

FM Meets AI: Equipping AI Agents with Formal-Methods Tools

Dirk Beyer, Thomas Lemberger, and Henrik Wachowitz

LMU Munich, Munich, Germany

Abstract. AI agents take an increasing role in software development, speeding up code generation and lowering the barrier to advanced technologies. At the same time, AI-generated code can be subtly wrong, violating safety properties or missing edge cases. Formal-methods tools offer rigorous correctness checks, but they have remained out of reach for AI agents: setting them up and invoking them correctly required expertise in formal methods. To close this gap, we introduce agent skills that provide AI agents such as Claude Code with the ability to list available formal-methods tools, automatically select the best-ranked tool for a given task, invoke it in an isolated container environment, and interpret the result—all without manual setup. The skills expose structured tool data from the `FM-TOOLS CATALOG` and competition results from `SV-COMP` and `Test-Comp`, giving AI agents direct access to over 60 verifiers and 20 test generators and the results of their comparative evaluation.

Keywords: Formal Methods · `FM-TOOLS CATALOG` · AI Agents · FAIR · Software Tools · Software Verification · Software Testing · Competitions

1 Introduction

AI agents have become an indispensable part of modern software development. They draft, refactor, and debug code at a speed that was unthinkable a few years ago, and they let developers use libraries and languages without extensive onboarding. This shift presents both significant opportunities and notable challenges. On the one hand, AI agents generate code that compiles, runs, and passes tests—but the code may still be subtly wrong: it may violate safety properties, miss corner cases, or rely on implicit assumptions that quietly break in production. On the other hand, the same agents have the potential to invoke powerful formal-methods tools such as verifiers that were previously restricted by a high barrier to entry, as expert knowledge was required to employ them [1, 2]. The formal-methods tools then provide rigorous answers about whether a system meets a specification or not.

In practice, this rarely happens. Formal-methods tools have been out of reach for AI agents and non-expert developers alike: each tool requires a specific runtime environment (e.g., a specific Linux distribution, JDK version, or LLVM version), each tool has its own extensive command-line interface, and each tool expects specifications in a particular format. Setting up a single verifier already requires

substantial expertise in formal methods; on top of this, choosing the right verifier for a given task is a research question of its own.

The community surrounding formal methods has invested considerable effort into lowering these barriers. The FM-TOOLS CATALOG [3] collects machine-readable metadata about formal-methods tools—descriptions, download locations, required packages, container images, and command-line options—and is already used by SV-COMP [4, 5, 6] and Test-Comp [7, 8, 9]. FM-WECK [10] uses these metadata to run formal-methods tools in containers without manual setup. Together, they make formal-methods tools *installable* and *executable* in a uniform way, but they do not yet make them *discoverable* and *usable* for an AI coding agent that has no prior knowledge of formal methods.

Contribution. We contribute a set of new *agent skills* that closes this gap. The skills let AI agents such as Claude Code list the available formal-methods tools, select the best-ranked tool for the task, invoke it on a given input program, and interpret the result—all in an isolated container environment and without manual setup. The skills build on data supplied by the FM-TOOLS CATALOG and on the ability to run the tools provided by FM-WECK. Together, they give an agent direct access to over 60 verifiers and 20 test generators maintained by the community. In sum, we contribute the following:

- We make structured, authoritative data on formal-methods tools accessible to AI agents via dedicated agent skill and `llms.txt` files.
- We make structured competition results from SV-COMP and Test-Comp accessible to AI agents, enabling data-driven tool selection without web scraping.
- We enable AI agents to automatically select and execute a tool that is best-ranked for the category of the given verification or test-generation task, containerized via FM-WECK and without manual setup.

Providing structured data directly reduces token consumption compared to agents scraping and parsing web pages, potentially saving cost and energy.

Related Work. The FM-TOOLS CATALOG and its accompanying command-line interface FM-WECK received significant attention due to their involvement in two major community competitions, SV-COMP and Test-Comp, but the desire to have such solutions is not new and has been reported by others already.

Components. The desire to treat verification tools and solvers as components has existed for decades. Formal-methods tools were considered as components before (e.g. [11, 12, 13, 14, 15]) and there were efforts to provide verification tools as a service [16, 17, 18, 19, 20]. Some specific tool projects offer web services to use their tools without installing them (e.g., CPAchecker [21] and Ultimate [22]).

Catalogs. There are also other attempts to collect purely the information about formal-methods tools, such as YAHODA [23, 24], PROVERB [25, 26], the CADP Zoo [27], and the EUROPROOFNET ATP WIKI [28]. The FM-TOOLS CATALOG additionally makes the tool versions accessible and provides recommended command-line options, and integrates with FM-WECK and some competitions.

Execution and Conservation. Container engines such as `PODMAN` or `DOCKER` expose a well-documented interface to create container images (blueprints for isolated environments) and run containers based on them. `BENCHEXEC` [29] is a tool for performing reproducible performance experiments. It provides sophisticated measurements for tools and a well-documented Python interface to integrate new tools into its benchmarking process.

Interaction with AI Agents. The closest related work is the `ESBMC` agent marketplace (<https://github.com/esbmc/agent-marketplace>), a Claude Code plugin that integrates the `ESBMC` verifier for source-code verification and security audits. The `ESBMC` marketplace focuses on a single verifier; our contribution, in contrast, exposes the entire `FM-TOOLS CATALOG` and supports tool selection based on the input program and property.

Mistral AI’s `Leanstral` (<https://mistral.ai/news/leanstral>) is an open-source code agent specialized for the interactive theorem prover `LEAN 4`. It generates formal proofs and definitions, and it connects to the agent through an MCP integration with the Lean language server.

2 Background

The ideas described in this paper are based on three core concepts: (1) treat formal-methods tools as off-the-shelf components (in binary format), (2) conserve information about formal-methods tools and their usage in a central catalog (`FM-TOOLS CATALOG`), and (3) execute formal-methods tools in an automated, containerized environment (`FM-WECK`). We use existing conventions and technology to provide knowledge about these concepts to AI agents and enable them to execute the tools.

Formal-Methods Tools as Off-the-Shelf Components. The idea of using solvers and verifiers as off-the-shelf components to build new tools is not new in the formal-methods community [13, 15, 30]. Automatic software verifiers use SMT solvers to solve complex boolean formulas (e.g., [21, 31]) and fuzzing-based test-case generators may incorporate informed symbolic solvers to guide their fuzzing seeds (e.g., [32]). As a natural extension, the community also wanted a way to use more complex formal-methods tools as off-the-shelf components. There were several attempts at this [15, 16, 17, 18, 19, 20]; `COVERITEAM` stands out because it uses a `YAML`-based tool-description format, which was later adapted and extended into the `FM-TOOLS` format used by the `FM-TOOLS CATALOG`.

FM-TOOLS CATALOG. The `FM-TOOLS CATALOG` [3] aggregates resources on tools prevalent in the community. At the heart of the catalog are the `FM-TOOLS` data files: `YAML` files containing information on the project and maintainers of a tool, as well as parameters necessary to run a given version of the tool. For each version of a tool, the `FM-TOOLS` data files also contains the DOI of the archive where the tool can be downloaded and information on the expected runtime environment.

FM-WECK. `FM-WECK` is a command-line tool that offers a unified way to interact with tools that are described in the `FM-TOOLS CATALOG`. Many features of `FM-WECK` are inspired by package managers such as Python’s `pipx` or

```
name: fm-tools-data
description: "Catalog lookup for metadata about formal methods tools
↳ listed at fm-tools.sosy-lab.org: maintainers, input languages,
↳ supported properties, container images, literature, and competition
↳ participation history. Trigger when the user asks WHAT a named
↳ verifier, theorem prover, or tester is, WHO maintains it, WHAT it
↳ supports, HOW it works, or WHERE to find it - for any tool such as
↳ CPAchecker, ESBMC, FuSeBMC, GDart, Goblint, JBMC, KeY, KLEE, Theta, or
↳ Ultimate Automizer. Also trigger when the user wants to LIST or
↳ COMPARE tools by technique, framework, or input language (fetch the
↳ grouped catalog page). Do NOT trigger for: quantitative rankings or
↳ scores (use competition-data), running a verifier on a program (use
↳ run-fm-weck), or general formal methods concepts that do not require
↳ catalog data."
```

Listing 1: Front matter of the skill `FM-TOOLS-DATA`, defining the name and the description that tells the agent when to load the skill

Node’s `npm`: downloading and installing a tool, listing tools and their versions, and running tools with a unified command line. A key feature of `FM-WECK` is its default mode of containerized execution: the runtime information in the `FM-TOOLS` data files tells `FM-WECK` where to obtain or how to construct a suitable container image to execute the tool.

AI Agents and Tooling Infrastructure. An *AI agent* (e.g., Claude Code) is a large language model in a feedback loop with access to tooling for reading and editing files, fetching web content, and running commands. There are three established mechanisms to connect such agents to external tools and information: (1) An *agent skill* is a Markdown instruction file (with a short YAML front matter) that the agent loads when its description matches the current task; the skill explains *when* and *how* to use a set of tools, and runs the tools inside the agent’s existing execution environment. (2) An *MCP server* ([Model Context Protocol](#)) exposes external services to an AI agent through a typed JSON-RPC interface. The MCP server is a server that must be installed on the same machine as the agent. It can be started by the agent on-demand or runs continuously in the background. It exposes a number of methods that the agent can call. Each method is a programmatic function that, when called by the agent, is executed. The return value of the function is communicated to the agent. (3) A text file [llms.txt](#) at a website’s root gives an LLM a curated overview of the site.

3 Agent Infrastructure for Formal-Methods Tools

Access to Data on Formal-Methods Tools. To give agents access to the information of formal-methods tools maintained in the `FM-TOOLS CATALOG`, we provide a new agent skill, `FM-TOOLS-DATA`. We choose the skill workflow over an

MCP server because skills are more lightweight, easier to provide, extend, and maintain, and easier to install for users. We design the description of the skill such that it is likely to trigger whenever the user asks about information about a specific formal-methods tool. [Listing 1](#) shows this description.

Once the skill is loaded, it provides the agent with step-by-step instructions to retrieving information on formal-methods tools from the FM-TOOLS CATALOG. First, the agent must fetch the `llms.txt` file from the FM-TOOLS website. We added this file to the website to direct AI agents to tool-specific information. The file lists all formal-methods tools that are available in the FM-TOOLS CATALOG, links to each tool’s description file, and provides links to overview pages for finding tools by used framework or capabilities. From this file, the agent can determine the location of the tool’s description file (in YAML format) and fetch it. If the user asks for a specific information about a tool, the agent is instructed to only extract and present this information. If the user asks for a general overview of a tool, the agent is told a specific structure in which to present the information. In case the user asks not for information on a specific tool, but for a number of tools that fulfill a certain criterion (e.g., tools that support a certain programming language or technique), the agent is told to use the overview pages linked in the `llms.txt` file to find the tools that fulfill the criterion. The FM-TOOLS CATALOG is open to contributions: new tools can be added at any time, and it follows the FAIR principles.

Access to Competition Data. Now the agent knows how to retrieve information on a given formal-methods tool, but it does not know how to select a tool for a given task if the user does not provide a specific tool name. For this, we aim to make the results of tool competitions available to AI agents. In this paper, we focus on the quantitative data that are available from SV-COMP and Test-Comp.

By default, if an agent is asked about the results of one of the competitions, it would search for the website, download the page as HTML, and read the included HTML table with the overall results. First, we help agents that access the official competition websites to read the main results more efficiently, by providing (a) CSV files as an alternative to the HTML tables, and (b) a single summary markdown file that explains the category naming and lists, for each category, the top three participants by score. To help agents find these files, we provide an `llms.txt` file that points to them. The `llms.txt` also points to the respective competition’s rules and task definitions in case the agent requires more detailed information.

Second, we provide the skill `COMPETITION-DATA`. The skill currently supports SV-COMP and Test-Comp and we plan to extend it to SAT-COMP [33] in the near future. The skill tells agents about the competition websites and how to select relevant categories and tools for a given task. Together, the auxiliary files on the competition web pages and the skills turn the agent into a recommender system that can use the latest competition data to make informed choices on which tool works best for a given task.

For example, when the agent has to decide which verifier to run, it can match the user’s input program to the closest competition category, and then use the summary markdown file to get the top three participants on that category.

```

name: run-fm-weck
description: "Use this skill when the user explicitly mentions fm-weck,
↳ asks to run a named formal software verifier (CPAchecker, Ultimate
↳ Automizer, ESBMC, CBMC, or similar) on a program, asks which tools or
↳ versions fm-weck supports, or asks to verify a program without naming
↳ a specific tool. In the last case, use the competition-data skill to
↳ map the program's property to a competition and category and select
↳ the best-ranked tool before proceeding. Do NOT trigger for general
↳ static analysis (linters, sanitizers, Valgrind) or formal methods
↳ discussion unrelated to running a tool."

```

Listing 2: Front matter of the skill RUN-FM-WECK

Using Formal-Methods Tools. The new skill RUN-FM-WECK instructs agents how to call FM-WECK to run a formal-methods tool that is available in the FM-TOOLS CATALOG. We designed the description of the skill such that it is likely to trigger when one of the following conditions is met: (1) the user mentions FM-WECK directly, (2) the user asks to verify a property of a C program, (3) the user asks to list available formal-methods tools, or (4) the user asks to verify a C program without naming a specific tool. Listing 2 shows this description.

The skill is structured into a handful of short sections, each answering one question that the agent has to address before running a formal-methods tool. An *Overview* states what FM-WECK does on a high level, and the *Prerequisites* list the system requirements for using FM-WECK.

The *Installation* section instructs the agent what to do if FM-WECK is not yet available on the user’s system. We describe the installation as a decision tree that picks the first method that works. What we want to avoid is that the agent installs the Python tool into the user’s global Python packages, so the decision tree guides the agent first to try `pipx`, which creates an isolated install of FM-WECK, before falling back to creating virtual Python environments on its own. *Basic Usage* explains how to run a formal-methods tool with FM-WECK and shows one concrete example. After that, two short sections explain how to use FM-WECK for listing available tools and specifications. If the user asks to verify or test a program without naming a specific tool, the skill instructs the agent to use COMPETITION-DATA to find a fitting tool based on competition data.

Coding Conventions for Formal-Methods Tools. The new skill FM-TOOLS-CONVENTIONS equips agents with the expected inputs supported by formal-methods tools. The skill focuses on the code conventions established by the SV-COMP and Test-Comp communities, because these conventions are not as clear for AI agents as, for example, DIMACS CNF for SAT solvers or JML for Java deductive verifiers. The skill describes how to encode non-deterministic input in C and Java programs, how to encode assumptions, and how to encode assertions.

Setup of Infrastructure. The skills follow the [agent skills convention](#) that is supported by all major AI agents. They are maintained at the website <https://>

```

> Verify the memory-safety of my list program.
* I'll use the run-fm-weck skill to verify memory-safety of list.i with a formal
  verifier.
* Skill(run-fm-weck)
  \ Successfully loaded skill
* I'll first check if fm-weck is installed, then use the competition-data skill
  to pick the best tool for valid-memsafety.
* Bash(fm-weck --version 2>/dev/null || echo "NOT_FOUND")
  \ fm-weck 1.7.2.dev0
* fm-weck is available. Now I'll consult competition-data to find the top-ranked
  tool for valid-memsafety on heap-heavy C programs.
* Skill(competition-data)
  \ Successfully loaded skill
* Fetch(https://sv-comp.sosy-lab.org/llms.txt)
  \ Received 3.5KB (200 OK)
* Fetch(https://sv-comp.sosy-lab.org/2026/results/results-verified/summary.md)
  \ Received 6.9KB (200 OK)
* The most fitting category for a doubly-linked circular list is
  C.valid-memsafety.LinkedList, where PredatorHP ranked 1st in SV-COMP 2026
  with a score of 220. Now let me fetch the full ranking to have fallback
  options, then cross-check with fm-weck --list.
[...]
```

Listing 3: Excerpt of the agent session for the memory-safety use case, showing the agent loading the skills `RUN-FM-WECK` and `COMPETITION-DATA`

gitlab.com/sosy-lab/benchmarking/fm-tools/-/tree/main/skills and can be installed via the following command:

```
npx skills install https://gitlab.com/sosy-lab/benchmarking/fm-tools
```

See www.skills.sh for more information on the Skills tool. The `llms.txt` and files linked from them are provided via the official SV-COMP and Test-Comp websites and should be automatically used by agents that access these sites.

4 Use Cases

We illustrate the enhanced capabilities that AI agents receive through our new infrastructure with four exemplary use cases: (1) getting information on a specific tool, (2) checking a C program for memory safety, (3) checking auto-generated code, and (4) acting as a research assistant to find literature on test generation for C programs. We exercise the use cases first without and then with the new skills to demonstrate the added value of our infrastructure. For our examples, we use Claude Code as AI agent, with model Sonnet 4.6. The full conversations are available on Zenodo [34] and at <https://www.sosy-lab.org/research/fm-tools>.

Getting Information about a Specific Tool. The user wants to know who maintains the tool `CPACHECKER` and what the latest version is. For this, the

user asks the agent about this information, without naming a specific source of information. With no skills provided, the agent performs several web searches; after fetching data from the CPACHECKER repository, it provides the correct information. With the skills available, the agent loads FM-TOOLS-DATA and retrieves both the latest tool version and both maintainers from the FM-TOOLS CATALOG.

As both, the attempt without the skill and the attempt with the skill are able to provide the correct information, this is a good use case to compare the cost of the two approaches. In our example, the attempt without skill requires about 40 k tokens of input (from the initial prompt and web searches) and 1.2 k tokens of output (total cost of \$0.18). The attempt with skill is not only more reliable because it does not require a web search, but also significantly cheaper, requiring about 9 k tokens of input (from the initial prompt, the skill, and fetching data from FM-TOOLS CATALOG) and 1.1 k tokens of output (total cost of \$0.093).

Checking a C Program for Memory Safety. The user wants to check whether an existing C program is memory safe, i.e., has no invalid dereferences, no invalid frees, and no memory leaks.

The agent without the skills available first checks whether its system has any verifiers installed. After the search returns no results, it falls back to performing a series of compiler built-in checks (e.g., AddressSanitizer). After these checks complete successfully, the agent concludes correctly that the program is memory safe.

The agent with the skill (excerpt in Listing 3) identifies the linked-list structure in the program, classifies the task as a verification problem, and loads the skill RUN-FM-WECK. It then loads the COMPETITION-DATA, fetches the latest SV-COMP results summary (provided through the `llms.txt` file) and finds the most suitable verifiers for memory safety on C programs that contain linked lists by looking at category `C.valid-memsafety.LinkedLists`. It runs the top-ranked tool in that category, PREDATORHP [35], on the program, and reports also correctly that the program is memory safe, but with the confidence of the best available formal-verification tool.

Checking Auto-Generated Code. A user generates an algorithm in Java with the help of an AI agent. The user wants to strengthen the trust in the generated code by checking it with a formal-methods tool: they task the agent with formulating suitable pre- and postconditions for the generated method, and then checking these conditions with a verifier.

Without skills, the agent starts by writing a Java program and a separate Dafny program to confirm the envisioned algorithm is correct. After the user intervenes to clarify that the Java code should be checked directly, the agent decides to write a Java program with JML annotations for pre- and postconditions. It then searches and downloads the OPENJML tool and installs its prerequisites. Ultimately, after trying several configurations, the agent manages to correctly verify the annotated Java program.

With skills, the agent foresees that it wants to verify the code after generation. It loads the skill FM-TOOLS-CONVENTIONS to create a Java program that already uses the correct constructs to introduce non-deterministic inputs to the algorithm (to verify the method for all possible inputs) and to specify pre- and postconditions. It then proceeds by writing a normal Java program. The agent

then loads the skills `RUN-FM-WECK` and `COMPETITION-DATA` to select a suitable verifier (in our example, SWAT [36]) and to run it on the prepared program. The verifier correctly verifies that the Java program fulfills the specification (consisting of calls to `verifierAssert()`).

Research Assistant. A researcher wants to find related work on test generation for C programs. They are interested in literature on the best test generators in the last iteration of Test-Comp. They are also interested in the names of the maintainers and literature about the tools. Without the skills, the agent performs a broad search across the web. The answer is partially correct, but also partially outdated and incomplete. With the skills, the agent loads the skill `COMPETITION-DATA` to get information from Test-Comp, and then the skill `FM-TOOLS-DATA` to get reliable information about the tools.

As both approaches yield at least similar results, we can again compare the cost of the approach without skills and with skills. In our example, the attempt without skills uses about 220 k tokens of input and 9.4 k tokens of output (total cost of \$0.61). The attempt with skills uses about 88 k tokens of input and 12 k tokens of output (total cost of \$0.45).

5 Conclusion

We equipped AI agents with the ability to access, select, and execute formal-methods tools without manual setup. To this end, we developed agent skills that expose the structured data of the `FM-TOOLS CATALOG` and the competition results of `SV-COMP` and `Test-Comp` to AI agents, and invoke tools via `FM-WECK` in an isolated container environment. The skills enable data-driven tool selection and reduce token consumption compared to agents scraping HTML pages.

We plan to add explicit support for `SAT-COMP` [33] in future work. We hope that lowering the barrier to using formal-methods tools benefits both AI agents and the human engineers who rely on them.

Data-Availability Statement. All components that are required to locally run our examples are available in a reproduction package [34] and on the supplementary web page at <https://www.sosy-lab.org/research/fm-tools>.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY) and 418257054 (COOP).

Acknowledgements. We thank the formal-methods community for their contributions to the catalog of formal-methods tools `FM-Tools`.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. 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

2. 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
3. Beyer, D.: Find, use, and conserve tools for formal methods. In: Proc. Festschrift Podelski 65th Birthday. pp. 75–91. LNCS 14765, Springer (2026). https://doi.org/10.1007/978-3-032-13711-1_5
4. 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
5. 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). https://doi.org/10.1007/978-3-031-90660-2_9
6. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026). In: Proc. TACAS (2). pp. 461–502. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_23
7. Beyer, D.: Automatic testing of C programs: Test-Comp 2024. In: TBA. Springer (2024)
8. Beyer, D.: Advances in automatic software testing: Test-Comp 2025. In: Proc. FASE. pp. 257–274. LNCS 15693, Springer (2025). https://doi.org/10.1007/978-3-031-90900-9_13
9. Beyer, D.: Evaluating tools for automatic software testing (Report on Test-Comp 2026). In: Proc. FASE. pp. 449–468. LNCS 16504, Springer (2026). https://doi.org/10.1007/978-3-032-22774-4_23
10. Beyer, D., Wachowitz, H.: FM-WECK: Containerized execution of formal-methods tools. In: Proc. FM. pp. 39–47. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_3
11. Spector, A.Z.: Invited talk: Modular architectures for distributed and databases systems. In: Proc. PODS. pp. 217–224. ACM (1989). <https://doi.org/10.1145/73721.73743>
12. Giunchiglia, E., Narizzano, M., Tacchella, A., Vardi, M.Y.: Towards an efficient library for SAT: A manifesto. *Electronic Notes in Discrete Mathematics* **9**, 290–310 (2001). [https://doi.org/10.1016/S1571-0653\(04\)00329-4](https://doi.org/10.1016/S1571-0653(04)00329-4)
13. Shankar, N.: Little engines of proof. In: Proc. FME. pp. 1–20. LNCS 2391, Springer (2002). https://doi.org/10.1007/3-540-45614-7_1
14. Lampson, B.: Software components: Only the giants survive. In: *Computer Systems. Monographs in Computer Science*. pp. 137–145. Springer (2004). https://doi.org/10.1007/0-387-21821-1_21
15. Beyer, D., Kanav, S.: COVERTEAM: 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
16. 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>
17. Margaria, T., Nagel, R., Steffen, B.: JETI: A tool for remote tool integration. In: Proc. TACAS. pp. 557–562. LNCS 3440, Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_38
18. Margaria, T., Nagel, R., Steffen, B.: Remote integration and coordination of verification tools in JETI. In: Proc. ECBS. pp. 431–436 (2005). <https://doi.org/10.1109/ECBS.2005.59>

19. Margaria, T.: Web services-based tool-integration in the ETI platform. *Software and Systems Modeling* **4**(2), 141–156 (2005). <https://doi.org/10.1007/s10270-004-0072-z>
20. Steffen, B.: The physics of software tools: SWOT analysis and vision. *Int. J. Softw. Tools Technol. Transf.* **19**(1), 1–7 (2017). <https://doi.org/10.1007/s10009-016-0446-x>
21. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPACHECKER 3.0: Tutorial and user guide. In: *Proc. FM*. pp. 543–570. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_30
22. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: *Proc. CAV*. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
23. Crhová, J., Krčál, P., Strejček, J., Šafránek, D., Šimeček, P.: YAHODA: Verification tools database. In: *Proc. Tools Day*. pp. 99–103. FI MU Report Series FIMU-RS-2002-05, Masaryk University (2002)
24. Crhová, J., Krčál, P., Strejček, J., Šafránek, D., Simecek, P.: YAHODA: Verification tools database. <https://web.archive.org/web/20111119200847/http://anna.fi.muni.cz/yahoda> (2002)
25. Lathouwers, S., Zaytsev, V.: Modelling program-verification tools for software engineers. In: *Proc. MODELS*. pp. 98–108. ACM (2022). <https://doi.org/10.1145/3550355.3552426>
26. Lathouwers, S., Zaytsev, V.: Proverb: Dataset of tools and formats for program verification. Zenodo (2024). <https://doi.org/10.5281/zenodo.10806218>
27. Community, C.: A catalog of tools for the quantitative zoo. <http://cadp.inria.fr/resources/zoo/> (2024), accessed: 2026-05-14
28. on Formal Proofs, E.R.N.: EuroProofNet Automated Theorem Prover Wiki. <https://github.com/EuroProofNet/ATP/wiki> (2026), accessed: 2026-05-14
29. 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>
30. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the Evidential Tool Bus. In: *Proc. VMCAL*. pp. 275–294. LNCS 7737, Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_18
31. Bentele, M., Barth, M., Ebbinghaus, M., Körner, J., Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: ULTIMATE AUTOMIZER with a one-dimensional memory model (competition contribution). In: *Proc. TACAS* (2). pp. 589–594. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_37
32. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FUSEBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. In: *Proc. TAP*. pp. 85–105. Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6
33. Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M.: SAT competition 2020. *Artif. Intell.* **301**, 103572:1–103572:25 (2021). <https://doi.org/10.1016/j.artint.2021.103572>
34. Beyer, D., Lemberger, T., Wachowitz, H.: Conversation records for the ISOLA 26 paper: FM meets AI: Equipping AI agents with formal-methods tools. Zenodo (2026). <https://doi.org/10.5281/zenodo.20319312>

35. Peringer, P., Šoková, V., Vojnar, T.: PREDATORHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Proc. TACAS (2). pp. 408–412. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_30
36. Loose, N., Sieck, F., Mächtle, F., Eisenbarth, T.: SWAT: Improvements to the symbolic executor (competition contribution). In: Proc. TACAS (2). pp. 577–582. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_35

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

