CoVeriTeam GUI: A No-Code Approach to Cooperative Software Verification

Thomas Lemberger
LMU Munich
Munich, Germany

Henrik Wachowitz
LMU Munich
Munich, Germany

Abstract

We present CoVeriTeam GUI, a No-Code web frontend to compose new software-verification workflows from existing analysis techniques. Verification approaches stopped relying on single techniques years ago, and instead combine selections that complement each other well. So far, such combinations were—under high implementation and maintenance cost—glued together with proprietary code. Now, CoVeriTeam GUI enables users to build new verification workflows without programming. Verification techniques can be combined through various *composition operators* in a drag-and-drop fashion directly in the browser, and an integration with a remote service allows to execute the built workflows with the click of a button. CoVeriTeam GUI is available open source under Apache 2.0:
https://gitlab.com/sosy-lab/software/coveriteam-gui
Demonstration video: https://youtu.be/oZoOARuIOuA

ACM Reference Format:
Thomas Lemberger and Henrik Wachowitz. 2024. CoVeriTeam GUI: A No-Code Approach to Cooperative Software Verification. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3691620.3695366

1 Introduction

Compared to testing, formal software verification can not only show the presence of bugs, but also prove software safe with regards to given properties. But no single verification technique can solve all verification tasks. Because of this, many verification approaches stopped relying on a single technique: The International Competition on Software Verification [2] (SV-COMP) provides a yearly snapshot of the state of the art in automated software verification. In SV-COMP 2024 [2], 12 of the 26 submitted tools and two of the three medalists in category Overall combined different verification techniques.

So far, these combinations are—under high implementation and maintenance cost—redundantly implemented and glued together. Cooperative verification [11] addresses this and proposes to combine existing verification tools off-the-shelf. While this sacrifices some potential performance optimizations, it avoids the high costs of (re-)implementation and still produces competitive results [6].

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695366

But cooperative verification is an abstract idea that does not propose a specific implementation or framework to build tool combinations. The CoVeriTeam [5] project tries to fill this gap with three features: (1) It gives researchers access to a plethora of verification tools through its integration with the community-maintained fm-tools [1] repository. (2) It provides a domain-specific language to define verification workflows over these tools, and (3) it provides a runtime to execute defined workflows locally.

On the downside, CoVeriTeam still requires researchers to learn its domain-specific language, which includes unintuitive quirks and expert-centered documentation (as often seen in research projects).

CoVeriTeam GUI lightens this burden. It is a web-based graphical user interface that enables users to build verification workflows in a drag-and-drop fashion without knowing the details of the CoVeriTeam language. These workflows can then also be executed on a remote service [8] with the click of a button.

To stay in sync with new versions of CoVeriTeam, CoVeriTeam GUI automatically synthesizes the available actors and artifacts from the CoVeriTeam source code.

Contributions. This paper provides the following contributions:

(1) *No-Code Approach to Cooperative Verification:* CoVeriTeam GUI provides a no-code solution to build verification workflows with a plethora of available verification tools.
(2) *Visualization:* CoVeriTeam GUI can import and visualize workflows that were built with the CoVeriTeam language. This helps with understanding existing workflows.
(3) *Automated Synthesis:* CoVeriTeam GUI automatically synthesizes the available GUI components from the CoVeriTeam source code. This makes it easy to keep CoVeriTeam GUI up-to-date.
(4) *Reuse:* CoVeriTeam GUI is available open source on GitLab [10]. It is implemented as a pure frontend application and can be deployed easily.

Running Example. When an automated verifier produces a verification verdict, it is common to increase the confidence in this verdict with a third-party validator [3]. In this setting, the verifier receives as inputs a program and a specification, and produces as outputs a verification verdict and a witness. The verification verdict is 'TRUE' if the program is claimed correct with respect to the specification, and 'FALSE' if the program is claimed to violate the specification. The witness contains machine-readable data that tries to support the respective verdict.

The third-party validator then receives as input the program, specification, and witness, and tries to confirm the verification verdict with the help of the witness. There are two categories of validators, with different capabilities: If we want to check verdict 'TRUE' we must run a *correctness validator*. To check verdict 'FALSE' we must run a *violation validator*.

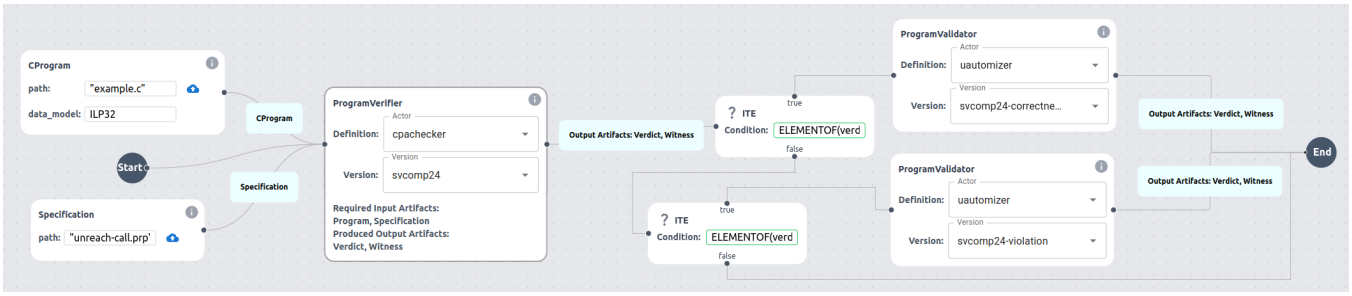

Figure 1: Composition of a validating verifier in CoVeriTeam GUI

```
class ProgramVerifier(Verifier, AtomicActor):
    _input_artifacts = {
      "program": Program, "spec": Specification
    }
    _output_artifacts = {
      "verdict": Verdict, "witness": Witness
    }
    ...
```

Figure 2: Excerpt of the implementation of the actor `ProgramVerifier` in CoVeriTeam

In a traditional setting we have to first run the verifier and then manually invoke the correct type of validator afterwards. As running example we avoid this work and directly combine a verifier and validator in a sequence that reflects the workflow described above. We call the resulting approach a *validating verifier* [5]. Figure 1 shows this composition in CoVeriTeam GUI.

2 Background: CoVeriTeam

The CoVeriTeam language is a domain specific language to build new workflows from existing automated verification techniques. A CoVeriTeam program consists of three parts: (1) Inputs to the program, (2) the definition of the verification workflow, and (3) the execution of the workflow.

Artifacts. All data that is used in a verification workflow is represented as an *artifact*. Artifacts are distinguished by their concrete type. Three examples: A *CProgram* artifact is a C program, a *Specification* artifact is a specification file, and a *Verdict* artifact is one of three string values that represent a verification verdict: 'TRUE', 'FALSE', and 'UNKNOWN'. Each concrete artifact that is used in a verification workflow also has a unique name that is set upon creation.

Atomic Actors. Existing verification techniques are represented by *atomic actors*. An atomic actor defines its expected input artifacts and output artifacts (both by name and type), and calls an external tool with the input artifacts to produce the output artifacts. Within CoVeriTeam, different types of atomic actors exist for different pairs of input- and output-artifacts. Figure 2 shows an excerpt of the internal implementation of the atomic actor `ProgramVerifier`, a standalone software verifier that receives a program and a specification and produces a verification verdict and a verification-result witness. If an actor does not receive all required input artifacts, it will not execute and CoVeriTeam will issue an appropriate error message.

Composition Operators. Composition operators combine atomic actors. CoVeriTeam supports the sequential composition, parallel composition, cyclic composition and if-then-else composition of

```
1  // Create verifier and validator from yml files
2  verifier = ActorFactory.create(ProgramVerifier,
     ↪ verifier_path, verifier_version);
3  validatorC = ActorFactory.create(ProgramValidator,
     ↪ validator_path, version_correct);
4  validatorV = ActorFactory.create(ProgramValidator,
     ↪ validator_path, version_violation);
5  // Use validatorC if verdict is true
6  // or validatorV if verdict is false
7  condition_true = ELEMENTOF(verdict, {TRUE});
8  condition_false = ELEMENTOF(verdict, {FALSE});
9  false_branch = ITE(condition_false, validatorV);
10 second_component = ITE(condition_true, validatorC,
     ↪ false_branch);
11 // Verifier and second component to be executed in
     ↪ sequence
12 validating_verifier = SEQUENCE(verifier,
     ↪ second_component);
13
14 // Inputs, received from the command line
15 program = ArtifactFactory.create(CProgram,
     ↪ program_path, data_model);
16 specification =
     ↪ ArtifactFactory.create(BehaviorSpecification,
     ↪ spec_path);
17 inputs = {'program':program, 'spec':specification};
18 // Execute the new component on the inputs
19 res = execute(validating_verifier, inputs);
20 print(res);
```

Figure 3: A validating verifier as a CoVeriTeam program

actors. Compositions can be arbitrarily nested. The if-then-else `ITE(c,A1,A2)` takes three arguments: a condition `c` over incoming artifacts, the actor `A1` that is executed when `c` is true, and the (optional) actor `A2` that is executed when `c` is false. The CoVeriTeam language supports instanceof, elementof, existence and equality checks. For example condition `ELEMENTOF(verdict, {TRUE, FALSE})` is true when the incoming verdict is either 'TRUE' or 'FALSE'.

Example CoVeriTeam Program. Figure 3 shows the definition of a validating verifier in the CoVeriTeam language. A program verifier (line 2) receives a program and a specification as input, and produces a verification verdict and a verification-result witness. It is combined in a sequence (line 12), with conditional validation as second component: If the produced verdict is true (line 10), it is checked by correctness validator `validatorC` (line 3). Otherwise, if the produced verdict is false (line 9), it is checked by violation validator `validatorV` (line 4). If the produced verdict is another value (unknown), there is no need to check the result and nothing happens after verification (no else-branch in the if-then-else in line 9). The input program and specification are defined in lines 15 and 16. Finally,

```
"actors": [
  {
    "name": "ProgramVerifier",
    "inputArtifacts": {
      "program": "Program",
      "spec": "Specification"
    },
    "outputArtifacts": {
      "verdict": "Verdict",
      "witness": "Witness"
    }
  },  /* .. snip other .. */
```

Figure 4: Excerpt of the synthesized JSON file

the composition is executed (lines 17–19) and the result is printed to the user. All undefined variables in a CoVeriTeam program must be provided upon execution with CoVeriTeam. Here this is 8 variables: `verifier_path`, `verifier_version`, `validator_path`, `version_correct`, `version_violation`, `program_path`, `data_model`, and `spec_path`.

We hope this example demonstrates how even a relatively simple sequential composition is complicated to write and execute with the CoVeriTeam language.

CoVeriTeam Service. Originally, verification workflows are executed through CoVeriTeam on the command line. CoVeriTeam Service [8] is an existing web service that allows to execute workflows remotely without a local installation of CoVeriTeam.

3 Contribution

CoVeriTeam GUI provides a no-code alternative to the text-based CoVeriTeam programs. It is implemented as a web application, where users can define a cooperative verification workflow visually and execute it remotely through CoVeriTeam Service [8].

3.1 Synthesizing the Frontend

A major concern in building CoVeriTeam GUI was the evolving nature of CoVeriTeam. Users and maintainers of CoVeriTeam add new atomic actors on a regular basis, for example to support tools with special output files like new witness formats or test suites. To keep CoVeriTeam GUI up-to-date with CoVeriTeam despite these frequent additions, we synthesize the atomic actors and artifacts directly from CoVeriTeam's source code.

This synthesizer is written in Python. It parses the CoVeriTeam code and extracts all classes that transitively inherit from class `AtomicActor`. The inputs and outputs of atomic actors are extracted from the class fields `_input_artifacts` and `_output_artifacts`. To extract the available artifacts, the synthesizer uses an analogous process. The full result of this analysis is processed into a JSON file that lists all atomic actors and artifact types. This can be consumed by the CoVeriTeam GUI to display the components accordingly. An excerpt of the synthesized JSON file is shown in Fig. 4.

3.2 No-Code Cooperative Verification

Figure 5 shows the initial state of CoVeriTeam GUI when a user creates a new composition. CoVeriTeam GUI visualizes the verification workflow on a canvas (area on the right in Fig. 5). For larger compositions that don't fit on the screen, a minimap on the bottom right helps the users navigate. Users can select compositions (box

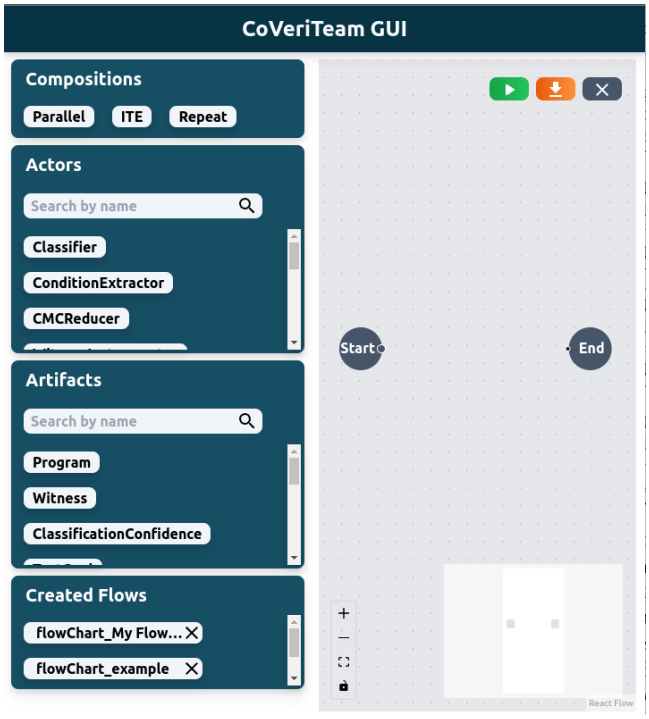

Figure 5: Full CoVeriTeam GUI with a blank composition

'Compositions') and atomic actors (box 'Actors') from the palette on the left to position them on the canvas.

The canvas lets users freely move and rearrange components. A sequential composition is created by connecting one block's outputs with another block's inputs. The edges are labeled with the artifacts that flow through them (see Fig. 1).

The atomic actors offer a drop-down menu where users can select from a large array of available verification tools and their versions. This removes one big source of unbound variables in text-based CoVeriTeam programs. CoVeriTeam GUI provides all tools that store their metadata in the fm-tools repository [1].

To provide inputs to the workflow, such as the program under analysis, users can select artifacts from the palette on the left. This places a matching artifact node on the canvas where users can upload the intended input file.

To execute their workflow users can click the green play button in the top-right corner. This executes the workflow remotely through CoVeriTeam Service. The output of the execution is displayed in a popup console window on the right side of the screen. When the execution finished, users can download the full output of CoVeriTeam as a `.zip` file. Users can also download their workflow as a CoVeriTeam program with a click on the orange download button on the top right. After the first execution or download, the workflow is stored in the left box 'Created Flows'.

Figure 1 shows how the running example of a validating verifier can be built in CoVeriTeam GUI. Notice how easy it is to process the data flow through the composition compared to Fig. 3, even though the view was optimized for space. For the verifier we have selected CPAchecker and for the validators two configurations of UAutomizer.

```
actor1 = ActorFactory.create(ProgramVerifier,
  ↪ "cpachecker.yml", "svcomp24");
actor2 = ActorFactory.create(ProgramValidator,
  ↪ "uautomizer.yml", "svcomp24-violation");
actor3 = ActorFactory.create(ProgramValidator,
  ↪ "uautomizer.yml", "svcomp24-correctness");

a_2 = ITE(ELEMENTOF(verdict, {FALSE}), actor2, );
a_1 = ITE(ELEMENTOF(verdict, {TRUE}), actor3, a_2);
a_0 = SEQUENCE(actor1, a_1);

artifact1 = ArtifactFactory.create(CProgram,
  ↪ "example.c", ILP32);
artifact2 = ArtifactFactory.create(Specification,
  ↪ "unreach-call.prp");

inputs = {'program': artifact1, 'spec': artifact2};

result = execute(a_0, inputs);
print(result);
```

Figure 6: CoVeriTeam program generated by CoVeriTeam GUI

3.3 Code Generation

Because CoVeriTeam GUI is implemented as a pure JavaScript frontend application, it generates a CoVeriTeam program from the visual representation of the workflow directly in the user's browser. To do this, it traverses the internal graph representation of the verification workflow. First, it generates the corresponding call to `ActorFactory` or `ArtifactFactory` for every atomic actor and input artifact. Then, the code for the compositions is generated by recursing through them bottom-up. First all children of a composition are generated and stored in variables, then the composition itself. For example, the workflow in Fig. 1 is translated into the CoVeriTeam program in Fig. 6 (snipped for brevity).

3.4 Deployment

CoVeriTeam GUI is implemented as a pure frontend application in JavaScript/React. We host an instance of CoVeriTeam GUI at coveriteam-service.sosy-lab.org/gui. CoVeriTeam GUI can also run locally or be hosted on your own web server through the packed HTML.

4 Related Work

Electronic Tools Integration (ETI) [17, 18] was a pioneering platform that offered verification tools to users through the internet. To use ETI, users had to learn the domain-specific language HLL. The Unite [19] project aims to make web hosting of verifiers easier. With only a few lines of configuration, developers can make their verifiers available through the web. Unite does not provide a graphical user interface to execute the tools, but a standardized API that enables integration into code editors. Multiple verification tools provide some form of accessibility through a web front-end [4, 13–15]. However, these web portals are tailored to single tools and do not offer the composition of multiple tools. No-Code solutions can be found in various other domains, such as blockchain [12] and education [16].

5 Conclusion

We presented CoVeriTeam GUI, a No-Code solution for cooperative verification. We showed how a user can assemble a validating verifier composition without prior knowledge of the CoVeriTeam language and execute it directly from the GUI.

Data-Availability Statement. We use CoVeriTeam GUI [10], version 1.0, CoVeriTeam [7], version 1.2.1, and CoVeriTeam Service [9], version 1.2. All three are open source under the Apache 2.0 license. A reproduction package for CoVeriTeam GUI is available on zenodo: 10.5281/zenodo.13757771. The GUI is available online for experimentation: https://coveriteam-service.sosy-lab.org/gui.

Acknowledgement. We thank Jasmin Krenz for help with the implementation of CoVeriTeam GUI as part of her bachelor thesis.

Funding Statement. This work was funded by the Deutsche Forschungsgesellschaft (DFG) — 378803395 (ConVeY).

References

[1] D. Beyer. 2024. Conservation and Accessibility of Tools for Formal Methods. In *Proc. Festschrift Podelski 65th Birthday*. Springer.
[2] D. Beyer. 2024. State of the Art in Software Verification and Witness Validation: SV-COMP 2024. In *Proc. TACAS (3) (LNCS 14572)*. Springer, 299–329. https://doi.org/10.1007/978-3-031-57256-2_15
[3] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig. 2022. Verification Witnesses. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 57:1–57:69. https://doi.org/10.1145/3477579
[4] D. Beyer, G. Dresler, and P. Wendler. 2014. Software Verification in the Google App-Engine Cloud. In *Proc. CAV (LNCS 8559)*. Springer, 327–333. https://doi.org/10.1007/978-3-319-08867-9_21
[5] D. Beyer and S. Kanav. 2022. CoVeriTeam: On-Demand Composition of Cooperative Verification Systems. In *Proc. TACAS (LNCS 13243)*. Springer, 561–579. https://doi.org/10.1007/978-3-030-99524-9_31
[6] D. Beyer, S. Kanav, and C. Richter. 2022. Construction of Verifier Combinations Based on Off-the-Shelf Verifiers. In *Proc. FASE*. Springer, 49–70. https://doi.org/10.1007/978-3-030-99429-7_3
[7] D. Beyer, S. Kanav, and H. Wachowitz. 2023. CoVeriTeam Release 1.0. Zenodo. https://doi.org/10.5281/zenodo.7635975
[8] D. Beyer, S. Kanav, and H. Wachowitz. 2023. CoVeriTeam Service: Verification as a Service. In *Proc. ICSE, companion*. IEEE, 21–25. https://doi.org/10.1109/ICSE-Companion58688.2023.00017
[9] D. Beyer, S. Kanav, and H. Wachowitz. 2024. CoVeriTeam Service Release 1.2. Zenodo. https://doi.org/10.5281/zenodo.12552828
[10] D. Beyer, T. Lemberger, and H. Wachowitz. 2024. Source-Code Repository of CoVeriTeam GUI. https://gitlab.com/sosy-lab/software/coveriteam-gui. Accessed: 2024-09-13.
[11] D. Beyer and H. Wehrheim. 2020. Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework. In *Proc. ISoLA (1) (LNCS 12476)*. Springer, 143–167. https://doi.org/10.1007/978-3-030-61362-4_8
[12] S. Curty, F. Härer, and H. Fill. 2023. Design of blockchain-based applications using model-driven engineering and low-code/no-code platforms: a structured literature review. *Softw. Syst. Model.* (2023), 1857–1895. https://doi.org/10.1007/S10270-023-01109-1
[13] Z. Esen and P. Rümmer. 2022. TriCera: Verifying C Programs Using the Theory of Heaps. In *Proc. FMCAD*. TU Wien Academic Press, 360–391. https://doi.org/10.34727/2022/isbn.978-3-85448-053-2
[14] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. 2013. Ultimate Automizer with SMTInterpol (Competition Contribution). In *Proc. TACAS (LNCS 7795)*. Springer, 641–643. https://doi.org/10.1007/978-3-642-36742-7_53
[15] N. Macedo, A. Cunha, J. Pereira, R. Carvalho, R. Silva, A. C. R. Paiva, M. S. Ramalho, and D. Silva. 2021. Experiences on Teaching Alloy with an Automated Assessment Platform. *Science of Computer Programming* 211 (2021), 102690. https://doi.org/10.1016/j.scico.2021.102690
[16] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* (2010), 1–15. https://doi.org/10.1145/1868358.1868363
[17] T. Margaria, R. Nagel, and B. Steffen. 2005. Remote integration and coordination of verification tools in JETI. In *Proc. ECBS*. 431–436. https://doi.org/10.1109/ECBS.2005.59
[18] Bernhard Steffen, Tiziana Margaria, and Volker Braun. 1997. The Electronic Tool Integration Platform: Concepts and Design. *STTT* 1, 1-2 (1997), 9–30. https://doi.org/10.1007/s100090050003
[19] O. Vašíček, J Fiedor, T. Kratochvíla, K. Bohuslav, A. Smrčka, and T. Vojnar. 2022. Unite: An Adapter for Transforming Analysis Tools to Web Services via OSLC. In *Proc. ESEC/FSE*. ACM. https://doi.org/10.1145/3540250.3558939