which often compute abstract models to prove system implementations correct or find errors. There is already a large pool of mature and well-established verification tools (for example, in the area of software verification [1, 2, 3, 4, 5]), and automatic tools are heavily used in industrial software-engineering applications [4, 6, 7, 8]. Sometimes such tools are even used as components in meta verifiers [9, 10, 11, 12, 13, 14]. However, the integration of verification tools provides multiple obstacles: (1) There exists a plentitude of research verification tools that are no longer maintained despite

A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 39–47, 2025.
https://doi.org/10.1007/978-3-031-71177-0_3

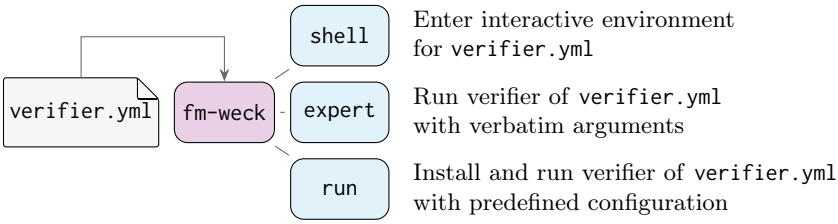


Fig. 1: Overview of FM-Weck

delivering interesting results, making them incapable of running in modern software environments, (2) the tools often provide poor documentation of their requirements on the environment (e.g., whether LLVM 9 or LLVM 12 is required), and which operating system they expect (e.g., Ubuntu 20.04), and (3) these tools often have a huge configuration space resulting in a complicated set of command-line interface (CLI) arguments that users have to understand and set correctly. These obstacles deter developers and researchers from experimenting or even integrating verification tools in their own processes and tools [15, 16].

FM-Weck is a command-line tool that mitigates these issues by using the developer-provided metadata from the FM-Tools repository [17, 18]. The FM-Tools repository is a community-maintained source of metadata for formal-methods tools. The repository and the metadata format has been adopted by the international competitions on software verification (SV-COMP) [19] and testing (Test-Comp) [20], and tool developers maintain the information about their tools, including the expected runtime environments and execution parameters. FM-Weck uses these data provided by experts to give researchers and users easy access to controlled runtime environments and execution of more than currently 50 verification tools for C and Java. Figure 1 gives an overview of FM-Weck's three modes of operation, `shell`, `expert`, and `run`, which assist users working with tools for formal methods. In the following section, we briefly introduce the FM-Tools metadata format that is used by the FM-Tools repository (for more details we refer to the format description [17]), then we present how to use FM-Weck's modes of operation before concluding with current applications of the tool.

Related Work. CoVeriTeam [21] is a tool and language for constructing tool compositions. It uses a YAML-based format for the atomic-actor definitions (information where to download, how to assemble command-lines). This format has inspired the format used in FM-Tools. Unfortunately, CoVeriTeam does not configure the execution environment for the tools and simply assumes that the host machine has all required packages readily installed, which FM-Weck solves. Conserving tools for formal methods is an old desire [22], also addressed by CoVeriTeam Service [9]. FM-Weck adds the use of Docker containers to make the environment reproducible and easy to run, also independently from web services.

2 FM-Tools: Tool Metadata

The FM-Tools repository aggregates relevant information about tools for formal methods: It specifies the download location, maintainers, command-line options,

as well as other related information. In addition, FM-Tools stores information about container images on which the tool is guaranteed to run according to the maintainers. An FM-Tools file for a specific tool is a YAML document with a precisely defined set of keys (a schema for the metadata of formal-methods tools is available in the repository). FM-Tools was adopted by SV-COMP and Test-Comp in their 2024 edition [19, 20]. As part of FM-Weck, we also provide a Python library [23] that helps users to parse, use, and modify FM-Tools.

```yaml
versions:
  - version: "svcomp24"
    doi: 10.5281/zenodo.10203297
    benchexec_toolinfo_options:
      ["-svcomp24", "-heap", "10000M",
       "-benchmark", "-timelimit", "900 s"]
    required_ubuntu_packages:
      - openjdk-17-jdk-headless
    base_container_images:
      - docker.io/ubuntu:22.04
    full_container_images:
      - registry.gitlab.com/sosy-lab/\
        benchmarking/competition-scripts/user:2024
```

Listing 1: Example of a tool entry in FM-Tools

Listing 1 shows an example of a tool-version entry in FM-Tools. The field `required_ubuntu_packages` specifies the Ubuntu packages that are required to run the tool. The field `base_container_images` specifies the Ubuntu container images on which the required Ubuntu packages can be installed, and with which the tool is guaranteed to run after package installation. The field `full_container_images` specifies self-contained container images that are guaranteed to run the tool out-of-the-box. For a tool t it shall hold that

$$\forall i \in \texttt{base\_container\_images} : (i \oplus \texttt{required\_ubuntu\_packages}) \models t \quad (1)$$

$$\forall i \in \texttt{full\_container\_images} : i \models t \quad (2)$$

where $\oplus$ denotes the operation of installing the packages on the image i and $i \models t$ denotes that the image i is sufficient to run the tool t.

FM-Tools currently refers to Ubuntu packages, because most tools run on Linux, and Ubuntu as a widespread distribution, whose long-term support keeps specifying and installing the packages straightforward across verifiers.

3 FM-Weck

FM-Weck is a command-line tool written in Python, which consumes FM-Tools [18] tool metadata to execute formal-methods tools inside of a container. (The tool's name is inspired by a German brand of jars for conserving food.) The utility can be used to run, develop, and experiment with formal-methods tools. The software architecture of FM-Weck also allows and encourages usage as a library.

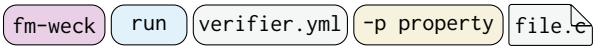
FM-Weck simplifies the execution of formal-methods tools by setting up and starting containers tailored for each tool. FM-Weck can also configure a

container runtime such that benchmarks with BenchExec are possible inside of them. To launch the actual container, FM-Weck uses `podman` [24] internally with the `crun` runtime [25]. The FM-Weck CLI comes with three modes of operation: `run`, `expert`, and `shell`.

3.1 FM-Weck Modes

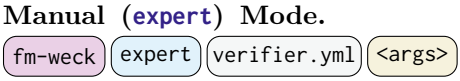
Every command in FM-Weck takes an FM-Tools file as input. This file can be specified either as a path, or as the identifier of the tool. In the latter case, FM-Weck uses the bundled file from the FM-Tools repository with the corresponding name. In any case, users can also specify a specific version of a verifier by appending it with a colon after the file path or name, e.g., `<verifier>:<version>`.

Automatic (`run`) Mode.



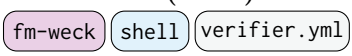
The `run` mode enables plug-and-play execution of formal-methods tools: it downloads and unpacks a tool from the archive specified in the FM-Tools metadata file ('verifier.yml' above) into a user-specified cache directory on the host system. This cache is mounted into the container, where the verifier is then executed with the given command-line arguments. The `run` mode takes two additional arguments: (1) the `-p` argument specifies a property file, i.e., the goal for the verifier—this can either be a path to the property file or the name of one of the properties used in SV-COMP or Test-Comp, and (2) the files that shall be passed to the tool. In the case of software verifiers, these program files are the input programs to be verified.

Manual (`expert`) Mode.



The `expert` mode is for manual interaction with a verifier: it executes a given verifier, specified through the corresponding FM-Tools YAML file, in its containerized environment passing any additional arguments verbatim to the verifier. Just like in the `run` mode, FM-Weck takes care of downloading and unpacking the verifier as well as setting up the container before the execution. All arguments following the tool `verifier.yml` are passed to the verifier in the container, which makes the `expert` mode essentially act like the verifier if it was executed directly on the host system. The following is an example execution that displays the version of the CPAchecker verifier: `fm-weck expert cpachecker -version`

Interactive (`shell`) Mode.



The `shell` mode enters an interactive shell inside of the container specified by the given verifier. The shell mode launches a Bash shell with the current working directory mounted inside. Users may mount additional directories through a configuration file described in Sect. 3.2. Like with the `expert` mode, the container information is extracted from the FM-Tools metadata file provided by the user. The shell mode takes no additional parameters. The following example starts an interactive shell in the container of Ultimate Automizer: `fm-weck shell uautomizer`

3.2 Project-Specific Configuration

FM-Weck works without any additional configuration, but expert users can still modify aspects of FM-Weck to their needs. Users may set default values and additional files or directories which shall be available inside the container. The configuration is specified in TomL format, as seen in Listing 2. If users define a default image file in this configuration, they can omit the `verifier.yml` in the `shell` mode, and the `*container_image` keys in the `expert` and `run` modes.

```
[defaults]
image = "some_image:latest"
[mount]
"local/path" = "/container/path"
```

Listing 2: Example of a run configuration

Relative paths in the configuration file are relative to the directory that contains the configuration file. If no configuration path is explicitly set via the command line, FM-Weck first looks for a configuration file `.weck` in the current working directory. If this does not exist, it looks for a configuration file `.config/weck` in the user's home directory.

4 Applications

FM-Weck is designed with three core applications in mind: (1) to execute a single tool based on its FM-Tools metadata, (2) to facilitate the execution of unmaintained tools in future competition instances, and (3) as a utility that enables OS-independent execution in CoVeriTeam [21].

4.1 Execution of a Single Tool

FM-Weck provides a bother-free user experience that encourages curious researchers and developers to try and experiment with different verification tools—from well established behemoths to cutting-edge research tools. Users do not have to worry about the tool's dependencies, installation, or complicated command-line configurations. The `run` mode of FM-Weck achieves this goal. Running CPAchecker to find overflows in a C program is as simple as:

```
fm-weck run cpachecker -p no-overflow program.c
```

4.2 Containerized Execution in CoVeriTeam

CoVeriTeam [21] is a framework for cooperative verification. Similar to `fm-weck`, CoVeriTeam takes tool metadata in a YAML format as input, to download and run the tools specified in a cooperative-verification workflow. Each tool is executed inside a containerized environment provided by BenchExec [26]. However, these BenchExec containers do not support OCI container images. This means that all tools running in a CoVeriTeam workflow must be able to run on the host system. We extend CoVeriTeam with an FM-Weck-based run mode. This enables the cooperation of actors regardless of their system requirements.

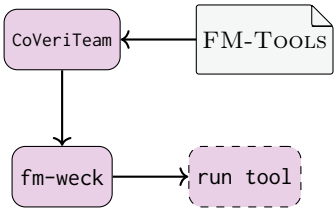


Fig. 2: FM-Weck as executor in CoVeriTeam

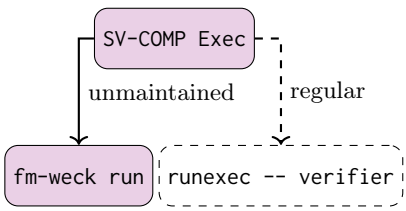


Fig. 3: FM-Weck as drop-in command for SV-COMP infrastructure

Figure 2 illustrates the integration of FM-Weck in CoVeriTeam. Instead of calling BenchExec, CoVeriTeam calls FM-Weck to instantiate a container for the given tool and execute the assembled command inside of it. CoVeriTeam is also written in Python and uses FM-Weck directly as a library.

4.3 Reliable Execution in SV-COMP 2025

SV-COMP comparatively evaluates more than 70 verification tools on an extensive benchmark set [19, 27]. The server infrastructure that executes these millions of verification and validation runs during the competition is hosted on the always-latest Ubuntu LTS Version. This is a formal requirement of the SV-COMP rules. However, there is a growing number of tools that are no longer actively maintained and serve as a retrospective baseline—the so-called hors-concours participants. These tools are benchmarked in the same way as the regular participants, but they do not compete in the ranking. Until SV-COMP 2024, these tools were manually migrated by volunteers to still work on the latest Ubuntu LTS Version, but with 26 hors-concurs participants, the amount of migration-labor becomes infeasible.

With FM-Weck we extend the functionality of the current SV-COMP infrastructure to execute these tools in the SV-COMP 2024 environment. Figure 3 illustrates how FM-Weck is used as a drop-in solution. We wrap the existing invocation of the benchmark command inside of a pre-built image. This image replicates the OS and installed packages of SV-COMP 2024. By default, BenchExec cannot run inside of another container: FM-Weck also sets up the container runtime such that BenchExec works inside of it.

5 Conclusion

We developed FM-Weck, a utility to run formal-methods tools in containerized environments. The goals are to (a) conserve the tools, such that they stay executable in the future, and (b) make it easy for researchers, practitioners, and educators to use and explore the existing tools for formal methods. The application scenarios in CoVeriTeam and SV-COMP infrastructure demonstrate the capabilities of FM-Weck as a library as well as a command-line tool. The tool is open source, licensed under Apache 2.0, and available on GitLab [28].

Data-Availability Statement. The metadata are available in the FM-Tools repository [18] and the source code in the FM-Weck repository [28]. A refined version [29] of the artifact submitted for evaluation [30] is available on Zenodo.

Funding Statement. FM-Weck was funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY).

References

1. 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
2. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate automizer and the abstraction of bitwise operations (competition contribution). In: Proc. TACAS (3). pp. 418–423. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
3. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Trtík, M., Zaoral, L., Ayaziová, P., Strejček, J.: Symbiotic 10: Lazy memory initialization and compact symbolic execution (competition contribution). In: Proc. TACAS (3). pp. 406–411. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_29
4. 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
5. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: The Goblint approach. In: Proc. ASE. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337
6. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Proc. IFM. pp. 1–20. LNCS 2999, Springer (2004). https://doi.org/10.1007/978-3-540-24756-2_1
7. Cook, B.: Formal reasoning about the security of Amazon web services. In: Proc. CAV (2). pp. 38–47. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
8. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. PLDI. pp. 196–207. ACM (2003). https://doi.org/10.1145/781131.781153
9. Beyer, D., Kanav, S., Wachowitz, H.: CoVeriTeam Service: Verification as a service. In: Proc. ICSE, companion. pp. 21–25. IEEE (2023). https://doi.org/10.1109/ICSE-Companion58688.2023.00017
10. Beyer, D., Lemberger, T., Wachowitz, H.: Reproduction package for TACAS 2024 submission 'Continuous verification: Mitigations of tool restarts for java verifiers'. Zenodo (2023). https://doi.org/10.5281/zenodo.8383787
11. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier (competition contribution). In: Proc. TACAS (3). pp. 365–370. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22
12. Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
13. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
14. He, F., Sun, Z., Fan, H.: Deagle: An SMT-based verifier for multi-threaded programs (competition contribution). In: Proc. TACAS (2). pp. 424–428. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_25

15. Alglave, J., Donaldson, A.F., Kröning, D., Tautschnig, M.: Making software verification tools really work. In: Proc. ATVA. pp. 28–42. LNCS 6996, Springer (2011). https://doi.org/10.1007/978-3-642-24372-1_3
16. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: Proc. FMICS. pp. 3–69. LNCS 12327, Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1
17. Beyer, D.: Conservation and accessibility of tools for formal methods. In: Proc. Festschrift Podelski 65th Birthday. Springer (2024), https://www.sosy-lab.org/research/pub/2024-Podelski65.Conservation_and_Accessibility_of_Tools_for_Formal_Methods.pdf
18. Beyer, D.: Formal-methods tools repository. https://gitlab.com/sosy-lab/benchmarking/fm-tools (2023), accessed: 2024-04-10
19. 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
20. Beyer, D.: Automatic testing of C programs: Test-Comp 2024. In: TBA. Springer (2024)
21. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
22. Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: Concepts and design. STTT **1**(1-2), 9–30 (1997). https://doi.org/10.1007/s100090050003
23. Beyer, D., Wachowitz, H.: `lib-fm-tools` repository. https://gitlab.com/sosy-lab/software/lib-fm-tools (2024), accessed: 2024-07-01
24. Podman. https://github.com/containers/podman, accessed: 2023-02-09
25. `crun` runtime. https://github.com/containers/crun (2024), accessed: 2024-04-26
26. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
27. Collection of verification tasks. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks, accessed: 2023-04-01
28. Beyer, D., Wachowitz, H.: FM-Weck repository. https://gitlab.com/sosy-lab/software/fm-weck (2024), accessed: 2024-07-01
29. Beyer, D., Wachowitz, H.: Reproduction package for the FM 2024 article 'FM-Weck: Containerized execution of formal-methods tools'. Zenodo (2024). https://doi.org/10.5281/zenodo.12606323
30. Beyer, D., Wachowitz, H.: Reproduction package for the FM 2024 submission 'FM-Weck: Containerized execution of formal-methods tools'. Zenodo (2024). https://doi.org/10.5281/zenodo.12205513