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. We demonstrate the approach on five use cases and measure the token and cost benefits in two of them.

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 happened so far. Formal-methods tools have been out of reach for AI agents and non-expert developers alike: many tools require specific runtime environments (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.

> The use of software has entered a new era: software that was classically written for human users should now be discoverable and usable by AI agents as well, effectively and efficiently. Some software will be written for AI agents alone.

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 using 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 skills 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.
- We supply AI agents with the encoding conventions of SV-COMP and Test-Comp, so that they generate C and Java code in a form that verifiers accept.

Providing structured data directly reduces input-token consumption compared to agents scraping and parsing web pages, potentially saving cost and energy. Figure 1 contrasts the resulting setup with the classic way of using a verification tool.

Related Work. The FM-Tools Catalog and its accompanying command-line interface FM-Weck received significant attention [3, 6, 9] 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

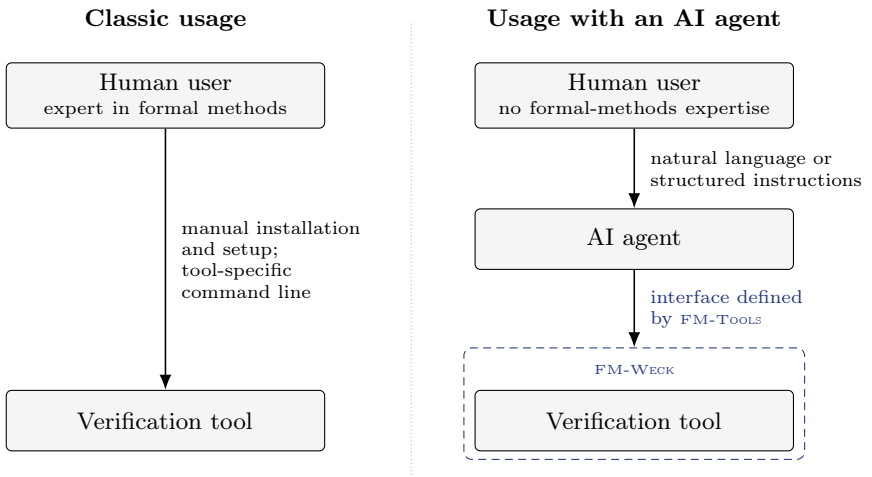

Fig. 1: Accessing a verification tool in the classic way (left) versus with an AI agent, FM-Tools, and FM-Weck (right); FM-Tools defines the interface between agent and tool, and FM-Weck encapsulates the tool for installation and execution

(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], the EuroProofNet ATP Wiki [28], and the FME Tool Catalogue [29]. The CoFI initiative [30] pursued a related goal in the context of algebraic specification: establishing a common framework and language shared across tools. 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 [31] 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, (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, 32]. Automatic software verifiers use SMT solvers to solve complex boolean formulas (e.g., [21, 33]) and fuzzing-based test-case generators may incorporate informed symbolic solvers to guide their fuzzing seeds (e.g., [34]). As a natural extension, the community also wanted a way to use more complex formal-methods tools as off-the-shelf components. This works in part because the communities often share a set of standards: automatic software verifiers support standard properties (specifications) for C programs, e.g., `unreach-call` to verify the (un-)reachability of an undesired program location or `valid-memsafety` to check memory safety of a program, and SAT solvers always expect a CNF formula as DIMACS input with the implied property of satisfiability. As a consequence, there were several attempts at combining formal-methods tools off-the-shelf [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 contain 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 Node's `npx`: 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 that the agent can load into its context. A separate description (defined in the file's front matter) provides guiding information that helps the agent decide when to load the skill. (2) An *MCP server* (Model Context Protocol) exposes external services to an AI agent through a typed JSON-RPC interface. 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

We present four skills to work with formal-methods tools: (1) Skill FM-TOOLS-DATA to retrieve information about tools, (2) skill COMPETITION-DATA to leverage the data from SV-COMP and Test-Comp to choose the right tool for a given task, (3) skill RUN-FM-WECK to run a tool and check its output, and (4) skill FM-TOOLS-CONVENTIONS to guide an agent how to set up code for verification. We choose skills over an MCP server because skills are more lightweight, easier to provide, extend, and maintain, and easier to install for users. Skills (1)–(3) are language agnostic. Skill (4) covers code for C and Java programs; in the examples below, we use C programs.

We complement the skills with `llms.txt` files on the FM-Tools, SV-COMP, and Test-Comp websites. These files provide AI agents guidance on how to access the data on these websites efficiently. Figure 2 gives an overview: each skill answers one question that the agent has to settle before it can report a result.

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://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
```

Access to Data on Formal-Methods Tools. The skill FM-TOOLS-DATA gives agents access to the information of formal-methods tools maintained in the FM-Tools Catalog. We design the description of the skill (Listing 1) so that it is likely to trigger whenever the user asks about information about a specific formal-methods tool.

Once the skill is loaded, it provides the agent with step-by-step instructions for 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. 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 (a YAML file) and fetch it. If the user asks

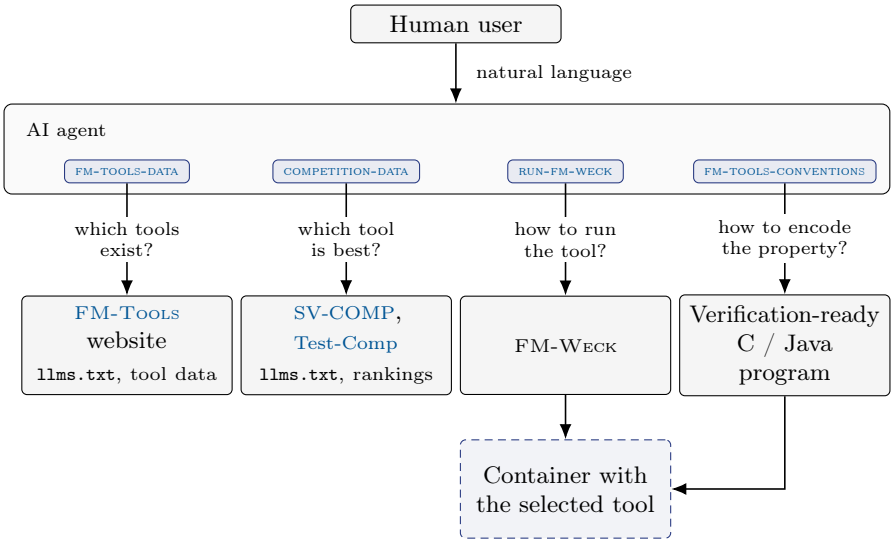

Fig. 2: Overview of the agent infrastructure; the four skills are loaded into the context of the AI agent, and each of them connects the agent to the data or the execution engine that answers one question

```
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

for 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 (findable, accessible, interoperable, reusable) [35].

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. We help agents that access the official competition websites to read the main results more efficiently by providing agent-focused data representations and the skill competition-data.

On the competition websites, we provide (a) a single summary markdown file that explains the category naming and lists the top three participants of each category (according to score), and (b) CSV versions of the existing HTML results tables. We provide an `llms.txt` on the respective competition's website that points to these new files. The `llms.txt` also point to the respective competition's rules and task definitions in case the agent requires more detailed information.

In addition to that we provide the skill competition-data. 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 in that category.

Executing Formal-Methods Tools. The 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 standard property (e.g., reachability or memory safety) of a C program, (3) the user asks to check a formula in DIMACS CNF format for satisfiability, (4) the user asks to list available formal-methods tools, or (5) 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

```
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) or SAT solver (kissat, CaDiCaL, or
↪ similar) on a program or formula, asks which tools or versions fm-weck
↪ supports, asks to verify a program without naming a specific tool, or
↪ asks to check, solve, or verify a SAT formula or .cnf / DIMACS file.
↪ For software verification without a named tool, use the
↪ competition-data skill to 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

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.
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 five exemplary use cases: (1) getting information on a specific tool, (2) checking a C program for memory safety, (3) checking a SAT formula for satisfiability, (4) checking auto-generated code, and (5) 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 with model Sonnet 4.6 as AI agent, FM-Tools in version 2.4 [36] as tools catalogue, and FM-Weck in version 1.7.2 [37] as execution engine. The full conversations are available on Zenodo [38] 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

```
> 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/[...]/summary.md)
  \  Received 6.9KB (200 OK)
* The most fitting category for a doubly-linked circular list is
  C.valid-memsafety.LinkedLists, 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 (recorded with pre-release version 1.7.2.dev0 of FM-WECK)

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 the latest tool version and both maintainers from the FM-TOOLS CATALOG.

As both the attempt without the skills and the attempt with the skills are able to provide the correct information, this is a good use case to anecdotally compare the cost of the two approaches. In our example, the attempt without the skills 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). We observed this difference in token use across multiple executions, which shows that providing additional, correct information can significantly reduce token use. Note that, since frontier models and agent harnesses change at a high pace, we do not claim that our skills always reduce token count. The main goal of the provided skills remains guiding agents to find and use information *correctly*.

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 skills (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 skill 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 [39], 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 a SAT Formula for Satisfiability. A user has a DIMACS CNF file that contains a logical formula and wants to check whether it is satisfiable.

Without the skill, the agent searches the system for an installed SAT solver; finding none, it checks the formula on its own by enumerating all variable assignments in a short Python script. For the small formula that we used in our example, the enumeration returned the correct answer quickly. But this relies on the reasoning of the agent and an inefficient method, instead of a dedicated solver. The result of the Python script also does not provide a proof certificate that can be validated independently.

With the skills available, the agent uses FM-WECK to list the available SAT solvers and decides to run KISSAT [40] on the file. It reports the same answer, but backed by a reliable SAT solver and a proof certificate.

The user could also use the agent to transform a natural-language problem into SAT problem encoded as DIMACS CNF and then check it with our skill; we consider this more elaborate use case out of scope for this paper, because transforming a natural-language problem into a logic formula is a research topic on its own [41, 42, 43, 44].

Checking Auto-Generated Code. A user generates an algorithm in Java with the help of an AI agent and 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 should 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 [45]) 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 and is 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 about tool performance from Test-Comp, and then the skill FM-TOOLS-DATA to get reliable information about the best-performing 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). The output token count may increase when our skills are used because the skills tell the agent the exact locations where to find information. This replaces a small set of generic web searches with a higher number of more precise, focused web fetches. Each web fetch is requested by the LLM separately and each of these requests counts towards the output tokens.

Lessons Learned. Across the five use cases, we observed four recurring patterns.

First, the skills rarely change the answer—they change how the answer is obtained. Without the skills, the agent reached a correct result in four of the five use cases, but by means that carry no guarantee: a dynamic memory check that cannot establish the absence of memory errors, an enumeration of variable assignments that yields no proof certificate, and web searches whose result was partly outdated. The gain from a formal-methods tool is not a different answer, but an answer whose validity the user can argue about.

Second, an agent selects tools by availability, not by suitability. In the memory-safety and in the SAT use case, the agent first scanned the local system for an installed tool and improvised as soon as that scan came back empty. A tool that is not installed is not part of the option space. On-demand containerized execution through FM-Weck is therefore a prerequisite for tool selection, not a convenience.

Third, a catalog alone does not let an agent choose. Faced with more than 60 verifiers, the agent needs a criterion. The category structure of SV-COMP and Test-Comp supplies one: in the memory-safety use case, the agent mapped a program with linked lists to category `C.valid-memsafety.LinkedLists` and ran the top-ranked tool of that category. Competition categories, designed to compare tools, also serve as a decision procedure for an agent.

Fourth, guidance before code generation pays more than guidance at the tool call. The clearest difference appeared in the use case on auto-generated code, where fm-tools-conventions led the agent to write a program in a verifiable form from the start, instead of generating code first and retrofitting annotations afterwards.

Our two cost measurements agree in direction: the input tokens drop, because the agent stops searching for information it can look up, while the output tokens can rise, because precise pointers produce more and smaller requests. The total cost fell in both cases.

Limitations. Our preliminary evaluation is currently limited to the five use cases and the single model of Claude Code, and we have not yet integrated witness validation [46, 47] of the results to ensure correctness of verification results.

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. Agents can also run SAT solvers such as Kissat through FM-Weck.

We hope that lowering the barrier to using formal-methods tools benefits both AI agents and the human engineers who rely on them. Formal-methods tools are one instance of a broader shift: software that was written for human users has to become discoverable and usable by AI agents as well.

Data-Availability Statement. All components that are required to locally run our examples are available in a reproduction package [38] and on the supplementary web page at `https://www.sosy-lab.org/research/fm-tools`. The version repositories of FM-Tools and FM-Weck are available on GitLab at `https://gitlab.com/sosy-lab/benchmarking/fm-tools` and `https://gitlab.com/sosy-lab/software/fm-weck`, and snapshots of the specific versions we used in our work are archived on Zenodo [36, 37]. We used assistance from Claude Code to prepare the material, all of which is available above.

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
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.W.: Software components: Only the giants survive. In: Computer Systems. pp. 137–145. Monographs in Computer Science, Springer (2004). https://doi.org/10.1007/0-387-21821-1_21
15. 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
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., Šimeček, 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/10.4121/20347950.v1
27. CADP Community: A catalog of tools for the quantitative zoo. http://cadp.inria.fr/resources/zoo/ (2024), accessed: 2026-05-14
28. EuroProofNet: EuroProofNet automated theorem prover wiki. https://github.com/EuroProofNet/ATP/wiki (2026), accessed: 2026-05-14
29. FME Industry Committee: FME tool catalogue. https://www.fmeurope.org/industry/tool-support/ (2026), accessed: 2026-07-01
30. Mosses, P.D.: CoFI: The common framework initiative for algebraic specification and development. In: Proc. TAPSOFT. LNCS, vol. 1214, pp. 115–137. Springer (1997). https://doi.org/10.1007/BFb0030591
31. 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
32. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the Evidential Tool Bus. In: Proc. VMCAI. pp. 275–294. LNCS 7737, Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_18
33. 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
34. 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. LNCS 12740, Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6

35. Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.W., d. Silva Santos, L.B., Bourne, P.E., Bouwman, J., Brookes, A.J., Clark, T., Crosas, M., Dillo, I., Dumon, O., Edmunds, S., Evelo, C.T., Finkers, R., Gonzalez-Beltran, A., Gray, A.J.G., Groth, P., Goble, C., Grethe, J.S., Heringa, J., 't Hoen, P.A.C., Hooft, R., Kuhn, T., Kok, R., Kok, J., Lusher, S.J., Martone, M.E., Mons, A., Packer, A.L., Persson, B., Rocca-Serra, P., Roos, M., v. Schaik, R., Sansone, S.A., Schultes, E., Sengstag, T., Slater, T., Strawn, G., Swertz, M.A., Thompson, M., v. d. Lei, J., v. Mulligen, E., Velterop, J., Waagmeester, A., Wittenburg, P., Wolstencroft, K., Zhao, J., Mons, B.: The FAIR guiding principles for scientific data management and stewardship. Sci. Data **3** (2016). https://doi.org/10.1038/sdata.2016.18
36. Beyer, D.: FM-Tools release 2.4: Data set of metadata about tools for formal methods. Zenodo (2026). https://doi.org/10.5281/zenodo.21829426
37. Beyer, D., Wachowitz, H.: FM-Weck Release 1.7.2. Zenodo (2026). https://doi.org/10.5281/zenodo.20533401
38. Beyer, D., Lemberger, T., Wachowitz, H.: Conversation records for the ISOLA 2026 article 'FM meets AI: Equipping AI agents with formal-methods tools'. Zenodo (2026). https://doi.org/10.5281/zenodo.21344019
39. 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
40. Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., Pollitt, F.: CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In: Proc. SAT Competition 2024. Department of Computer Science Report Series B, vol. B-2024-1, pp. 8–10. University of Helsinki (2024)
41. Han, S., Schoelkopf, H., Zhao, Y., Qi, Z., Riddell, M., Zhou, W., Coady, J., Peng, D., Qiao, Y., Benson, L., Sun, L., Wardle-Solano, A., Szabó, H., Zubova, E., Burtell, M., Fan, J., Liu, Y., Wong, B., Sailor, M., Ni, A., Nan, L., Kasai, J., Yu, T., Zhang, R., Fabbri, A.R., Kryscinski, W.M., Yavuz, S., Liu, Y., Lin, X.V., Joty, S., Zhou, Y., Xiong, C., Ying, R., Cohan, A., Radev, D.: FOLIO: Natural language reasoning with first-order logic. In: Proc. EMNLP. pp. 22017–22031. ACL (2024). https://doi.org/10.18653/v1/2024.emnlp-main.1229
42. Hahn, C., Schmitt, F., Tillman, J.J., Metzger, N., Siber, J., Finkbeiner, B.: Formal specifications from natural language. arXiv/CoRR **2206**(01962) (June 2022). https://doi.org/10.48550/arXiv.2206.01962
43. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In: Proc. CAV (2). pp. 383–396. LNCS 13965, Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_18
44. Yang, Y., Xiong, S., Payani, A., Shareghi, E., Fekri, F.: Harnessing the power of large language models for natural language to first-order logic translation. In: Proc. ACL (1). pp. 6942–6959. Association for Computational Linguistics (2024). https://doi.org/10.18653/v1/2024.acl-long.375
45. 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
46. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867

47. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. pp. 184–203. LNCS 14624, Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_11